AFRL-RI-RS-TR-2019-164

DEVELOPING ALGORITHMS THAT LEAK OR EXPLODE IN COMPLEXITY (DALEC)

RAYTHEON BBN TECHNOLOGIES

AUGUST 2019

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2019-164 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
AMANDA P. OZANAM
Work Unit Manager

/ S /
JAMES S. PERRETTA
Deputy Chief, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUGUST 2019	FINAL TECHNICAL REPORT	APR 2015 – APR 2018

4. TITLE AND SUBTITLE

DEVELOPING ALGORITHMS THAT LEAK OR EXPLODE IN COMPLEXITY (DALEC)

5a. CONTRACT NUMBER
FA8750-15-C-0108

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Steven Jilcott
Stanislav Ponomarev

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
BB

5f. WORK UNIT NUMBER
NT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Raytheon BBN Technologies
10 Moulton Street
Cambridge, MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

DARPA/I2O
675 North Randolph St.
Arlington, VA 22203

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-164

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of the Space/Time Analysis for Cybersecurity (STAC) program is to develop new analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms, including vulnerabilities to algorithmic complexity and side channel attacks. As an adversarial challenge team, BBN develops challenge applications that illuminate the behavior of the R&D team tools, allowing not only measurement of their performance, but also diagnosis of their behavior.

15. SUBJECT TERMS

Space, Time, Complexity, Challenges, Applications

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**AMANDA P. OZANAM**
U	U	U	UU	242	19b. TELEPHONE NUMBER (*Include area code*) **(315) 330-4517**

TABLE OF CONTENTS

Section **Page**

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

1.0 SUMMARY

Raytheon BBN Technologies Corp. (BBN) and teammate Assured Information Security (AIS) completed the *Developing Algorithms that Leak and Explode in Complexity* (DALEC) project, an Adversarial Challenge effort under Technical Area (TA) 2. In this effort, the BBN team provided implementations of algorithms and applications that leverage them – called "challenge programs" – to exercise TA1 program analysis tools. The BBN team leveraged the Scientific Adversarial Challenge (SAC) process that was pioneered by BBN and used to create challenge applications for DARPA's Automated Program Analysis for Cybersecurity (APAC) program. BBN has demonstrated viability and scalability of this approach by creating 41 challenge applications over approximately three years, helping the R&D teams to significantly enhance their tools. SAC's key benefit is its hypothesis-driven approach to challenge design, where, instead of focusing on false or missed detections, each challenge intentionally differentiates a specific R&D Team capability and exposes strengths and weaknesses of an approach. Teammate AIS provided a practical background in the study and exploitation of side channel and complexity attacks in a variety of large-scale, real-world systems, including the exploitation of cache timing algorithms as covert channels and denial of service attacks on distributed data storage algorithms.

The DALEC team made it harder for TA1 program analysis tools to locate optimizing assumptions in the bytecode and understand their effects on algorithm behavior. DALEC tailored the program analysis difficulty by targeting challenge program implementations at static analysis questions that are known to be difficult to answer. The DALEC team crafted implementations of tuned algorithms with varying degrees of complexity for such program structures.

We found many generalizable ways to leak information or explode resource usage. For example, optimizing code to make common cases fast or abusing databases to slow runtime can leak potentially sensitive information. Algorithmic complexity attacks tended to be the easiest to introduce, often using research from some well-known algorithm, but also proved the easiest to detect. Of course, many side channels may be already present in an application without an adversary needing to introduce them.

A recurring theme on the STAC program was the presence of unintended vulnerabilities. In general, we attempted to guard against these but only by informally reasoning about code behavior. This perhaps best illustrates how challenging it may be for traditional software developers to mitigate this class of vulnerability since even security-aware developers could introduce them unintentionally. Developers may know to use memory-safe functions but have little motivation to understand the security implications of time and space. In fact, they may even be incentivized to optimize for time and space, which may introduce side channel vulnerabilities. Recent findings in hardware side-channels proved the usefulness of STAC-related efforts and highlighted a need for available ground truth about such vulnerabilities – data that the DALEC team provided.

2.0 INTRODUCTION

Due to the partitioning of DALEC's effort this document's structure is broken into per challenge application sections. As a result, introduction to each challenge, results, and discussion are split for each challenge. You will find write-up for each challenge in the following section.

This document includes the combined content of CDRLs A011, B011, and C011 generated under STAC phases 1, 2 and 3 respectively. Content generated for each CDRL under each program phase is as follows:

CDRL C011 / STAC phase 3:
4.1 Engagement 7 Challenge Programs
4.2 Engagement 6 Challenge Programs

CDRL B011 / STAC phase 2:
4.3 Engagement 5 Challenge Programs
4.4 Engagement 3 and 4 Challenge Programs

CDRL A011 / STAC phase 1:
4.5 Engagement 2 Challenges Programs
4.6 Engagement 1 Challenge Programs

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

All challenge development followed the following procedures, which are summarized in Figure 1.

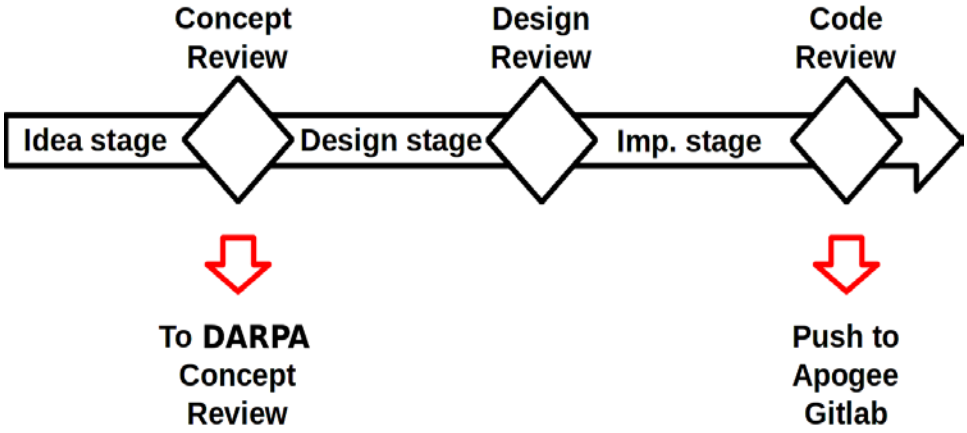


Figure 1. STAC Challenge Development Procedure Overview

3.1 Idea stage

Required documentation – Concept Document consisting of all of the following:

Digest of these questions for DARPA:

- What does the program do?
- What type of vulnerability? TC (Time Complexity), SC (Space Complexity), TSC (Time Side-Channel), SSC (Space Side-Channel)
- (If a side channel) What is the secret?
- Who is the victim?
- Who is the adversary?
    - Are there any special assumptions about who the adversary is or his level of access?
- What is the vulnerability?
    - What is the vulnerable algorithm and how does the vulnerability work?
    - Provide an example of how an input will trigger the vulnerable code path
- What operations can the adversary do?
    - Walk through a basic story of how the adversary triggers the complexity attack, OR
    - Walk through a basic story of a sequence of operations that allows an adversary to extract a secret

Specific digest questions for side channels:

- Is it noise free?
- Are there relatively few collisions?

- Will every adversary operation witness an observable symbol?
- Can the adversary learn the association between secret and observable?

Review Gate

Concept review. Informal document review by all team members (no meeting, by email). Then the concept document is forward to DARPA for his approval.

- Does the scenario for adversary and victim involve too many or unrealistic assumptions?
- Do the inputs and the vulnerability as described match up with the STAC operational definitions?

3.2 Design stage

Required documentation – Design Document consisting of all of the following:

Software design

- Architecture of app (provide a diagram of deployment and major components)
    - Use of libraries and whether they will be internal to the challenge java archive jar
- Design of input and output formats
    - Any guards on input size to constrain unintended vulnerabilities
- Key classes and key class members
    - Roadmap of class responsibilities, anticipated communications/events between classes
- Algorithm pseudocode, if applicable

Proof analysis (complexity)

- Identify the complexity parameter for the vulnerable algorithm (# nodes, length of list, etc.)
- Explain how the number of bytes in the input format maps to the complexity parameter for the algorithm – is the vulnerability to a single large input or to a sequence of special inputs?
- Analyze how common a "worst case input" is – provide an argument that bad inputs are relatively rare
- How will you construct a worst case input?
- How will you measure the time or space consumed?

Proof analysis (side channel)

- What is the proof strategy – how will your harness iterate over the adversary actions and extract a secret?
- What is the worst case secret value and justification for picking it?
- Analysis of expected number of operations in the worst case (an estimate of a budget). How do you plan to account for nondeterminism or noise and do you need empirical data to help you decide what the worst case is?

Review Gate

Design Review. Formal document review by all team members (with meeting to discuss)

- Is there enough information that the team members would be sufficiently oriented for a code review?
- Is the document clear and understandable enough for Apogee to understand the vulnerability, the proof, and how to construct questions and budgets?

3.3 Implementation stage

Required documentation

All Gitlab projects shall use the same folder structure as the delivery to Apogee.

- The description.txt, fulldescription.txt, APIsUsed.txt, budgets, and proofs folders must be populated. Documentation should be drawn from the Concept and Design docs.
- description.txt – includes the application description, and the operational context. Make sure this info is appropriate to the Blue Team and describes the adversary model without giving away the vulnerability. Also includes a "man page" or instructions for running/interacting with the app and sample inputs.
- fulldescription.txt – A complete description, including the description of the vulnerability and proof analysis.

Review Gate

Code Review. Formal review with meeting to discuss defects. Reviewers will look at:

- Are the documents included in the delivery above adequate and do they match up with the contents of the Concept and Design docs?
- Is there more than one vulnerability of the same type in the challenge program? Can you see the possibility of a second side channel, for instance?
- Are there any "tells" that might give away the location or character of the vulnerability to the Blue Team tools or team members?
- Does the proof execute on the NUC (Intel's Next Unit of Computing) and do the reviewers get results consistent with the claimed budget?

4.0 RESULTS AND DISCUSSION - CHALLENGE PROGRAM DESIGN

4.1 Engagement 7 Challenge Programs

4.1.1 PhoneMaster

4.1.1.1 Description

Phonemaster is a record keeping service for phone calls and bills. Users may create new accounts for tracking their phone calls and bills. After logging in, they may submit call information that is stored in the backend server. Additionally, users may also request information about specific calls and bills, such as duration or charge amounts. Lastly, users may request a list of all users in the Phonemaster system sorted by identifier (ID).

Phonemaster contains a **timing side channel vulnerability** where the secret is the real name of any user from the victim script that sends an encrypted request to Phonemaster.

Phonemaster's login function is vulnerable to a timing-side channel. When a user logs in, the time it takes to check their credentials is based on the order in which they signed up. User 1 will take the least amount of time, while the most recently signed up user will take the longest. The login time is therefore a function of the age of the account. The login function's timing variations will be achieved by first checking the accompanying credentials against the stored credentials for User 1, then those for User 2, and so on. Passwords will be repeatedly hashed enough times to make the side-channel usable.

A malicious user may create a user and login to Phonemaster and time the response. Repeating this process for each userlist index allows an attacker to build a table of average response times for each index. From then on, the malicious user may observe the timing between requests and responses for other users and apply the table to discover the user ID of the person making any given request. Finally, the malicious user may request a list of Phonemaster users (which are returned in the order they signed up) to effectively de-anonymize benign users, whose requests are encrypted while moving over the wire.

To make the side channel harder to exploit, the base response rate for the login request will be randomized across repeated runs. This effectively changes the added overhead in every run and will force any proof script to learn this value on the fly.

4.1.1.2 Software Design

4.1.1.2.1 Related Class Roadmap

Figure 2 depicts the PhoneMaster class diagram.

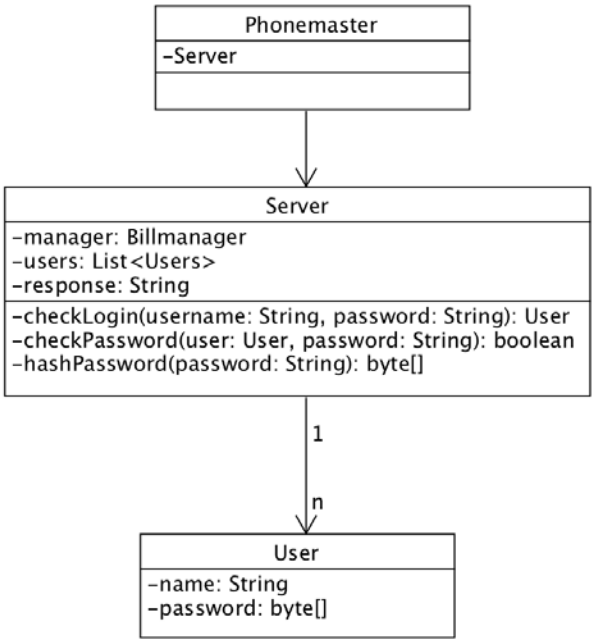


Figure 2. Phonemaster Class Diagram

4.1.1.2.2 Server Class

The **Server** class is responsible for interfacing between the user and all of Phonemaster's systems. First, it contains a Bill Manager that maintains a list of phone bills, adds calls to said bills, and retrieves them for printing. Secondly, Phonemaster tracks a list of users and validates all incoming requests. The vulnerability is located here, encapsulated in the checkLogin -> checkPassword -> hashPassword sequence. When a user sends a request to Phonemaster, it first checks to see if the password is in its userlist. If there is no match for the 1st user in Phonemaster's userlist, it repeats this process until it either finds a match or reaches the end of the list. In turn, the farther down the list of usernames a user is, the longer it will take to validate their request, effectively opening up a timing side-channel.

This side-channel is further obfuscated in the following way: the additional overhead is changed each run via a secure random number. This effectively forces dynamic analysis.

4.1.1.2.3 User Class

The **User** class simply tracks a name and a password hash.

4.1.1.2.4 Vulnerability code

Below, the checkLogin -> checkPassword -> hashPassword sequence is shown:

```
// check username and get game client
User CheckLogin(String name, String password) {

// Iterate over every user here:
for (User user : users) {
if (checkPassword(user, password)) {
return user;
}
}
// Otherwise, create a new user.
}
```

The above code illustrates how Phonemaster cycles through all usernames in its userlist until it finds a match. If the checkPassword function produces enough overhead, the time difference between logging in as the first registered user will be significantly faster than logging in as the last registered user.

```
boolean checkPassword(User user, String password) {
return (Arrays.equals(user.password, hashPassword(password)));
}

byte[] hashPassword(String password) {
MessageDigest md = null;
md = MessageDigest.getInstance("MD5");
md.update(password.getBytes());
byte[] digest = md.digest();
// Slowing operation here:
for (int i = 0; i < 50000; i++) {
md.update(digest);
digest = md.digest();
}
return digest;
}
```

The above code shows the additional overhead that is introduced to slow down the login function. Here, the password is hashed 50,000 times before it is checked against the password on file, which has also been hashed 50,000 times.

4.1.2 Thermomaster

4.1.2.1 Description

Thermomaster is a networked server/thermostat system for power plant temperature control and prediction, the server being the challenge program and thermostat being the user. The first part of the system, the thermostat network, reads temperatures from its environment and sends them to the backend server. In addition to sending their current readings, the thermostats can submit a setpoint temperature to the server. In turn, the server inputs the current temperature and setpoint into a modified proportional–integral–derivative (PID) algorithm to predict how the temperature

will change over time before sending its result back to the thermostat. Lastly, the thermostats may change their temperature metric (Fahrenheit to Celsius and vice versa) which is pushed to and persists in the backend server.

The second part of the Thermomaster system, the backend server, provides a few operations in addition to the aforementioned temperature feedback loop. First, in absence of thermostat feedback, the server continuously runs its PID algorithm on a temperature simulation to predict the thermostat readings. It also replies with the adjustments the thermostat should make to achieve its setpoint.

Thermomaster contains a **time complexity vulnerability.**

The Thermomaster server's PID algorithm is vulnerable to a time-complexity attack that arises from a weak guard around the subnormal range of floating point numbers. On modern architectures, floating point numbers are represented by a series of bits indicating a sign, significand, and exponent. According to the Institute of Electrical and Electronics Engineers (IEEE) 754 floating point standard [1], "normal" numbers have no leading 0's in the significand. Instead, any leading 0's are pushed into the exponent. For example, the number 0.0005 is represented as $5 * 10^{-4}$. However, since there is a limited number of exponent bits, and normal numbers may not contain any leading 0's, an underflow gap persists around zero. To address this, the standard introduces "subnormal" numbers that contain leading 0's in the significand, effectively adding precision around zero. This is illustrated in Figure 3.

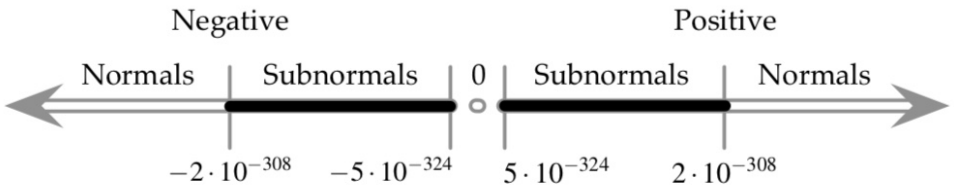


Figure 3. Normal and Subnormal 64-bit Floating Point Range

Here, subnormal values are interesting because they induce a performance hit when they are either used as an operand, or produced from an operation. Relative to normal numbers, subnormal numbers have been found to cause worst-case slowdowns up to a factor of 100^2. For confirmation, running a micro-benchmark [2] on the STAC reference platform NUCs produced a slowdown by a factor of ~10.

To protect against this slowdown, Thermomaster employs a guard around the subnormal range. If a thermostat sends a current temperature and setpoint whose error is subnormal, the server will throw out the input.

However, this guard is vulnerable as a subsequent "change metric" request will push a *nearly* subnormal error into the subnormal range, after it has already passed the guard. By issuing a change from Fahrenheit to Celsius, the error will be multiplied by 5/9ths, effectively nudging it into the

vulnerability triggering range. Once in the subnormal range, the server will suffer repeated slowdowns as it operates on subnormal values.

4.1.2.2 Software Design

4.1.2.2.1 Related Class Roadmap

Figure 4 depicts the ThermoMaster vulnerable class diagram.

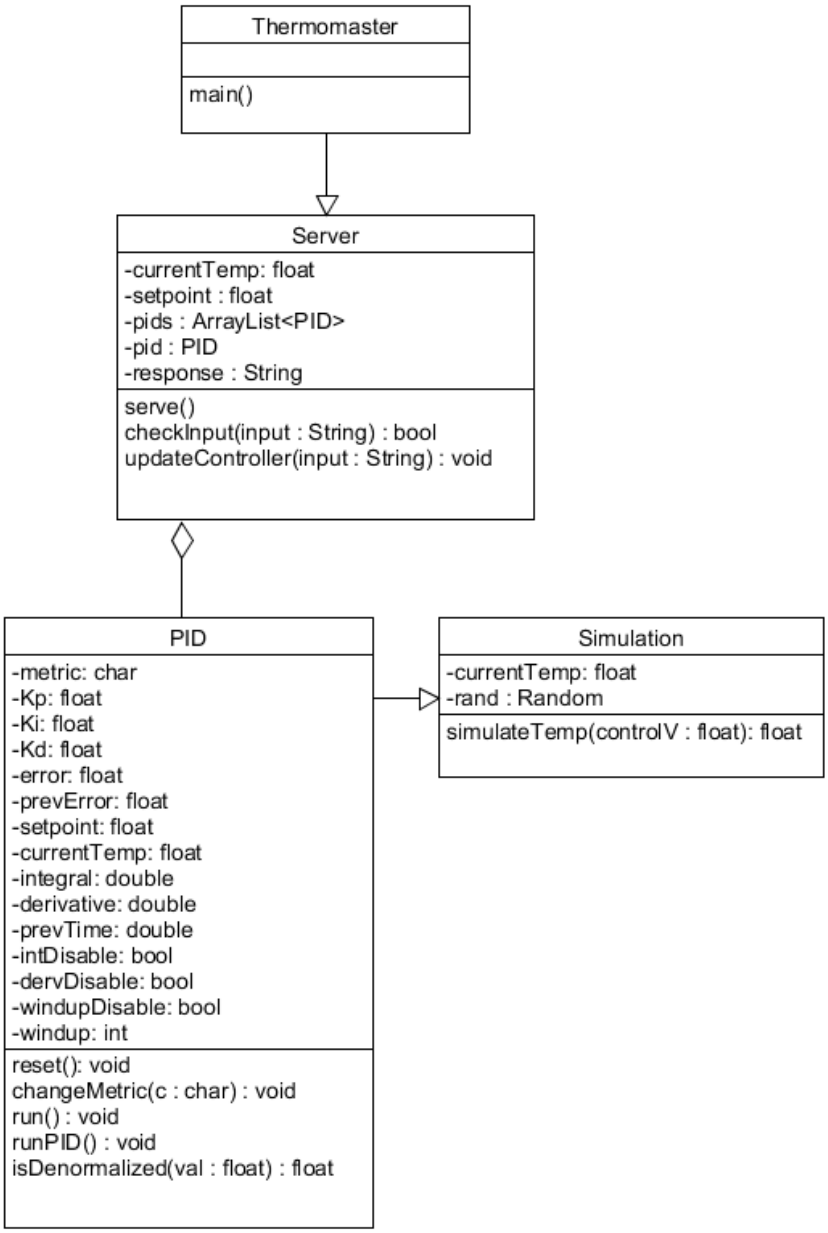


Figure 4. Thermomaster Vulnerable Class Diagram

Server class

The **Server** class is the responsible for managing input/output (I/O) between thermostats and the list of various PID controllers.

PID class

The **PID** class is responsible for running PID updates in Thermomaster.

Logger class

The **Logger** class is responsible for logging error information.

User class

The **User** class is responsible maintaining information about a user.

Simulation class

The **Simulation** class is responsible for simulating individual thermostats. It takes in a control variable, adds it to the last simulated temperature, and returns the new temperature to the PID class.

Vulnerability code

Step 1: Bypass of subnormal number guard

Pseudocode:

```
error = thermostat current temp – setpoint
if error is subnormal:
reject input
// Here if the error is within ε of the subnormal range,
// changing the metric will push the error in the subnormal range.
if a metric change was included in the request:
change the metric
update temperatures
```

Step 2: Amplification of subnormal performance degradation

Pseudocode:

```
count = 0
while no new input has been received by the thermostat && count <= 32,000,000:
count = count + 1
run PID algorithm on temperature error
simulate temperature change by adding PID output to current temp
      error = thermostat current temp – setpoint
      // guard guarantee that no benign input triggers vulnerability
```

```
      if initial error was normal and new error is subnormal:
      error = 0
if count > 32,000,000:
send prediction to thermostat along with adjustment instructions
```

Here, the vulnerability is amplified to an observable time difference. Operations on subnormal numbers can be slower than normal operations by a factor of ~10. Here, calculations are run 32,000,000 times to amplify the running time such that the server takes longer to respond to the thermostat. In the case where a malicious user has inserted a subnormal number into this loop, the report rate will slow considerably.

What is the sequence of adversary actions that will trigger the vulnerability?

1. Input a temperature and setpoint whose error is within ε of the subnormal range.
2. Include a change metric request to the server.

4.1.2.2.2 Challenge Design

4.1.2.2.3 Use Cases

Thermomaster is a temperature prediction and control system. All requests must be preceded by a username and password. First, the user may input a current temperature and setpoint to the server. In this case, the server will return the next 10 adjustments the thermostat should make, and second, a forecasted temperature if the user makes the said adjustments and continues to send updates to the server. In addition, the user may specify a number of modifications to the PID prediction algorithm that change its operation. They are as follows:

- Enable/disable integral term
- Enable/disable derivative term
- Enable/disable integral windup
- Set term constants
- Set integral windup
- Change the metric from Fahrenheit to Celsius
- Change the metric from Celsius to Fahrenheit

Lastly, the user may request the server to display the current settings or reset the PID controller to a default state.

4.1.2.2.4 Overall code structure

Thermomaster consists of the following classes and methods (Figure 5):

Figure 5. Thermomaster Class Diagram

4.1.2.2.5 Inputs and Outputs

All inputs to the server must be preceded by a username and password first, and an identifying PID name second. If the PID name is not found, the server creates a new PID controller with said

name. After which, any number of commands may be added in any order, which will be processed after first passing validation. “set setpoint”, “set current”, “change metric”, and “set windup” must all be followed by an argument on the next line. The input format is shown below. Example inputs and responses are also provided:

Template:
Username
Password
PID name
Operation 1
Argument to operation 1
Operation 2
Argument to operation 2
…
Example:
Bob
Bobspassword
PID1
set setpoint
1.0
set current
-5.0
change metric
C
display settings
disable integral

4.1.3 WordShare

4.1.3.1 Description

WordShare is a cloud-based word processing application that allows users to securely store documents. In support of this, WordShare includes two important features: 1) it can encrypt words in a document's text inline, and 2) it provides a search feature which allows users to find strings in documents made by all users. The first feature is provided via a document markup format that includes tags for labelling text as secure, combined with support for encryption of these document parts to ensure only trusted users can read them. The second feature is provided via an algorithm that stores documents in a trie-based indexing structure to make them searchable.

WordShare contains both a time side channel vulnerability and space complexity vulnerability.

For the time side channel, the secret is the encrypted text within a WordShare document.

The WordShare application is a secure document storage and search server. It provides security by encrypting sensitive words in a document using a secure encryption algorithm with a unique initialization vector[2] for each encrypted section. This capability seemingly protects the sensitive content of a document from disclosure, but it does not protect its integrity. Integrity protection requires the use of a cryptographically secure Hashed Message Authentication Code (HMAC) [3],

which WordShare provides but only for the entire document's plaintext rather than for each encrypted section in isolation. As such, users who are authorized to edit a document can alter the cyphertext of any encrypted sections contained therein. In and of itself, this does not leak the plaintext of those encrypted sections. However, when this oversight is paired with two other capabilities accessible to an attacker, the plaintext may be inferred.

These other two capabilities leveraged by the attacker are: 1) the ability to modify the encoding scheme of a document (i.e., like base64), and 2) the ability to observe differentials in the timings of operations based on whether or not a document with the same plaintext content already exists on the server in its trie-based index.

Timing differentials occur when the search algorithm is asked to delete a document from the trie-based indexing structure, which indexes the plaintext of all documents including their secure sections. In a trie index, deletion of a document involves removing all unique trie nodes starting backwards from the end of the structure until a node is found that is shared with another document. When a document in the trie is duplicated exactly, no nodes in the trie need to be deleted and only the reference from the last node in the trie to the duplicated document has to be removed. As a result, deletion of a duplicated document in a trie index is guaranteed to take one operation, whereas removal of a unique document can take at worst as many steps as the document has indexed characters. With this in mind, the attacker's goal is to create a duplicate of a secure document in the trie where the attacker knows the contents of the duplicate's plaintext and can then determine if that plaintext matches the secure document's plaintext based on observations of deletion timings.

To accomplish this, the attacker relies on his ability to alter a document's encodings. A WordShare encoding allows users to map byte values that are unused by WordShare to a pair, (*w, g*). Here, *w* is a word (i.e. a string of supported characters that are not broken by a space or other delimiting character) and *g* is the unused byte that represents the word *w*. WordShare will only use a portion of the possible characters a byte can represent, say the first 26 byte-values (i.e., 0-25) to the 26 lowercase characters of the English alphabet, these are the supported characters. All other possible byte values (i.e., 26-255) will map to nothing and be considered unused. However, user-supplied encodings can include mappings from any unused byte-value to any word. This intention of this feature is to allow users to compress common words by mapping them to a single byte, but it also enables the attacker to map the remaining 230 possible byte-values to some particular used word, say the single character word ‘z’ or '@'. Now, when the attacker changes the document encoding to his own and then changes the value of a cyphertext byte, there is a ~90% (= 230/256) chance that the corresponding decrypted plaintext becomes a ‘@’. This makes it easy for the attacker to make changes to the cyphertext of a word that will cause the corresponding plaintext to very likely consist of just their specially chosen character.

Using this flaw, the attacker can modify all but one character in the cyphertext of a word, then create an identical document that contains all ‘@’s, for example, except at that location and use timing observations in the document delete operation to identify the remaining unmodified character. To do so, the attacker would enter unencrypted plaintext words into the document that resemble the modified plaintext of the encrypted word but which differs in the desired location. The attacker then varies that character and observes the *delete document* operation’s timing to see

if the current version of the word has been seen previously. If it has, then the attacker knows he has found the correct character. To discover the remaining unknown characters, the attacker repeats this process for each unknown encrypted character.

The space complexity vulnerability is caused by hash collisions over the WordShare document name field. When a document is added to WordShare, it can also have alias names set. When these alias names collide with each other, or with the original document name, the result is a call the WordShareDoc object's equals() function. Equals, in this case, has been overridden to check whether or not the contents of the on-disk file related to the two names are also equal. To do this it loads the on-disk file into a byte array for each object and then rewrites these files to disk after performing the equality check. However, when it does this, it rewrites them to the alias name, not the actual document name. Prior to equals being called, only one copy of the file exists under the actual document name. After equals is called, a copy exists for each unique hash collision.

It should be noted that the equals() method is never called directly by WordShare, it is called only by Java's HashMap function only if a collision occurs.

To make the attack work, the attacker just has to upload one large file (approximately 135 KiloBytes (KB) in size) and give that file a name with many alias that all collide. In total 294 collisions are provided in the proof, taking up about 35MB in size.

The large file and the alias string are both sent in via the POST data and, together, can equal no more than 200KB in size.

4.1.3.2 Software Design

Related Module and Class Roadmap

The important classes in the design of WordShare include:

- The WordShareServer: HTTP server accepts new docs and requests to find or delete docs.
- WordShareDocControl: Parses requests.
- The WordShareAuth: Handles keys for users, performs encryption and signature checks on document contents.
- The WordShareDoc Class: Holds information about documents, their creator, contents, etc…
- The MapTrie: The trie index, holds the relationships between nodes (i.e. TrieNodes) in the index.
- The TrieNode: An entry in the trie. Essentially an object wrapper around a character (byte) in the trie that points to another character or a terminal document.

The *Concept Description* section above provides an overview of how the WordShare vulnerability works. This section provides additional low-level details about the WordShare followed by a step-by-step walkthrough of the vulnerability.

Some low-level details about the WordShare implementation to consider are:

- The WordShare server indexes in unencrypted form. It, however, stores and transmits documents with secret sections encrypted.
- Only individual words in a WordShare document are encrypted. Each encrypted section is 16 bytes in length (ignoring initialization vector) and can hold a word up to 16 bytes in length.
- Documents are checked for digital signatures when added by a user. Documents with invalid signatures are stored but are not indexed. Whether or not the document's signature was validated or not is transparent except through observation of the deletion-based timing side channel.
- Documents are sent to and from the WordShare server in hypertext markup language (HTML) format. There are certain limitations on document sizes and on parts of the document, such as:
    - Max total document size is 200 MegaBytes (MB): This is the entire HTML document that is stored on disk by the server. This size is limited by the hypertext transfer protocol (HTTP) form post maximum size default value in Jetty.
    - Max total text content of an HTML file (does not include tags): 10000 bytes. This includes all text in 'p' tags and 'secure' div tags.
    - Max size of a plaintext word in a 'secret' div tag: 16 bytes
    - The trie only expects words of 16 bytes in length to be indexed. If a word is entered that is larger than 7KB in bytes (by taking advantage of bugs in WordShare), the trie will throw a StackOverflow Exception if it is asked to delete that document.
    - Max size of an encoding div tags text content: 1500 bytes
    - Max size of a document name field in HTTP request: 200 bytes
    - WordShare accepts the following characters as part of words in the plaintext: any UTF-8 (Unicode Transformation Format) character between hex value 0x30 and 0x5a (inclusive). This includes number, upper case letters, and several miscellaneous characters like the '@' symbol. -- These are considered to be the WordShare supported characters.
    - WordShare recognizes the space ' ' and '.' characters as delimiters between words.
    - Note: text content refers to the strings between HTML tags. Example: This is the text .

With consideration of the information above, the following is an ordered list of the exact series of steps required to execute the WordShare time side-channel vulnerability:

1. **Attacker downloads a document with a secret word:** This results in the following relevant actions occurring on the server:

a. Server sends the stored (on disk) version of the document. This includes secret words encrypted using a privileged user's key. The attacker does not have access to this key.

2. **Attacker crafts and submits a document with known characters in place of secret.** The attacker performs the following actions to create this document:
    a. Attacker makes a new document that is over 7k bytes in size in terms of plaintext alone where all the plaintext characters (i.e. bytes) form one big word in the index. Note: Because the server limits individual word lengths to 16 bytes, this is accomplished by taking advantage of a flaw in the server guard that checks word lengths.
    b. The attacker creates a 16 byte string with all bytes consisting of '@' symbols. Attacker adds this string to the document.
    c. Attacker signs the plaintext using their key.
    d. Attacker submits their specially crafted document to server.
3. **The server receives the attacker document and stores it on disk and in the trie index.**
4. **Attacker creates a new version of the document they just submitted, but replaces the 16 byte '@' string with the secret div from the victim document.**
5. **Attacker attempts to alter the secret document's IV to cause it to have identical contents to their crafted document and submits their own new document with the modified IV, replacing the '@' string above with the cyphertext from the original secret document.** This involves the following steps:
    a. Modifying the secret document's initialization vector and replacing it with a random initialization vector (IV) [4].
    b. Submitting the modified doc with the signature obtained from signing their version of the plaintext from the previously submitted, non-secure document.
6. **The server receives attacker modified secret document and performs the following actions:**
    a. Decrypts the sensitive words and recreates the complete plaintext by combing the decrypted plaintext with rest of the document's plaintext.
    b. Enters the document's complete contents into the trie index, character-by-character, and makes a reference to the document for searching.
7. **Attacker deletes the submitted secret document from the server and observes timing to discover if document already exists:**
    a. If the deletion timing is fast, it already exists indicating to the attacker that they have found an IV that creates an all '@ version of the secret.

8. **Attacker repeats above steps by creating random IV's until they find an all '@' version of the document's secrets.** Once they find the all '@' version, they use the IV that creates it as a baseline and repeat the attack for each individual letter by modifying only one byte in the baseline IV for each character up to the max word size. (Each character is equal to one byte).

4.1.4 YapMaster

4.1.4.1 Description

YapMaster is a secure message broadcasting system. It works similar to a Twitter clone except user messages, called *Yaps*, are private by default. Users have the ability to either keep their *Yaps* private or share them with a select group of users. To ensure privacy, *Yaps* are encrypted at all critical points with a key that is unique to the individual *Yap*. The goal is to enable a fine-grained level of secrecy.

The YapMaster challenge contains a **Space side channel and Space complexity vulnerabilities**. With regard to the space side channel, the secret is the encrypted full contents of any private Yap message.

The vulnerability is related to a buffer write overflow caused by improperly checked length fields. The buffers involved are the global Yap message buffers, which hold the encrypted and decrypted Yap messages of multiple users, and the Rivest Cipher 4 (RC4) streaming encryption key buffers, which are unique to each user. Normally, there exists a one-to-one relationship between the corresponding byte contents of all these buffers. However, due to a glitch in the length check guard of the logic that decrypts a Yap message, an attacker may cause one of their own Yap messages in the global buffer to exceed its expected buffer area. When this occurs, the Yap message processing logic will think the attacker's Yap message is longer than it really is and, as a result, keep looping even after it exceeds the message's intended buffer area. This additional looping activity causes the related decryption routine to overflow and eventually be used to decrypt the neighboring Yap message's encrypted contents, but by using extra bytes in the attacker's key.

These decrypted bytes then alter the victim's plaintext. As a result, they create a situation where the attacker can craft a special version of their own decryption key that, when combined with the cyphertext in the adjacent victim user's Yap buffer, causes attacker chosen bytes to be written into the beginning of the victim's Yap message's plaintext buffer, which is calculated just prior to transmission. If crafted correctly, these bytes form a prefix in the victim's message that causes the next byte after the prefix to be treated as the length field for the victim's private Yap message. Subsequently, this causes the Yap message processor to create a packet for transmission that is the size of the byte value it thinks is the length field. However, since that length field was actually a byte value of the victim's Yap message, the length of the packet will correspond to the unencrypted value of that byte and give away the secret value of the character which that byte represents. This, in turn, results in a space observable that reveals the victim's secret Yap message one character at a time as the length of the prefix is increased.

YapMaster Space Attack

YapMaster contains a simple space based vulnerability. The vulnerability works by allowing the user to submit a GET request for a Yap that does not yet exist and has a very large ID value. When this occurs, the Yap Server bypasses the cache and attempts to get the Yap message from the Random Access Memory Mapped disk store. The Memory Mapped disk store reacts to this request by multiplying the index of the requested non-existent Yap by the normal Yap message size and expanding the size of the memory mapped buffer by this size. Even though the operation that performs this step is flagged READ_ONLY, the increased buffer size is reflected on disk. So, if the attacker asks for the Yap with ID 900,000 the resulting output file will be 115200000 bytes or 115MB (900k * 128). It should be noted that Yap messages cannot actually be larger than ID 19999. This max number, however, is NOT checked when attempting to read a Yap, only when creating them.

4.1.4.2 Software Design

Related Module and Class Roadmap

YapMaster has three primary modules: The Service Control Module, the Authentication Module, and the YapMessage Store Module(see Figure 6).

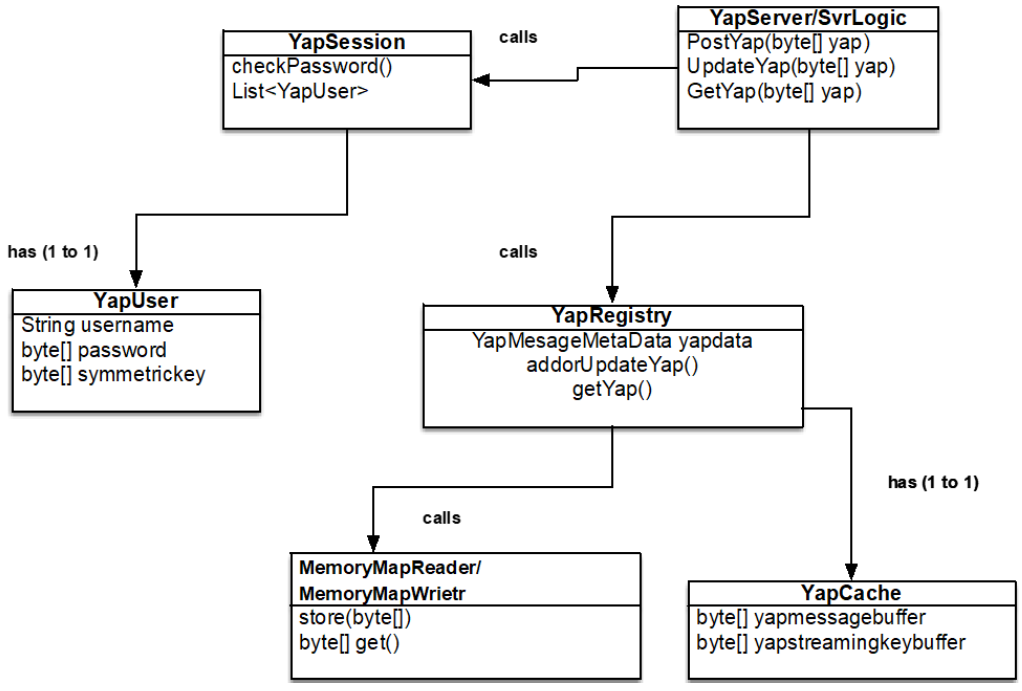


Figure 6. Yapmaster Vulnerablity Class Diagram

The Service Control Module (SCM)

The classes in the SCM are responsible for starting the service, monitoring it, checking inputs, and forwarding user requests to their proper location. It includes the following classes:

- YapWebService: Implements the Representational State Transfer (REST) interface http server, starts the server, handles exceptions from other modules or those thrown by the embedded http server. Also verifies messages have not been modified using HMAC and passwords are correct -- although it only does each of these tasks for certain operations (as is described in comments in code).
- SvrLogic: The SvrLogic class is the main input handler. It parses requests and coordinates how they are stored or retrieved from the YapMaster server. This includes whether or not they are retrieved from cache or disk and what their unique ID number should be.
- SvrLogicWrapper: Small class, handles basic routines for associating requests with particular users via the retrieval and creation of YapSessions.

The Authentication Module (AuthMod)

The AuthMod are utility classes that are responsible for tracking user ids, managing keys, checking passwords, and decrypting requests. It is found in the packages yap.auth, yap.user, yap.user.auth, and yap.util.auth. includes the following classes:

- YapAuth: Performs password authentication.
- YapUser: A class that holds information related to a user, like the user's name and encryption keys (i.e. the seed that is used to create the key).
- YapMasterUser: These are the users that can create other users (although they cannot actually Yap). They can be thought of more as domains for users than actual users. New YapMasterUser's cannot be created after the server has started. Their credentials are stored on disk and loaded at server start time. There is no real logic in this class, it inherits off of YapUser. In order to perform the actions of a master user, however, a YapUser class instance must also be of this type; this is checked by calls to instanceof.
- YapSession: Holds state information about YapUsers at runtime. Also holds a reference to a user's cache. The manner in which it holds this reference makes the cache overwrite necessary for the space side channel possible.

YapMessage Store Module (YapStore)

- The YapStore is responsible for storing all Yap Messages and related data either on disk or in memory in a cache. It includes the following classes:
- YapRegistry: Stores information about all Yaps and their locations in storage or in a cache. The registry manages a collection that relates the user ID to their cache. The collection allows collisions via poorly coded logic in the equals() method which enables the main space side channel vulnerability by allowing the attacker to add messages to another user's cache.
- YapCache: Holds a cache of Yap Messages in memory. The cache contains the byte array(s) that are involved in the buffer overflow, these arrays hold the bytes associated with a Yap's contents and its streaming key.

- MemoryMapReader/ MemoryMapWriter: Retrieves/Stores Yap Messages on disk. The Reader has code which allows the space complexity attack.
- YapStats: A class that stores tracked user activity, like number of Yaps. The tracking of the number of Yaps is used in the vulnerability to discover at what position in the cache a passively observed Yap POST is placed, as is described in Vulnerability Code below.
- YapLabels: Stores the label portions of a Yap message and associates them with the message they came from by ID value. See the Yap Message format below for clarification.

The format of a YapMessage: Table 1 shows the format of a Yap message in a HTTP Yap request. All messages are intended to be 128 bytes regardless of the length of actual contents, although this is not enforced to make the oracle described in the Vulnerability Code section below possible. This only describes the Yap message contents, additional fields in the HTTP request, like the Yap username, are discussed in the Inputs and Outputs section. The Label is the prefix of a Yap message; it acts like a publically searchable hashtag even though it is encrypted when initially sent to the server.

Table 1. Format of a Yap Message

1 byte: The yap Label length (x)	The Label: x bytes	1 byte: The yap message length (y)	The yap Message: y bytes

4.1.4.2.1 Vulnerability Code

The Concept Description section above provides an overview of how the YapMaster vulnerability works as the result of a buffer overflow that enables the modification of memory-resident, secret plaintext by attacker supplied data. What that section does not describe, though, are the mechanics of how exactly an attacker manages to position their own data into the vulnerable buffers so that it is directly adjacent to the victims' data and thus can overwrite critical bytes to alter packet sizes. Those details are provided in this section.

Before describing the exact sequence of steps that allow this to happen, there a few low-level details about how the YapServer handles buffers that should be clarified first. These low-level details are:

- The YapServer stores Yaps, and their related decryption keys, in two locations: on-disk (YapDiskStore class) and in a cache (YapCache class). A separate instance of each of these locations is associated with each unique YapUser in the YapMessageRegistry class.
- Each YapUser's cache consists of byte buffers that store Yap messages and their corresponding RC4, streaming encryption keys. Both message and key storing byte buffers are the same size and relate to each other in a byte-by-byte sense.
- A YapUser's cache stores the latest x number of new messages POSTed by its corresponding YapUser, only. The assumption is that recent Yaps are always the most frequently accessed, so only recent Yaps are cached.

- Cache buffers are statically sized and circular, meaning that new Yaps are appended into the buffer at the next location until the max buffer size is reached. Once the max location is reached, the write location resets to the beginning of the buffer and begins overwriting older messages.
- All Yap messages and their keys take up the same amount of space (the max Yap size) in the cache's buffers regardless of their actual size. In other words, if the buffer size is 1280 bytes and the max Yap message size is 128 bytes, then the buffer will hold ten Yaps. Yap messages smaller than 128 will be padded.
- For lookup purposes, the YapServer maintains a lookup array that maps Yap usernames to caches. The array equality check is based on username. However, the check does not account for means by which collisions are possible and are not handled correctly (more details in proof comments). As is described below, creating a collision between usernames is an essential step in performing the attack as this allows the attacker to write into the victim's cache buffer using their colliding username.
- The attack destroys the victim's message in the cache, it does not alter their on-disk representation.

In other words, the gist of all the above bullets is that there exists a pair of in-memory buffers that cache the most recent Yaps and their keys. These buffers store messages adjacent to each other in order of when they are posted and, although they are intended to be accessible only by the owning user, they are susceptible to be written into by another user as a result of a collision bug. Once this happens, the attacker can perform a series of yet to be described steps that allow them to overflow intended areas of the buffer and, as a result, allow them to observe a secret's value byte-by-byte.

With consideration of the information above, the following is an ordered list of the exact series of steps required to execute the YapMaster space side-channel vulnerability:

1. Attacker observes a Victim posting a new, secret (encrypted) Yap message and extracts the cyphertext of the victim's Yap (1 passive operation). The attacker must know the cyphertext of the first byte of the Yap they are attempting to discover; if they do not, they will not be able to determine which message byte they are modifying. The attacker learns the cyphertext by observing a victim's Yap ADD request on the network.
2. Attacker checks YapServer to determine the number of the victim Yap messages currently on the server (1 active operation): The YapServer has an interface that provides public information about all users via the YapStats class. The attacker requires this information for overwriting another user's buffer (reason described below).
3. Attacker gets the Label associated with the last victim ADD call: In order to know the length of the last Yap message added, the attacker calls the LABEL command to get the labels associated with that user. Knowledge of the label text allows the attacker to know the maximum size of the last submitted secret Yap.
4. Attacker creates account with a username that has a collision with victim's username to enable overwriting into victim's YapCache (1 active operation): takes advantage of string matching bug described in proof comments.

5. Attacker POSTs Yap messages to their new account equal to the number of victim Yaps in the cache (y active operations): The attacker must create enough Yap messages so that the index of their message corresponds to the index of the most recent victim Yap in the cache. The cache holds 10 messages; if the attacker wants to steal the 10th message in the victim cache, they must add 9 of their own messages.
6. Attacker Crafts Yap message packet with incorrect field on Yap message and encryption key: The message contains an incorrect value that, when parsed by the server, causes the encryption key's length to be one greater than the actual buffer. This extra byte in the encryption key must be chosen so that, when it combines with the cyphertext of the victim in the adjacent buffer, it results in a decrypted value that is a legitimate Yap message prefix (label) length field.
7. Attacker Submits Crafted Message (1 active operation).
8. Server parses attacker's Yap and, as a result of naming collision, places it in the same cache as the victim's Yaps: The server updates the existing victim Yap at that location in the cache with the one submitted by the attacker. It then decrypts the attacker's Yap using the attacker's key. Then, using the one extra byte in attacker's key, it decrypts just the first byte of victim's Yap. This first byte determines the length of the Yap prefix. The value immediately after the prefix's termination is the Yap message length field, which the server now confuses for a regular byte in the victim's Yap as a result of being fooled into thinking the Yap prefix is a different length than it actually is.
9. Attacker Requests content of their own Yap message that corresponds to the victim Yap (1 active operation).
10. Server checks Yap message at location of victim Yap: Server sees the existing victim Yap with first byte decrypted by the attacker's key, then it finds byte it now thinks is the Yap message size field and copies that many bytes from the victim's decrypted Yap buffer and encrypts them using the non-affected portion of the victim's key and sends an improperly sized, but encrypted, Yap back to the attacker.
11. Attacker a Receives a padded, encrypted Message with its size field and message length set to the value of a secret Yap Message Byte: The message contains the information but is encrypted. Attacker must perform next step to discover size.
12. Attacker modifies message size by truncating bytes from end of message and sending truncated packet to an oracle on the YapServer: The oracle sends back different sized error messages for each of the three possible situations: message is too small, message is exact size of length field, or message is too big. This enables attacker to perform a binary search.
13. Attacker repeats this process (steps 6 through 12): For each repetition, the attacker updates the attack Yap by modifying the value of the overflowing byte value of their encryption key to modify the first byte of the victim Yap as needed to learn a new byte in the secret text. Note: guards in the Yap system will only let the attacker modify the first byte of the victim's key in the cache.

4.1.5 Emu6502

4.1.5.1 Description

The program is a web application that provides an "emulator" (actually just an interpreter) for the 8-bit 6502 microprocessor. A user may select from a number of provided 6502 assembly language sources and execute them in the browser, viewing the register and memory values like a debugger.

Emu6502 contains a **timing side channel** that leaks what source code a user is currently executing.

As a piece of source code is interpreted, the challenge application replies with updated register/memory values that are Secure Socket Layer (SSL)-encrypted to prevent trivial observation. However, when a jump/branch instruction is taken, the interpreter throws an exception (a rather slow operation in Java) resulting in a small but observable delay in its reply. This means an adversary with network access may observe the branches a program takes by monitoring the network for delays. They may then cross-reference the observed delays with the corpus of source codes to discover which program a benign user is executing based on the location of delays (branches taken).

4.1.5.2 Software Design

4.1.5.2.1 Related Class Roadmap

Figure 7 depicts the Emu6502 classes related to the vulnerability.

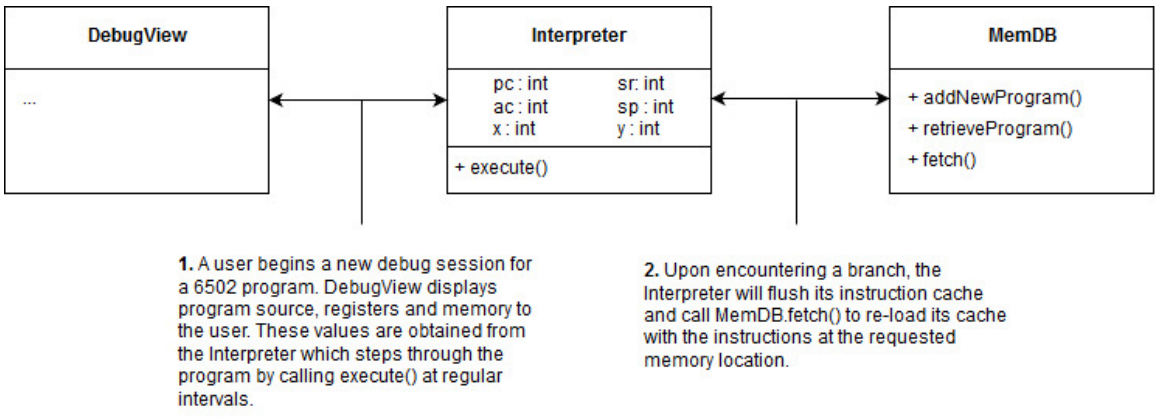


Figure 7. Emu6502 Classes Related to the Vulnerability

- **MemDb:** Service that acts as a wrapper around a SQLite database. The database itself stores emulated programs including source code, debugging information and compiled (binary) code. In terms of the vulnerability, it acts as a crude "memory management unit" for the emulator, allowing for the emulator to fetch instructions at arbitrary memory locations. This introduces a time penalty since the method fetch() executes a Structured Query Language (SQL) query, roughly simulating the delay due to the interpreter re-loading its cache.

- **Interpreter:** Represents a 6502 processor with corresponding registers and memory. When a user begins executing a submitted program, the Interpreter receives compiled (binary) code which it executes one instruction at a time. Instructions themselves are stored in a HashMap of opcodes (Integers) to anonymous lambda functions. To execute the compiled code of the program, the interpreter performs a get() using the current instruction as a key and executes the resulting function. All Y instructions of the 6502 are implemented here, with a subset of X of them triggering the vulnerability. Any branch or jump instruction calls MemDb.fetch() to get the instructions at the destination location whereas other instructions simply increment the Program Counter (PC).
- **DebugView:** User interface component that displays the executing program to the client, showing the values of the registers at each step. This is implemented using Vaadin which renders it as a web page such that every step forces the register values to be updated so an adversary may monitor the network for traffic.

4.1.5.2.2 Vulnerability code

The "Interpreter" contains a cache of instructions loaded from the submitted program. Each instruction is executed by calls to execute() which loads the instruction at the PC and attempts to interpret it, updating registers and memory as necessary. Instructions themselves are machine codes (e.g. 0xEA for NOP, 0x4C for JMP to an absolute location) which are stored in a HashTable of Integers to anonymous lambda expressions. As an example, two instructions, NOP and JMP, are implemented above. A JMP will trigger the vulnerability by re-loading the instruction cache from MemDB while a NOP simply increments the PC to the next instruction as normal.

MemDB handles saving and loading programs from a SQLite database stored on disk. It also implements fetch() which causes a noticeable delay when called by branching instructions. The vulnerability itself is triggered when the Interpreter executes an instruction that calls fetch (any branch) and an attacker can discern that a branch has been taken. In instances where branches are not taken, there will be no delay as the instruction cache will not be re-loaded by fetch() and the PC will be incremented normally.

4.1.6 PowerState

4.1.6.1 Description

PowerState is a power system state estimator (PSSE) for electrical power systems. In essence, a PSSE resolves the unknown variables in a system of differential equations with unknown error in the known values to determine values for the voltage, current and power in each section of the circuit. PowerState determines the most likely current state given an electrical network specification and a set of observed values.

PowerState contains a **time complexity vulnerability**.

Solving a system of partial differential equations using methods such as Newton-Raphson relies upon the ability of the solver to approach the best solution through stepwise approximation. With sufficiently incongruous initial values, the solver will either approach the solution very slowly or will never converge on plausible outputs. It will be challenging for the blue team to attempt to classify inputs into those which will successfully be solved and ones which will not.

4.1.6.2 Software Design

4.1.6.2.1 Vulnerability code

The vulnerable code is located in PowerStateSolver.java file, PowerStateSolver.solve() method.

Where is a while (tol > 1) loop which loops to achieve low error tolerance. The value of tolerance is assigned by applying Newton's method to partial differential equations for power flow so the next value of tol is impossible to predict without actually running the code. System does some sanity checking on the input ahead of time – makes sure the power grid is fully connected and that the power generation capacity is larger than the system load. However, power line data is modeled with impedance only (resistance for alternative current, which accounts for phase changes). Therefore power dissipated over power lines is unspecified in the input file and sanity check does not account for malicious values in power line resistance. It is possible to set power line resistance to be too high, which in turn increases the load on the system past the generation capacity and the solver will not converge.

An important node: there are infinitely many stable configurations, but also infinitely many unstable configurations.

4.1.6.2.2 Use Cases

1. User uploads MATPOWER file
2. User gets estimated solution or error

4.1.6.2.3 Overall code structure

WebServer.java is the entry point. It calls RequestHandler for each network request, which in turn gets the input file, passes it to PowerStateMatpowerParser, which uses an OpfMatpowerAdapter from IEEE library to actually parse the file. Then PowerStateMatpowerParser converts the Optimal Power Flow's (OPF) eXtensible Markup Language (XML) structure to the type describe by State.java and finally PowerStateSolver operates on State.

4.1.6.2.4 Inputs and Outputs

Input is IEEE MATPOWER file type. Output is a list of voltages and phases or errors about misformed input.

4.1.7 PSA

4.1.7.1 Description

Python Static Analyzer (PSA) performs static program analysis on python bytecode. It provides an intermediate representation (IR) for python bytecode, as well as a modular framework for executing reaching-definition-based analyses (RDA). It comes pre-packaged with the following modules:

- Partial Static Single Assignment (SSA) Conversion
- Calling Context Tracing
- Object data tracing/context sensitivity.

- Basic Static Value Analysis

PSA provides a web front-end that allows users to retrieve results from previous analyses, and either download them or perform additional processing. It provides the following operations:

- Create a project.
- Add a python module to an existing project.
- Execute one or more analyses on a given project under specific configurations.
- Download analysis IR from a project.

To avoid resource starvation, analysis requests are queued internally.

PSA has both time and space complexity attacks.

4.1.7.1.1 Time Complexity Attack

PSA's time complexity attack is a general weakness of contextualized RDA algorithms. RDA works by converting the instructions in the program into an implicit or explicit graph format. The problem is that this is only half of the picture; one instruction can have multiple possible impacts depending on the state of the program when it's executed. This state can't be observed directly from the static program data. To combat this, RDA algorithms use Abstract Interpretation; they simulate enough of the state of the program for their purposes, then compute each instruction's impact on that state.

The key is that the analyzer needs to process each instruction under each state separately. A single instruction executed in 100 states takes as much time to process as 100 instructions in one state. Additionally, an instruction that forks multiple states – such as a function call or indirect data reference – multiplies the contexts that all future instructions need to deal with. As these multiplications of complexity stack with each other, they can cause a combinatorial explosion of contexts the analyzer needs to deal with.

The annoying thing is that it's extremely difficult to predict how complex a given program will get before analysis. Size of the program has some impact, in that it allows more room for complex instructions, but even small programs can have extremely complicated abstract state.

The key to PSA's vulnerability is in its calling context tracing module. This module allows the user to configure the maximum depth of the virtual call stack that they want to maintain. Setting this to a high number and then feeding in a heavily recursive program will cause the analyzer to experience an explosion in execution contexts.

To exploit this vulnerability:

1. Adversary creates a new project.
2. Adversary uploads a custom module "madrecursive".
3. Adversary requests call graph analysis with a virtual stack depth greater than 20.

4. The analysis will take an extremely long time to terminate, blocking all future queries as the queue will never unlock.

The virtual stack depth is an optional parameter; it defaults to a value of 3, which will rapidly terminate for program inputs within the budget.

4.1.7.1.2 Space Complexity Attack

The space complexity attack occurs because PSA stores its analysis results locally. There is an optional serializer module that dumps all analysis state to disk to allow checkpointing. This module has an optional "verbose-on-error" mode that serializes every revision of every module if the first pass of PSA produced an error. A program that produces a large number of results early, and requires a large number of revisions before forcing the analyzer into an error state, will consume a correspondingly-large amount of disk space.

4.1.7.2 Software Design

4.1.7.2.1 Time Complexity Attack

The time vulnerability gets triggered if you try to run callgraph analysis on python code similar to the following:

```
#In module m:
def f(x):
f(x)
f(x)
```

The python bytecode will get converted into a single basic block with the following IR:

```
0: Local[m.f.stack_push@0] = Global[f]
3: Local[m.f.stack_push@3] =
      Argument[function = f.stack_push@0, name = "return"]
6: Local[m.f.stack_push@6] =
      Argument[function = f.stack_push@0, name = "return"]
```

Callgraph analysis (CGA) propagates the root context to f():

```
Contexts for m.f:
!ROOT!
```

Points-to Analysis (PTA) resolves Global[f] as a reference to the function object m.f(). It propagates that function object to stack_push@0:

```
Allocs for Local[m.f.stack_push@0] under context !ROOT!:
Function[m.f]
```

CGA now has both a context and a function object for the two argument references. It resolves them as references to m.f(), and builds new contexts for m.f().

```
Contexts for m.f:
```

```
!ROOT!
!ROOT!|m.f@3 => m.f
!ROOT!|m.f@6 => m.f
```

PTA now has to handle stack_push@0 under three contexts; the root contexts, and the two recursive contexts. It propagates the function object for all of them:

```
Allocs for Local[m.f.stack_push@0] under context !ROOT!:
Function[m.f]
Allocs for Local[m.f.stack_push@0] under context !ROOT!|m.f@3 => m.f:
Function[m.f]
Allocs for Local[m.f.stack_push@0] under context !ROOT!|m.f@6 => m.f:
Function[m.f]
```

This, in turn, allows CGA to generate two new contexts for each of the recursive contexts:

```
Contexts for m.f:
!ROOT!
!ROOT!|m.f@3 => m.f
!ROOT!|m.f@3 => m.f|m.f@3 => m.f
!ROOT!|m.f@3 => m.f|m.f@6 => m.f
!ROOT!|m.f@6 => m.f
!ROOT!|m.f@6 => m.f|m.f@3 => m.f
!ROOT!|m.f@6 => m.f|m.f@6 => m.f
```

This cycle will continue until the number of function calls encoded in the context exceeds cg.max_call_depth, which is optionally configurable by the adversary. At that point, CGA will start truncating elements off the left-hand sides of contexts until it has discovered all possible sequences of recursive calls of length cg.max_call_depth. This is trivial for the default call depth of 3, but once it passes 18, it starts taking minutes into hours to complete one feedback cycle between PTA and CGA.

Meanwhile, the front-end REST application programming interface (API) will put a hold on all incoming analysis requests until this job completes, locking up the analyzer for all other users.

4.1.7.2.2 Space Complexity Attack

The time vulnerability gets triggered if you try to run callgraph analysis on python code similar to the following:

```
#In module m:
def f(x):
x = x + x
f(x)

def main():
f(“This is a string.”)
```

Just from a callgraph/PTA perspective, this is a simple program; the analyzer will explore f’s infinite recursive calls until it runs out of call depth. The problem is with the value analyzer. It will detect the string constant passed into f’s parameter x, and assign it to the appropriate variables:

```
Values for Local[m.main.stack_push@3] under context !ROOT! => m.main:
String[“This is a string.”]
```

```
Values for Local[m.f.x] under context !ROOT! => m.main|m.main@9 => m.f
String[“This is a string.”]
```

It then propagates that value through the “+” operator, evaluating the result. In practice, value analyzers rarely work this way for this exact reason; storing every single possible result of every operation given every input in every context is intractable. PSA is not so smart:

```
Values for Local[m.f.stack_push@0] under context !ROOT! => m.main|m.main@9 =>
m.f
String[“This is a string.”]
Values for Local[m.f.stack_push@3] under context !ROOT! => m.main|m.main@9 =>
m.f
String[“This is a string.”]
Values for Local[m.f.stack_push@9] under context !ROOT! => m.main|m.main@9 =>
m.f
String[“This is a string. This is a string.”]
```

This new string then gets passed to the x parameter of the recursive call to f(), and gets doubled again. In theory, this is bounded by the calling context depth. However, Java arrays – which store the characters in a string – can have a maximum size of 2^{31}-1 because their indexes are 32-bit signed integers. With a sufficiently-large virtual stack depth, the size of the strings generated will keep doubling until one breaches that limit and causes an out-of-memory error.

Under normal configuration, PSA will terminate cleanly. However, the driver supports an optional retry_on_error argument. If this is set, the analyzer will retry a failed analysis once. Meanwhile, the serializer module has an optional verbose_on_error argument that only activates during a retry.

Normally, the serializer waits for all analysis modules to finish, and then writes their internal state out to the file system to be loaded later. In verbose mode, the serializer writes out every revision produced by every analysis module. In the case of our infinitely-doubling strings, this will produce some impressively-large data sets before failing. A clever adversary could use a larger version of the example presented here to produce gigabytes of output before the analysis fails its retry.

4.1.7.2.3 Vulnerable Code

4.1.7.2.4 Time Complexity Attack

The following code includes the critical segments of the PTA and CGA modules. Please see sequence_diagram.pdf for a complete overview of the vulnerable workflow.

4.1.7.2.5 Points-To Analysis

Resolve default allocs for global variables for a given function. This is how Python references modules and functions within a given function’s namespace.

```
private Set<Alloc> injectScopedGlobals(PyFunction function, GlobalRef ref) {
Set<Alloc> out = new HashSet<>();
if(function != null) {
      if(function.getGlobals().containsKey(ref.getName())) {
            PyValue val = function.getGlobals().get(ref.getName());
            switch(val.getValType()) {
                  case FUNCTION:
                        PyFunction func = val.getAsFunction();
                        out.add(new FunctionAlloc(func.getFullName()));
```

```
                        break;
                    case CLASS:
                        PyClass cls = val.getAsClass();
                        out.add(new ClassAlloc(cls.getFullName()));
                        break;
                    case MODULE:
                        PyModule module = val.getAsModule();
                        out.add(new ModuleAlloc(module.getFullName()));
                        break;
                    default:
                        //Do nothing.
                    }
                }
            }

return out;
}
```

Collect the alloc sets for a global or local variable reference:

```
private Set<Alloc> getAllocsForSimpleRef(String context, PyFunction function,
StorageRef ref) {
long revision = this.getCurrentRevision() + 1;

//Global objects don't obey calling context.
if(ref.getRefType() == REF_TYPE.GLOBAL) {
      context = "GLOBAL";
}

Map<String, Map<StorageRef, Set<Alloc>>> currentAllocs;
if(!this.simpleAllocs.containsKey(revision)) {
      currentAllocs = new HashMap<>();
      this.simpleAllocs.put(revision, currentAllocs);
} else {
      currentAllocs = this.simpleAllocs.get(this.getCurrentRevision() + 1);
}

Map<StorageRef, Set<Alloc>> allocsForContext;
if(!currentAllocs.containsKey(context)) {
      allocsForContext = new HashMap<>();
      currentAllocs.put(context, allocsForContext);
} else {
      allocsForContext = currentAllocs.get(context);
}

Set<Alloc> out;
if(!allocsForContext.containsKey(ref)) {
      out = new HashSet<>();
      allocsForContext.put(ref, out);
} else {
      out = allocsForContext.get(ref);
}
```

```
if(ref.getRefType() == REF_TYPE.GLOBAL) {
      out.addAll(injectScopedGlobals(function, ref.getAsGlobalRef()));
}

return out;
}
```

Transfer allocs across a simple variable assignment:

```
private boolean propagateThroughAssignment(PyFunction function, int offset,
StorageRef lhs, StorageRef rhs, String context, long callgraph_revision, long
val_revision) {
      List<Set<Alloc>> lhsAllocs;
      switch(lhs.getRefType()) {
            case GLOBAL:
            case LOCAL:
                  lhsAllocs = Collections.singletonList(
                        getAllocsForSimpleRef(context, function, lhs));
                  break;
            case PROPERTY:
                  lhsAllocs = getAllocsForProperty(context, function, lhs);
                  break;
            case ARGUMENT:
                  lhsAllocs = getAllocsForArgument(context, lhs, offset,
            val_revision);
                  break;
            default:
                  lhsAllocs = null;
}
if(lhsAllocs == null || lhsAllocs.size() == 0) {
      return false;
}
List<Set<Alloc>> prevLhsAllocs = new ArrayList<>();
for(Set<Alloc> allocs: lhsAllocs) {
      prevLhsAllocs.add(new HashSet<>(allocs));
}
int size = 0;
for(Set<Alloc> s: lhsAllocs) {
      size += s.size();
}

switch(rhs.getRefType()) {
      case CONSTANT:
            for(Set<Alloc> allocs: lhsAllocs) {
                  Alloc constAlloc = generateAllocForConstant(context,
                        rhs, function, offset);
                  if(constAlloc != null) {
                        allocs.add(constAlloc);
                  }
            }
            break;
      case ALLOC:
            for(Set<Alloc> allocs: lhsAllocs) {
```

```
                        allocs.add(generateNewAlloc(context, rhs, function,
                  offset));
                  }
                  break;
            case GLOBAL:
            case LOCAL:
                  for(Set<Alloc> allocs: lhsAllocs) {
                        allocs.addAll(getAllocsForSimpleRef(context, function,
                  rhs));
                  }
                  break;
            case PROPERTY:
                  for(Set<Alloc> allocs: lhsAllocs) {
                        for(Set<Alloc> propAllocs: getAllocsForProperty(context,
                  function, rhs)) {
                              allocs.addAll(propAllocs);
                        }
                  }
                  break;
            case ARGUMENT:
                  for(Set<Alloc> allocs: lhsAllocs) {
                        for(Set<Alloc> argAllocs: getAllocsForArgument(context,
                  rhs, offset, val_revision)) {
                              allocs.addAll(argAllocs);
                        }
                  }
                  break;
            default:
                  //Not a potential source.
            }

      int newSize = 0;
      for(Set<Alloc> s: lhsAllocs) {
            newSize += s.size();
      }

      return newSize != size;
      }
```

4.1.7.2.6 Callgraph Analysis

Set the maximum call depth based on the user-provided properties:

```
public void setup(Properties props) {
this.mainFunction = props.getProperty("cg.main_function");
this.maxContextDepth                                                    =
Integer.parseInt(props.getProperty("cg.max_context_depth"));
}
```

Create a new calling context based on a caller/callee function pair and an existing context:

```
public static String buildNewContext(String context, int offset, PyFunction
caller, PyFunction callee, int maxContextDepth) {
String[] components = context.split("\\|");
```

```
String newComponent = caller.getFullName() + "." + offset + " => " +
callee.getFullName();
if(components.length == maxContextDepth) {
for(int i = 0; i < components.length - 1; i++) {
components[i] = components[i + 1];
}
components[components.length - 1] = newComponent;
} else {
String[] newComponents = new String[components.length + 1];
for(int i = 0; i < components.length; i++) {
newComponents[i] = components[i];
}
newComponents[components.length] = newComponent;
components = newComponents;
}

String newContext = String.join("|", components);
return newContext;
}
```

Recover calling contexts based on allocs assigned to the “callee” parameter of the CALL_FUNCTION opcodes:

```
private boolean processContextsForAllocs(ArgumentRef aRef, PyFunction
function, int offset, String context, long pta_revision) {
long revision = this.getCurrentRevision() + 1;
boolean changed = false;

//Get all allocs stored in the function reference under this context.
StorageRef funcRef = aRef.getFunction();
Map<String, Map<StorageRef, Set<Alloc>>> simple_allocs =
      DependencyManager.v().getModule(PTA_NAME)
      .getData("simple_allocs", null, pta_revision);

if(simple_allocs.containsKey(context)) {
if(simple_allocs.get(context).containsKey(funcRef)) {
for(Alloc alloc: simple_allocs.get(context).get(funcRef)) {
//Resolve callee based on the alloc.
PyFunction callee;
switch(alloc.getType()) {
case FUNCTION:
callee =
      DependencyManager.v().getModule(DIS_NAME)
      .getData("function",
      alloc.getAsFunctionAlloc().name);
break;
case CLASS:
callee =
      DependencyManager.v().getModule(DIS_NAME)
      .getData("function",
      alloc.getAsClassAlloc().name + ".__init__");
break;
```

```
default:
//Some objects are callable, and have a "__call__" function.
callee =
      alloc.getAsSimpleAlloc().type
      .getMethods().get("__call__");
}
//If we didn't resolve a callee, try another alloc.
if(callee == null) {
continue;
}

//Create a new context string based on the current context,
//and the callee we identified.
String newContext = CallGraphUtil.buildNewContext(context, offset, function,
callee, this.maxContextDepth);
//Look up or create a set of contexts for the callee.
Set<String> contexts;
      if(!this.contexts.get(revision).containsKey(callee.getFullName())) {
contexts = new HashSet<>();
this.contexts.get(revision).put(callee.getFullName(), contexts);
} else {
contexts = this.contexts.get(revision).get(callee.getFullName());
}
//Add our new context to the callee;
//this constitutes a change for purposes of determing if we're done.
if(!contexts.contains(newContext)) {
changed = true;
contexts.add(newContext);
}
}
} else {
}
} else {
}
return changed;
}
```

4.1.7.2.7 Space Complexity Attack

4.1.7.2.8 Value Analysis

Handle binary operator opcodes, including the “+” operator:

```
private boolean propagateBinaryOperator(PyFunction function, DataTransfer dt,
String context, long cg_revision,
long pta_revision) {
      StorageRef lhs = dt.getLhs();
      StorageRef rhs1 = dt.getRhs().get(0);
      StorageRef rhs2 = dt.getRhs().get(1);
      List<Set<PyValue>> lhsValues = getValuesForRef(lhs, function,
dt.getOffset(), context, cg_revision, pta_revision);
      List<Set<PyValue>> rhs1Values = getValuesForRef(rhs1, function,
dt.getOffset(), context, cg_revision, pta_revision);
```

```
	List<Set<PyValue>> rhs2Values = getValuesForRef(rhs2, function,
dt.getOffset(), context, cg_revision, pta_revision);
	int size = 0;
	for(Set<PyValue> set: lhsValues) {
		size += set.size();
	}

//Value analysis explodes violently if you have arithmetic ops in a loop.
//The value analyzer can detect that data from one iteration feeds into the
next.
//Without extremely complex path sensitivity, it can't tell when to stop.
//Current "solution" is to limit the number of results possible.
if(size >= maxValueSetSize) {
	return false;
}

for(Set<PyValue> as: lhsValues) {
	for(Set<PyValue> bs: rhs1Values) {
		for(Set<PyValue> cs: rhs2Values) {
			for(PyValue b: bs) {
				for(PyValue c: cs) {
					PyValue val = computeBinaryResult(dt.getType(),
				b, c);
					if(val != null) {
						as.add(val);
					}
				}
			}
		}
	}
}

int newSize = 0;
for(Set<PyValue> set: lhsValues) {
	newSize += set.size();
}
return newSize != size;
}

Handle binary operators relevant to strings.
private PyValue computeBinaryString(EXPR_TYPE type, String a, String b) {
	switch(type) {
		case ADD:
			return PyValueFactory.valueOf(a + b);
		case IN:
			return PyValueFactory.valueOf(a.contains(b));
		case NOT_IN:
			return PyValueFactory.valueOf(!a.contains(b));
		case IS:
			return PyValueFactory.valueOf(a.equals(b));
		case IS_NOT:
			return PyValueFactory.valueOf(!a.equals(b));
		default:
```

```
                return null;
            }
        }
```

4.1.7.2.9 Analysis Driver

Execute the application lifecycle. If an exception is caught and teardown requests a retry, restart the lifecycle.

```
public boolean execute() {
this.init();
this.checkRequiredProps();
this.setup();
this.start();
boolean caughtException = this.join();
if(this.teardown(caughtException)) {
this.execute();
this.teardown(this.join());
}
return caughtException;
}
```

Handle analysis teardown, including the logic for the restart_on_error flag. If this flag is set and the caughtException parameter is true, signal for a retry.

```
public boolean teardown(boolean caughtException) {
DependencyManager.v().shutdown();
if(caughtException && this.props.containsKey("driver.restart_on_error") &&
!this.props.containsKey("driver.error")) {
logger.severe("PSA encountered an error; restarting");
this.props.put("driver.error", "true");
return true;

} else {
LogManager.teardownParentLogger(this.sessionID);
return false;
}
}
```

4.1.7.2.10 Result Serialization

Set up the module. This includes determining if verbose mode should be enabled, based on the driver's retry flag and the verbose_on_retry property.

```
public void setup(Properties props) {
if(props.containsKey("serialize.data_dir")) {
      dataDirectoryName = props.getProperty("serialize.data_dir");
} else if(props.containsKey("serialize.use_default_dir") ) {
      dataDirectoryName = props.getProperty("driver.output_dir") +
"/serialized";
} else {
      throw new IllegalStateException("No data directory specified.");
```

```
}
      verboseMode = props.containsKey("driver.error") &&
props.containsKey("serialize.verbose_on_error");
}
```

If in verbose mode, lazily serialize data as it becomes available using separate threads for each module.

```
private class SerializerThread implements Runnable {
private Module m;
private File serializeDir;
public SerializerThread(Module m, File serializeDir) {
this.m = m;
this.serializeDir = serializeDir;
logger.info("Serializer thread launched for " + m.getName());
}

@Override
public void run() {
boolean done = false;
long currentRevision = 0l;
while(!done) {
try {
currentRevision = m.waitOnRevisionIncrease(currentRevision);
} catch(InterruptedException e) {
return;
}
serializeModuleData(m, currentRevision, serializeDir);
if(m.isFinalRevision(currentRevision)) {
done = true;
}
}
}
}
```

Write serialized data to files defined by categories/keys from the specific module:

```
private void serializeModuleData(Module m, long revision, File serializeDir) {
logger.info("Serializing data for " + m.getName() + " revision " + revision);
File moduleDir = new File(serializeDir, m.getName());
moduleDir.mkdir();
if(verboseMode) {
moduleDir = new File(moduleDir, String.valueOf(revision));
moduleDir.mkdir();
}
for(String type: m.getTypes()) {
File typeDir = new File(moduleDir, type);
typeDir.mkdir();
for(String key: m.getKeys(type)) {
File objFile = new File(typeDir, key + ".dat");
try {
objFile.createNewFile();
} catch (IOException e1) {
throw new IllegalStateException("Could not create data file " + objFile, e1);
```

```
}
try (FileOutputStream fos = new FileOutputStream(objFile);
ObjectOutputStream oos = new ObjectOutputStream(fos)){
oos.writeObject(m.getData(type, key));
} catch (FileNotFoundException e) {
throw new IllegalStateException("Failure serializing " + m.getName() + "." +
type + "." + key, e);
} catch (IOException e) {
throw new IllegalStateException("Failure serializing " + m.getName() + "." +
type + "." + key, e);
}
}
}
}
```

4.1.7.2.11 Use Cases

PSA's REST API allows users to do the following:

- Create analysis projects.
- Upload, download, and delete files in a project.
- Launch an analysis on a project.
- Download analysis result files.
- Delete all results for a project.
- Read help docs on the analyses available.

4.1.7.2.12 Architecture

The challenge entrypoint configures an embedded Jetty web server. This configures HTTP Secure (HTTPS) interaction, and launches several servelets that define the REST API. Most of the possible interactions exposed by the REST API are direct operations on the local filesystem; the server maintains a directory structure in its working directory that precisely parallels the Universal Resource Locator (URL) structure defined by the REST API.

On the server, python scripts are grouped by projects; these are simply sub-directories that encompass all scripts relevant to a particular analysis. When the user requests an analysis of a particular project, the server enqueues the request, and waits for any previously-requested analyses to complete. In this way, each analysis can have full command of system resources.

Once an analysis request is dequeued, a RestDriver object is created to service it. The RestDriver class configures the analyzer parameters based on the server environment and the parameters passed by the user as part of the HTTP request. Based on these parameters, the driver dynamically loads one or more analysis modules, configures them, and then launches them all simultaneously.

Each analysis in PSA is a subclass of Module. All modules execute in parallel; they each have their own main threads that get launched by RestDriver at the start of the analysis lifecycle. Each is responsible for generating one kind of data or performing a particular transform on existing data. By referencing data produced by other modules, they cooperate to produce as complete a picture of the scripts under analysis as their algorithms will allow.

Each module is identified by a unique name. Modules reference each other by name through a singleton DependencyManager object. This facilitates decoupling of the different modules into analysis packages, with each package contained in its own java archive (JAR) file. It also allows for overloading of module interfaces; multiple implementations of one module can replace each other if they all share the same name.

In order to facilitate cooperation, each Module maintains an internal revision control system. This allows the module to work on its latest revision, while its dependents read from older revisions without risk of data corruption. The operation for this system is straightforward: a given Module waits on a condition variable exposed by its dependency. When the dependency produces new data, it notifies the condition variable and returns a new revision number. The module can then perform its algorithm on the data contained in that revision until it runs out of new discoveries. It then waits for a new revision and new data.

Because of this non-localization, determining when the analysis is done is more complicated; a given module may have reached the end of its own processing, and there may be no data available from its direct dependencies, but a module on the other side of the network could still be working and could propagate changes that impact the rest of the network.

PSA does not maintain any centralized view of all modules, as this would be costly to maintain. Instead, it provides algorithms for detecting completion through emergent behavior; a module will only declare itself complete based on its observation of its neighbors. The subclasses of Module – SingleDependencyModule and MultiDependencyModule – contain code to handle this sort of complex dependency handling.

Once all modules have detected completion and shut down, the RestDriver joins their threads and terminates the analysis task. In the case of an unrecoverable exception, the faulting module's UncaughtExceptionHandler will signal the rest of the modules for a graceful shutdown.

4.1.7.2.13 Available Modules and Dependencies

PSA comes pre-packaged with a number of analysis packages. A developer is free to write their own by extending Module or one of its direct sub-classes.

- **py_disassembler:** Python-based disassembler. This calls a python script that loads the project modules, and uses Python's own introspection libraries to recover the code structure for the module and its dependencies.
- **py_deserializer:** Optional replacement for the disassembler. This loads serialized analysis state back into each module.
- **py_model_generator:** Augments existing code with extra model constructs to assist analysis. Specifically, this targets opaque parts of the python language – object constructors and super constructors – that need to be tailored to each object in order to bypass limitations of the analyzer.
- **py_cfg:** Control-flow graph generator. For each function, this takes the straight-line code generated by the disassembler and produces basic blocks.
- **py_ir_constructor:** Intermediate Representation generator. Transforms the python opcodes in each basic block and generates semantic data transfers.

- **py_callgraph:** Calling relationship analysis. Uses PTA information to figure out which functions are called at a specific call site, and uses this information to maintain virtual call stacks.
- **py_pta:** Points-to analysis. Tracks the propagation of objects between storage locations. Uses callgraph and value analysis information to improve results.
- **py_value_analysis:** Tracks the propagation and transformation of primitive values between storage locations. Uses PTA and callgraph information to improve results.
- **py_*_report:** Generates a plaintext report for a specific module. These are stored under the project's analysis results directory.
- **py_serializer:** Serializes the final analysis state of all modules. When paired with py_deserializer, can be used to checkpoint an analysis and restart at a later date with additional modules or more intense resolution.

4.1.8 SecurGate

4.1.8.1 Description

The program is a checkpoint video feed backend server that receives a video feed, identifies license plates, and transmits images of the plates over a secure channel to a client that identifies the plate numbers. Since the reference platform does not have a specified video device, a set of images of license plates will be canned with the challenge and a random image from the set will be chosen every time a simulated car passes the checkpoint.

SecurGate contains a **space side channel** vulnerability where the license plate number is the secret that is leaked.

Joint Photographic Experts Group (JPEG) format is used to compress the license plates before transmission. JPEG compression breaks an image into 8x8 pixel blocks and compresses them individually. Normally, all of these blocks will be appended and stored in a file. The checkpoint software was designed before storage was abundant, and instead of storing 8x8 blocks in a file, each block is encrypted and sent to a text-recognition client (not part of the challenge). Because of how JPEG compression works, the size of the compressed block is proportional to the amount of high-frequency data in that block. As a result, 8x8 blocks that contain abrupt changes will be larger than blocks that contain color gradients. Figure 8 shows the input to the JPEG algorithm (left) and the output of the side-channel (right).

Figure 8. SecurGate Video Feed for License Plate Identification

The output was generated by a proof-of-concept python script. The output is also 1/8 of the width and height of the source image, so the original text needs to be large enough to stay readable after the size-reduction.

4.1.8.2 Software Design

4.1.8.2.1 Related Class Roadmap

Figure 9 depicts the SecurGate class roadmap.

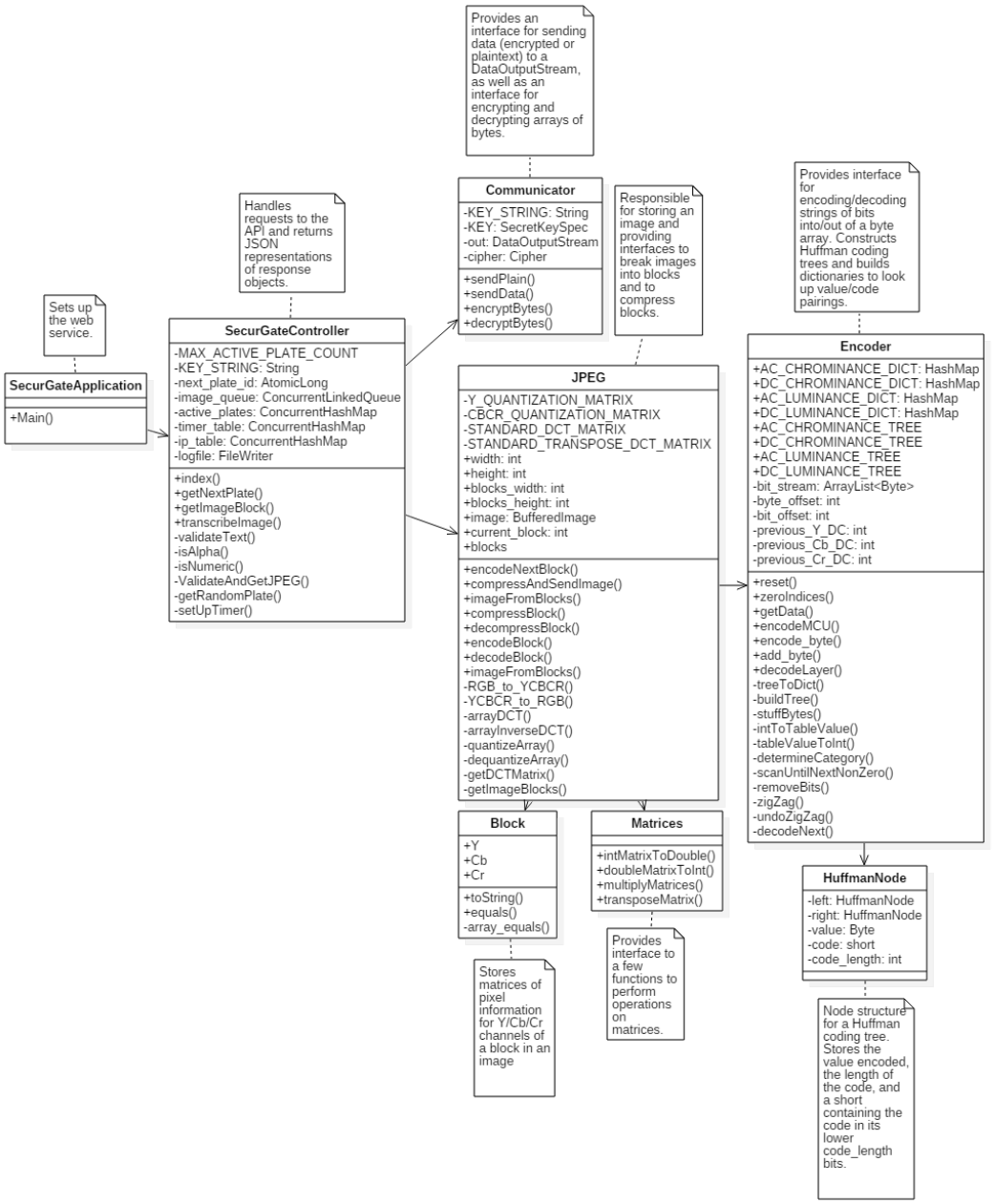


Figure 9. SecurGate Class Roadmap

Classes used during vulnerable execution:

- **SecurGateController:** Handles HTTPS requests, opens license plate images, and sends the encrypted and encoded blocks one at a time to the user.
- **JPEG:** Stores information about an image, and provides interfaces to JPEG compress and decompress the image block by block.
- **ImageBlock:** Stores information in responses to requests at /get_image_block.

Vulnerability code

J**PEG.java:** The algorithm encrypts and sends each block individually, instead of concatenating the blocks together and encrypting the whole file at once. Because of this, an adversary can individually see the length of the data of each block, and can thus determine if the block contains high or low frequency content.

4.1.8.2.2 Challenge Design

Use Cases

The application is an interface for receiving license plate images from a toll gate and sending transcribed image text back to the server. The application supports the following interactions:

- Getting the ID and access token for the next license plate in the queue. Once the plate is de-queued, the user can request blocks of data from it.
- Getting a block of encoded and encrypted image data from the block with a given ID.
- In the case of a user not accessing an image for 1 minute (likely due to an error on the client side), the image is inactivated and added back to the queue (in order to prevent memory leaks from opening images then not reading them to completion.)
- Transcribing image content. A user who decodes the image data can transcribe the plate text and sent it to the server to be logged.

4.1.8.2.3 Overall code structure

Components that don’t participate in vulnerable operation:

- **PlateID:** Class to store info for responses to requests at /get_next_plate.
- **FailedToSendException, EncodingExceptions, EncryptionException:** Classes for exceptions during sending data, encoding image data, or encrypting data.
- **Matrices:** Provides some helper functions to operate on two-dimensional (2D) arrays.
- **HuffmanNode:** Representation of a node in a Huffman Coding tree, used to generate the lookup table for JPEG Huffman codes.
- **Encoder:** Provides interface to Huffman code image data and store bits into an array of bytes. Also can decode bits from the array to get back the image data.

- **Communicator:** Provides interface to encrypt and decrypt data.

Libraries Used:

Spring Framework and Spring Boot – Web framework used to handle the web interface to the service. External to the application.

4.1.8.2.4 Inputs and Outputs

Inputs:

- Connections to the /get_next_plate endpoint.
- The id/token passed to /get_image_block?id=<id>&token=<token>.
- The id/token/encrypted_message parameters passed to /transcribe_image?id=<id>&token=<token>&enc_text=<encrypted + base 64 encoded plate text>.

Outputs:

Responses from the /get_next_plate endpoint follow the following format:

On normal request:

```
{
"id":<id of next queued plate>,
"width":<image width>,
"height":<image height>
}
```

On request when max number of active plates is reached:

```
"Maximum number of active plates exceeded. Please wait for plates to expire."
```

On request when an active plate is already opened for the current internet protocol (IP) address:

```
"There is already a plate opened for your IP."
```

On any other error:

```
"Something went wrong when opening a new plate."
```

Responses from the /get_image_block endpoint follow the following format:

On valid request:

```
{
"data":"<base 64 of encoded and encrypted block, null if endOfBlocks is true>",
"endOfBlocks":<true iff there are no more blocks>
}
```

On invalid id/token pair:

```
 "Invalid ID/token pair"
```

On request for valid plate whose timer has expired:

```
"Plate has expired"
```

Responses from the /transcribe_image endpoint follow the following format:

On enc_text being null:

`“enc_text required”.`

On incorrect id or token:

`“Invalid ID/token pair”.`

On invalid encrypted + base 64 encoded text:

`“enc_text invalid”.`

On invalid decrypted text format (wrong size, or invalid characters):

`“Text’s characters or length invalid”.`

On valid input:

`“OK”`

The documentation of inputs and outputs can also be found by visiting https://<server_ip>:8443 while running the service.

4.1.9 DoorMaster

4.1.9.1 Description

The program is a key-fob management and authentication system for a door access control. It contains a **time side channel** vulnerability where the key-fob private key is the secret.

The challenge pings a network node one ping per second as a safety measure in case the software has been cut from the network. If the challenge does not get a pong, it goes into the shutdown mode and does not authorize any key-fob until manually restarted. The adversary added code to modulate the authorization data (private key) on the delay between the pings. The data is transported in such a way that a single observation of the side channel may be too noisy to reconstruct 1 bit of the data, but multiple observations allow reconstruction of the data.

4.1.9.2 Software Design

4.1.9.2.1 Vulnerability Design

The vulnerability design is depicted in Figure 10.

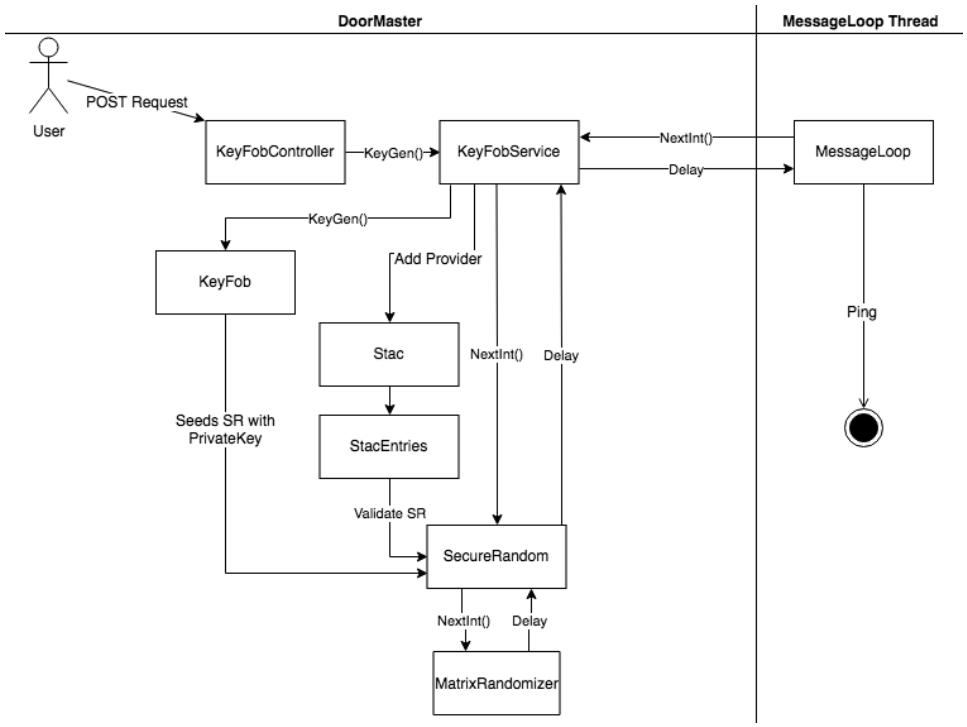


Figure 10. DoorMaster Vulnerability Design

In DoorMaster, the vulnerability is hidden in the delay between the pings. Upon start up, the MessageLoop thread pings the network server every second to make sure that the connection is present. However, once a key is generated upon user request, a fake SecureRandom instance is created using the Stac provider, and is seeded with the binary representation of the private key. MessageLoop then delays 0.15 seconds if the next value in the seed is a 0, and 0.3 seconds if the next value is a 1. Those values are able to be changed.

Following are several code snippets related to the implementation of the vulnerability:

```
public void run() {
    System.out.println("Ping thread is running.");
    while(true) {
        try {
            KeyFobService.conn = ping(); //Contact the server
            flag = KeyFobService.flag; //Identifying whether the KeyFobService has been initialized.
        } catch (InterruptedException e) {}
        try {
                Thread.sleep((flag ? KeyFobService.nextInt() : 1000 ));
        } catch (InterruptedException e) {}
    }
}
```

Shown above is the loop that contacts the server, and determines how long to wait. The Boolean “flag” is set true as soon as the first key is generated, and KeyFobService.init() is called.

```
//Initialize the random provider
public static void init() {
    Security.insertProviderAt(new Stac(), 2);
    try {
        srInstance = SecureRandom.getInstance("SHA1PRNG", "STAC");
    } catch(NoSuchAlgorithmException | NoSuchProviderException e) {
        throw new RuntimeException("Missing STAC crypto provider.");
    }
}

//Generate a random value
public static int nextInt() {
    x = srInstance.generateSeed(0);
    String s = Byte.toString(x[0]);
    return ((Integer.parseInt(s)+1)*150);
}
```

The two preceding methods are within the KeyFobService class. Init() inserts Stac as a SecureRandom provider, and nextInt() returns the delay for the MessageLoop class based on the next bit of the fake SecureRandom's seed.

```
public KeyFob(String name) {
    super();
    //Generate KeyPair
    KeyPairGenerator keyGen;
    SecureRandom random;
    try {
        keyGen = KeyPairGenerator.getInstance("RSA");
        random = SecureRandom.getInstance("SHA1PRNG");
        keyGen.initialize(512,random);
        this.kp = keyGen.generateKeyPair();
        this.pvt = kp.getPrivate();
    } catch (NoSuchAlgorithmException e) {}
    String X = new BigInteger(1,pvt.getEncoded()).toString(2);
    byte[] bval = new BigInteger(X,2).toByteArray();
    KeyFobService.srInstance.setSeed(bval);
    this.setPubKey(kp.getPublic());
    System.out.println(name + " created."); //Confirmation
}
```

Featured above is the constructor for a KeyFob object. Upon creation, the SecureRandom srInstance's seed is set to be the private key of the newly created KeyFob.

```
synchronized public void engineSetSeed(byte[] seed) {
    temp = temp + new BigInteger(1,seed).toString(2);
}
```

Setting the seed simply concatenates the key onto a String containing other keys, as shown in engineSetSeed() above.

```
public byte[] engineGenerateSeed(int num) {
    byte[] output = new byte[1];
    String var = null;
    if(count == temp.length()) {
        count = 0;
    }
    var = Character.toString(temp.charAt(count));
    count++;
    int[] arr = MatrixRandomizer.Matrix(var);
    output = new byte[] {(byte)arr[0]};
    return output;
}
```

EngineGenerateSeed() is a method within the fake SecureRandom class that traverses along the temp string, which has been set to match the list of private keys, outputting one bit (as a byte) at a time.

4.1.9.2.2 Use Cases

The DoorMaster application emulates a KeyFob authentication server used to garner entrance through a secure door. As long as both the DoorMaster server and the Server that handles PublicKey translation look-up are active, then the user's REST requests will proceed smoothly. A POST request to the proper address will create a new key. A POST request to the auth URL is equivalent to attempting entrance into the door, and will return either true or false. A DELETE request can be used to remove access from a particular key.

4.1.9.2.3 Overall code structure

Dependencies

DoorMaster has the third-party dependencies listed in Table 2.

Table 2. DoorMaster Third-Party Dependencies

Dependency	Internal/external?	URL
Spring Framework	External	https://projects.spring.io/spring-framework/
Spring Boot	External	http://projects.spring.io/spring-boot/
Apache Commons Codec	Internal	https://mvnrepository.com/artifact/commons-codec/commons-codec/1.9

The DoorMaster application consists of the following classes:

- StacDoorMasterApp: Initializes the Spring framework and allows arguments to be passed.
- DoorMaster: Handles the arguments that are passed: Arg1 = portNo (default 5056) and Arg2 = ipAddr (default 127.0.0.1). Initializes the KeyFobService class. Spawns a thread for the MessageLoop class.
- MessageLoop: Establish the socket connection with the Server. Ping the Server with the required delays, as described in the vulnerability section.

- KeyFob: The KeyFob object, which contains a KeyPair. That KeyPair is properly generated with an actual SecureRandom instance, but its PrivateKey is used to seed a fake SecureRandom class, as described in the vulnerability section.
- KeyFobController: Handles the REST API requests, and maps them to the appropriate method in the KeyFobService class.
- KeyFobService: The primary class of the DoorMaster application.
    - Contains a mapping of KeyFobs via key_id.
    - Contains methods for loading, saving, encrypting, and decrypting using PublicKeys and PrivateKeys.
    - Contains the HandShake method which uses proper encryption and decryption to allow or deny entrance through a door.
    - Contains the method that initializes the fake SecureRandom Stac provider.
    - Contains the method nextInt that MessageLoop calls to calculate the proper delay based on the PrivateKeys' binary strings.
    - Handles communication with the Server's mapping based on key generation and key deletion.
    - Stac: Part of the procedure to add Stac as a valid provider.
- StacEntries: Maps SecureRandom to Stac to allow for utilization of custom methods.
- SecureRandom: Extends the SecureRandomSPI. SetSeed assigns the "temp" variable to the binary version of a KeyFob's PrivateKey. Each PrivateKey is appended onto the temp string. GenerateSeed will return the next bit each time it is called.
- MatrixRandomizer: Helper class used to add "extra randomization" to the bytes. In reality it is essentially a No-Op, solely there to add fake complexity.
- Server: A separate server that holds a mapping of PublicKeys based on key_id. This server must be up and running, with a proper connection to the DoorMaster server, in order for the DoorMaster application to run successfully.

4.1.9.2.4 Inputs and Outputs

DoorMaster is built upon a simple REST API:

- Send a POST Request to http://localhost:8080/keyFobs/ to create a key
- Send a POST Request to http://localhost:8080/keyFobs/{base 64 encoded private key} to attempt authorization
- Send a DELETE Request to http://localhost:8080/keyFobs/{ base 64 encoded private key} to attempt deletion

Refer to the DoorMaster Sample script in BT/challenges for the complete syntax.

Outputs will be displayed on the console that is running the DoorMaster Application, and oftentimes in the application used to send the REST request as well.

Additionally, the application is also transmitting Internet Control Message Protocol packets, and those are monitored in order to obtain the vulnerable information.

4.2 Engagement 6 Challenge Programs

4.2.1 CaseDB

4.2.1.1 Description

CaseDB is a server that hosts criminal case records. It allows users to store case records and to retrieve collections of case records, where a single case can have many associated records. To insure data integrity, CaseDB employs a data redundancy algorithm that works by duplicating each case record across a subset of the backing stores. Retrieval of requested case records from the backing stores is facilitated by a load balancing algorithm, which informs the client from which backing store each individual record will be retrieved.

CaseDB contains a space side channel vulnerability. The secret is the identity of the informant associated with a case whose records were requested by a legitimate privileged user.

An attacker on the network can send requests as a non-privileged user to CaseDB. Additionally, from his vantage point on the network, the attacker can passively observe a legitimate user's request for case records and the responses from CaseDB back to the user. However, the attacker cannot observe the actual plain-text records in transit as that portion of the communication is encrypted.

The vulnerability exists in CaseDB's load balancer. The load balancer is responsible for creating a retrieval plan that prescribes how the records required to satisfy a user's request will be retrieved from CaseDB's various backing stores. When a client requests a collection of case records, the load balancer constructs and transmits back to the client a retrieval plan which the server will follow to effectively balance the requested load across the backing stores. The retrieval plan is sent back to the user in chunks (one chunk per packet), where each chunk specifies a backing store and the records to be retrieved from it. Each chunk is simply a list of indices denoting the positions of the relevant records within the specified backing store. For example, assuming two backing stores A and B, a retrieval plan might come back as two chunks containing [A:12,21] and [B:35] respectively. This retrieval plan tells the client that records 12 and 21 will be sent to it from backing store A, and record 35 will be sent to it from backing store B.

The plan sent to the client is purely informative in nature. Once the plan is transmitted, the server immediately follows through on the plan by automatically transmitting the requested records from their respectively prescribed backing stores to the client.

The load balancer caches computed retrieval plans. That is, in cases where a request's query-string has already been seen previously, the cached version of the retrieval plan is used to avoid unnecessary re-computation. Another quirk of the load-balancer is that when a request's query-string has clauses separated by disjunctions, each clause is treated as a distinct request and is given its own entry in the cache. After the retrieval plans for each individual disjunction are computed or retrieved from the cache, the load-balancer merges them in a predictable way into a single overall balanced retrieval plan that gets sent back to the requesting user.

When a privileged user requests the records associated with a case, one of the chunks of the retrieval plan delivered back to him will contain an entry for the case's secret informant record. If the attacker replays the privileged user's request then the load balancer will send back to him the cached request plan that was previously computed for the privileged user, except a guard will strip out any retrieval plan chunk entries corresponding to secret records to which the non-privileged attacker does not have access. In other words, the attacker would get back the same plan that the privileged user did minus the entry corresponding to the case's secret informant record.

By using knowledge of the load balancing algorithm, the attacker can additionally figure out how an arbitrary request for only public records would be load-balanced. Using this capability, the attacker can create a request with multiple disjunctive clauses where only one of the clauses collides with the privileged user's prior request, but where the other clause(s) are structured in a way that changes the resulting merged retrieval plan in a predictable manner. That is, the attacker can add a disjunctive clause for which the corresponding record would be planned for retrieval from backing store A in the absence of a secret record, but whose corresponding record instead gets planned for retrieval from backing store B due to the presence of the secret record in the cached retrieval plan at merging time. Since the attacker can also add his own arbitrary records to CaseDB, he can use this ability to binary search for the secret informant by adding new queries, with lexicographical ordering, for those records and observing whether they get bumped from their expected position in the plan to some other backing store.

4.2.1.2 Software Design

CASEDB is built as a distributed application architecture. This architecture has three primary modules: the CASEDB Server Module, the Storage Node, and the Remote Client. To achieve robustness, the architecture is built on a distributed framework called Apache Camel.

Apache Camel is a message routing and component wiring framework. It implements patterns for linking distributed enterprise applications, called Enterprise Integration Patterns. These patterns are accessed by programmers via a pure Java API that allows them to define all intermediary points between discrete, and potentially distributed, Java class components called Beans. Beans can either be just Plain Old Java Object instances (POJOs) or wrappers to remote network services. They implement the message processing components, but not the message routing and queueing aspects. For STAC, this means all CASEDB logic is implemented as Beans and the glue logic is implemented as Java calls to the Camel Routing API.

A modified Java datastore called CQEngine is used. CQEngine is included as part of the CASEDB application as a Java library running in the same java virtual machine (JVM) memory space as CASEDB, not an external service. CQEgine is an in memory database that uses Java Collections as its store, and enables SQL searches over Collection objects. Some modules/classes of CASEDB are developed as direct additions to/modifications of the CQEngine source, as is noted in the descriptions below.

The following is a description of the three main components of CASEDB and their interconnectedness; see Figure 11.

- The CASEDB Server Component: The CASEDB Server Component handles all query processing. It also tracks record locations and node instances. This information is used to

satisfy client requests and make query plans, which it returns to the requesting client. There is only one CASEDB Server Module instance per CASEDB application.

- The Case Node: Each Case Node module holds a subset of the total records. In CASEDB, the number of Case Nodes is hardcoded to five. The nodes and their contents are tracked by the CASEDB Server.
- The Case Client: The client interfaces with users and turns their requests into queries that it sends to the Server. It also retrieves the files that satisfy those queries from CASEDB Nodes.

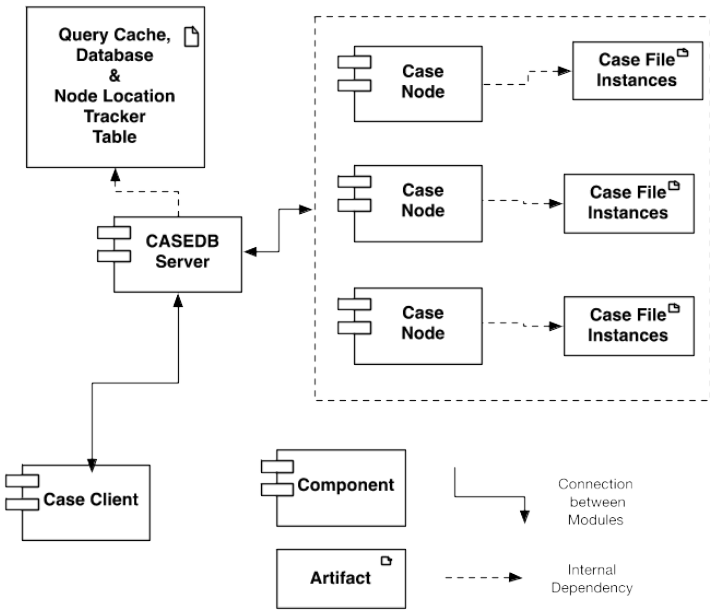


Figure 11. CaseDB Main Component Interactions Overview

Class Interaction Diagram

A class diagram for CASEDB is shown in Figure 12. This diagram depicts the important interactions between classes/modules of the CASEDB application and acts as guide to the CASEDB code. Descriptions for each of these classes/modules is provided below.

CASEDB Server: The server module is the main listening interface of the CASEDB application. It waits for queries to come from the client and forwards them on to the internal query processing classes of the CASEDB application. The server module is implemented using the APIs of Apache Camel. Camel provides routing functionality for messages sent to the server.

All CASEDB Server functionality is located in the package *com.ainfosec.casedbcamel.svr*.

Routing Implementation Module: This class uses the Apache Camel library APIs to implement components that route incoming requests to their respective handlers. This CAMEL API implements pure Java logic to make connections reliable and decoupled. For example, the API supplies a method that implements a queue in between components to provide backpressure

support, thereby allowing two components interacting on either end of the queue to run as independent threads that asynchronously process data.

The Routing Implementation Module functionality is located in the package *com.ainfosec.casedbcamel.svr.routing*.

Query Handler: The query handler module is responsible for tracking query state and decomposing queries into parts that may become separate queries, like when a disjunction is found.

The Query Handler Module functionality is located in the package *com.ainfosec.casedbcamel.svr.queryhandler*.

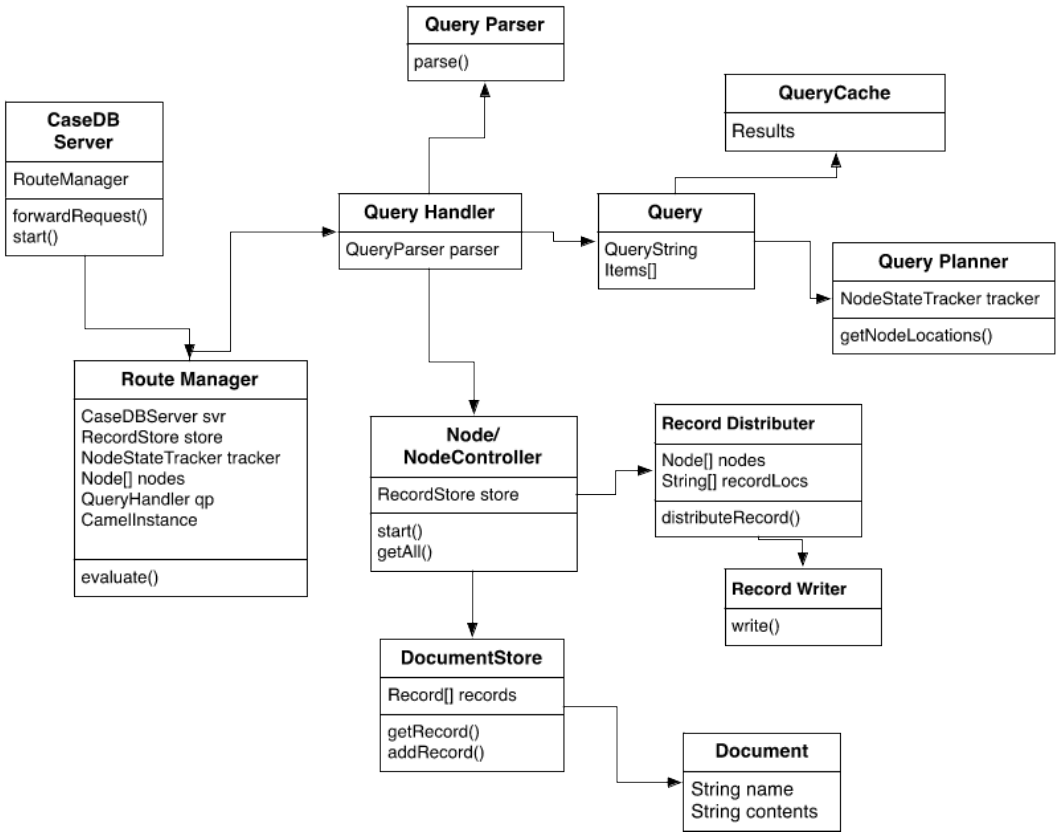


Figure 12. CaseDB Class/module Interaction Diagram

One class file exists in this package for each type of query handled by CASEDB. The three classes that are important for implementing the vulnerability are the AddDoc class, the GetCase class, and the SearchDoc class. See Section *Inputs and Outputs* for the descriptions of these three types of queries.

Query Parser Module: Parses incoming queries using a subset of SQL. The parser utilizes an external JavaCC SQL parser and implements the portions of SQL it needs via a visitor pattern.

Code for the query parser module is found in the *com.ainfosec.cqengine.query.parser.common* package. The query parser is part of the CQengine. Two parser implementations are included, one for the CQN language and one for SQL, but CASEDB only uses the SQL parser. The grammar files for these languages can be found in the folder:

EL/source/CASEDBService/src/main/antlr4/com/ainfosec/cqengine/query/parser/sql/grammar/.

Query Planner: This module is responsible for matching the records requested to their locations on Nodes.

Code for the Query Planner is found in two locations:

1. com.ainfosec.cqengine.query.QueryPlanner.class
2. com.ainfosec.cqengine.resultset.order.ResultPlanner.class

The QueryPlanner class is the basic planner. It selects Nodes to retrieve records from and creates lists of records to be retrieved for each node.

ResultPlanner is a specialized version of the QueryPlanner. It works on queries with ordered result sets -- i.e. queries where the user includes the "ORDER BY" clause. **This specialized version of the QueryPlanner contains the logic necessary to enable the vulnerability**. If the attacker does not order the results with an ORDER BY clause, they will not be able to perform the binary search needed to find the secret name of the informant.

Note that the Queryplanner and the ResultPlanner only differ in the ordering imposed on the elements of the plan -- i.e., the order in which the results of each Node's planning chunk are returned. The ordering does not actually affect how the plan is made or what nodes are selected to retrieve the records. In other words, the contents of each plan chunk do not change, just the order of the elements in each chunk. This is important for enabling the vulnerability because it means that a privileged user's query does not need to be ordered because ordering does not affect the overall length of each Node's record list in a plan.

Query Cache Class: This class stores already developed Query Plans to avoid need for recalculation.

The com.ainfosec.cqengine.query.QueryResult class implements the Query Cache.

CASEDB Node & Node Controller:

- **Node (i.e. a backing store):** An instance of a remote node for storing records. There may be many of these, but in practice there are five.
- **Node Controller:** Controls access to Nodes, tracks state of requests, and holds intermediate results.
- Node and Node Controller implementation are located in package *com.ainfosec.nodemgr*.

Record Distributer: This class decides how records are written to the distributed Nodes.

The Record Distributer is located in the *com.ainfosec.nodemgr.DocDistributer* class.

Record/Document Manager Module: The logic that controls, stores, and secures CASEDB Records/Documents. This module has the three sub-modules:

- **Doc Writer:** This class writes document records to a permanent store on the Nodes.
- **Doc:** A class containing a document with information about a case.
- **Doc Controller and Secure Doc Controller:** Classes that controls access to a permanent storage place for documents.

The document management module classes are located in the *ainfosec.docmgr* package.

TableVulnerability Flow

The following collectively describes how the vulnerability manifests in the CASEDB code and how an attacker triggers the vulnerability to learn the ID of the secret informant record.

1. **Privileged User (Victim) makes a Get query**: The Get request is a query for all records, including the secret informant record associated with a specified case. The Get request is not encrypted, so the attacker can observe it during transmission.
2. CASDEDB parses query: The parser breaks the Get query's disjunctions up into separate subqueries if necessary.
3. CASEDB Server collects result set of record IDs: For each disjoint clause of the query, the server checks to see if the clause is in the cache. If so, the server retrieves the corresponding record IDs from the cache. If the disjoint clause is not in the cache, the database is queried directly to obtain the corresponding record IDs. The complete result set for the query is generated as the union of the disjoint query clauses' individual result sets.
4. CASEDB Server calls Query Planner Module to calculate a query plan: The planner module receives the complete result set of record IDs, then generates the retrieval plan for them in a predictable way. In particular, the string representation of a record's ID seeds a random number generator from which the first number is drawn (modulo the number of Case Nodes) to determine the default Case Node from which that record is to be retrieved. After calculating the default Case Node associated with each record ID in the query's result set, the Planner checks to see if the Plan is balanced. If too many of the query's returned records share the same default Case Node, the Planner re-balances the plan using a deterministic algorithm. The major take-away here is that CASEDB's entire planning process is completely predictable when you know the IDs of the records that will be returned by a given query. Given the Record IDs returned by a query, a user with knowledge of CASEDB's internal workings can work out the retrieval plan that CASEDB will generate with 100% accuracy.
5. CASEDB Server sends Query Plan to privileged user: For each Node referenced in the overall retrieval plan, the Planner sends back to the user a single chunk (i.e. packet) containing the name of the Case Node and the list of records that will be received from that node. Each returned chunk is encrypted. Each chunk may be of a different size; this is the observable of the space side channel. For example, assuming two Nodes A and B, a retrieval plan to return three records respectively having IDs "Parties", "ArrestReport", and

"Suspects" might come back as two separate chunks containing [A:"Parties", "ArrestReport"] and [B:"Suspects"].

6. Attacker Makes Same Query as Privileged User (Active Operation): The attacker replays the same query.
7. CASDEDB Parses query: Same as #2.
8. CASEDB Server collects result set of record IDs: Since the attacker's query is a replay of the victim's query, all the query's clauses have cache entries. Since the cache entries were populated with the results of the privileged user's query, the result set retrieved from the cache contains the secret informant record ID. Note that if the unprivileged attacker had instead sent a different query that didn't already have a cache entry, the back-end database lookup would be smart enough to not return any secret record IDs in the result set.
9. CASEDB Server calls Query Planner Module to calculate a query plan: The Query Planner generates the plan for the records in the result set, including the informant record. Since the result set is the same as it was for the privileged user's query, the Query Planner generates the exact same plan as before. Then, prior to transmitting the plan to the client, a guard notices that the attacker isn't privileged and therefore it strips out the Chunk entry for the secret informant record.
10. Attacker Learns Which Case Node the Secret is Retrieved From: By comparing the new plan's chunk sizes to those that were sent to the privileged user, the attacker determines from which Case Node the protected informant record was delivered to the privileged user.
11. Attacker Performs Binary Search:
    a. The attacker uses Add requests to add new records, one record per Add request, to the case containing the sought-after secret informant record. The goal is to continue to add new records until you're able to add one whose default Case Node is the same as the Case Node from which the secret informant record was delivered to the privileged user. Moreover, the newly added record should have a lexicographically "early" record ID; i.e., a good record ID to add is "A", not "ZZZZZ".
    b. Once the attacker successfully adds such a record to the database, he then submits a new Query request that is the same as the original replayed query except with an added disjoint clause that 1) requests his newly added record and 2) adds the 'ORDER BY' keyword that causes the returned result set to be lexicographically ordered.
    c. The attacker observes the sizes of the plan chunks returned to him. Using his knowledge of how retrieval plans are constructed, the attacker knows if the insertion of the new record will force the load balancer to re-plan the retrieval of some record from the secret record's original retrieval Case Node to a different Case Node. Moreover, by adding the 'ORDER BY' keyword to the query, the attacker knows that if the load balancer does re-plan retrieval of a record to a

different Case Node, it would be the source Case Node's lexicographically last record that is re-planned.

i. By observing the retrieval plan's chunk sizes that subsequently come across the wire, the attacker can check if it was the secret informant record or some other record that got bumped from the secret record's default Case Node to some other node. He does so by checking if the retrieval plan Chunk that he expects to grow by one record due to the re-balance (using his knowledge of how the planner deterministically re-balances) does indeed grow by one record.

1. If it does grow, then the record that was bumped from its default retrieval Case Node to some other node is necessarily a non-secret record (since it wasn't subsequently stripped out by the guard).

2. If it doesn't grow, then it was the secret informant record that got bumped from its default retrieval Case Node to some other Case Node. After being re-balanced into the Chunk for a different Case Node, the guard catches that it is a secret record ID and strips it back out of the Chunk before transmitting the Chunk to the attacker, thus the Chunk's size doesn't change.

ii. By observing whether it was the secret informant record or a non-secret record that was re-planned into a different plan Chunk, the attacker learns whether the secret record's ID lexicographically precedes or succeeds the lexicographically last non-secret record in the secret record's original retrieval plan Chunk.

d. Using his refined knowledge of the lexicographic placement of the secret informant record ID, the attacker repeatedly performs steps 11a through 11c while Adding progressively tighter record ID's in step 11a to narrow down the secret record ID. This is ultimately just a binary search.

Inputs and Outputs:

CaseDB supports three types of user requests: an *add* request, a *query* request, and a *get* request. In the case of the *query* and *get* requests, retrieval plan data is returned to the client asynchronously to port TCP:6166 on the client, then the records themselves are asynchronously transmitted to the client on port TCP:6167. In the case of the *add* request, an asynchronous response is returned to TCP:6166 that contains the ID of the added document.

For all request types, a synchronous response is also returned to the client that includes an error message if one occurs, or is empty if no error occurs.

Note: The data in all packets below is Base 64 encoded prior to transmission. No Base 64 padding is required.

Add Request:

The *Add* Request is sent from the Client to the Remote Node. An *Add* request includes a string containing the text data of the document to be added (see Table 3).

Table 3. CASEDB Add Request Format

1 byte: request type (a 0 for Add Case)	1 byte: length of username	variable bytes: username	1 byte: length of authen-tication info: x	8 bytes: initialization vector	x-8 bytes: encrypted password	1 byte: length of data	variable length: The document data

Add Request Response:

The ID of the added document.

variable number of bytes: A string containing the ID of the added document. One byte per character in the string.

Query Request:

The *Query* Request is sent from the Client to the Remote Node. A *Query* request includes a query string. The server returns a plan (multiple packets) for retrieving all records matching the query string from the nodes, and then it returns the records too (see Table 4).

Table 4. CASEDB Query Request Format

1 byte: request type (a 2 for Get Case)	1 byte: length of username	variable bytes: username	1 byte: length of authen-tication info: x	8 bytes: initialization vector	x-8 bytes: encrypted password	4 bytes: length of request	variable length: The request query

Asynchronous Query Request Response: One or more responses are sent for each query request to port 6166 and 6167.

Each response to port 6166 includes a portion of the query plan. Assuming x number of parts in the plan, there are x responses, where each response gives the retrieval plan for a particular node. Because the packaged CaseDB server is configured for 5 nodes, expect 5 of these response packets per query.

Each individual response to 6166 includes multiple results, one for each document record that is expected to be retrieved from a particular node. This information is encoded in the last 2 fields which repeat with a 'Result length' followed by a 'Result value' for each document record (see Table 5).

Table 5. Asynchronous Query Response Format

4 bytes: Message Id int (always 1, placeholder)	4 bytes: message number int (y out of x messages, this field contains the y value)	4 bytes: total messages int (y out of x messages, this field contains the x value)	4 bytes: Node ID. The ID of the node this response corresponds to	4 bytes: number of results. Indicates how many results are in packet	Result length (variable number of lengths) 4 bytes: length of result: z	Result value (z length bytes) z bytes: The name of the retrieved document

Similarly, for port 6167, there are x responses, with a response for each part of the plan. The responses to 6167, however, include the contents of each document as retrieved from the nodes and not the query plan. Each response has the following format per document content entry. A response may contain multiple document content entries concatenated together. Each response to port 6167 corresponds to a query plan portion response sent to port 6166. If a query plan portion contained multiple document references, then a corresponding response below will contain an equal number of concatenated document content entries (see Table 6).

Table 6. Response Format per Document Content Entry

multiple bytes: String representation of document ID	1 byte: ':' character as a delimiter after the doc ID	multiple bytes: String with document contents. May be encrypted. If encrypted, first byte is the initialization vector	2 bytes: ';;' characters as a delimiter after the doc contents

Get Request:

The Get request is sent from the Client to the Remote Node. A Get request includes the name of a case. The server returns a plan (multiple packets) for retrieving all records for that case from the nodes, and then it returns the records too. The Get request is just a convenience wrapper. When you send the Get request, the server generates an appropriate SQL query that will retrieve all records associated with the specified case. The same type of query could be submitted directly by the user via a *Query* request (see Table 7).

Table 7. Get Request Format

1 byte: request type (a 2 for Get Case)	1 byte: length of username	variable bytes: username	1 byte: length of authen-tication info: x	8 bytes: initialization vector	x-8 bytes: encrypted password	1 byte: length of request	variable length: Case Name

Get Request Response: Same as *Query Request Response*

Synchronous Error response:

Returned to the requesting client port for all requests.

Variable length, empty if no error

4.2.2 Chessmaster

4.2.2.1 Description

Chessmaster is a game server for chess enthusiasts. Players can play chess against an Artificial Intelligence (AI) after connecting to a central server. More specifically, players may start a new game and input moves to the server after first supplying a username and password. The server is responsible for setting up a new game, keeping track of past moves and pieces taken, and making informed moves against the player. Additionally, the program supports a console interface that displays the board and accepts input.

Chessmaster exhibits a time-complexity vulnerability.

The Chessmaster AI's move algorithm is vulnerable to a time-complexity attack that arises from an irreconcilable boardstate. Like in regular chess, once a pawn reaches the opposite edge of the board, it is eligible for a promotion to either a knight, bishop, rook, or queen. Due to a bug in an autocorrect function, an adversary is able to promote passed pawns to kings.

To make a move, the player supplies input according to algebraic notational guidelines. When promoting a pawn, the player appends the name of the piece they would like to promote to. The capitalization independent string "king" is specifically disallowed. However, an autocorrect feature accepts player input two Levenshteinian distance [5] away from their entry. This allows an adversary to bypass the guard and promote a pawn into a king.

Under normal operation, Chessmaster's move algorithm performs a depth-first adversarial search to decide what move to make. To evaluate a state in the search tree, the algorithm analyses all possible moves from a given position, and gives each a score. For example, capturing a pawn would be worth a few points, capturing a queen would be worth many points, and taking the enemy king would be worth the maximum amount of points. For each of these moves, the algorithm then evaluates all of their possible next moves.

Since the number of leaves in the search tree is exponential, the algorithm is bounded by a determined depth. For each call to the search function, the depth is determined as a function of the total difference in points between each color. For example, if there are two kings and one pawn left on the board, the difference will be set to 20,000 and the depth will be set to 3. In the case where a pawn has been promoted to a king, the depth determining function will set the depth to an arbitrarily large number. In turn, the move searching algorithm will search for a prohibitively long time, locking up the main thread.

4.2.2.2 Software Design

The vulnerable class design is depicted in Figure 13.

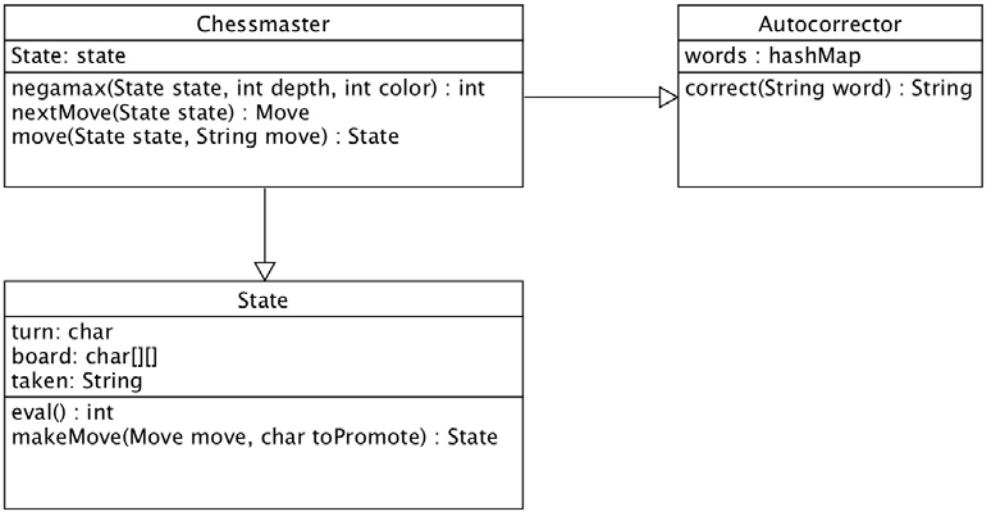


Figure 13. Vulnerable Class Diagram

Chessmaster class

The **Chessmaster** class is the entry point responsible for running the Chessmaster program. It is responsible for keeping track of the current boardstate and interfacing between the player. The chess opponent AI is additionally located here, with function definitions for minimax search and making piece moves. The Chessmaster vulnerability entry point is located here in the "move" function that allows a player to make a move. By entering a pawn promoting move, and then a misspelling of "king", the Chessmaster program allows the player to promote a pawn to a king that subsequently triggers the time-complexity vulnerability.

State Class

The **State** class is responsible for keeping track of any given boardstate. While no vulnerable code is located here, the eval() and depth functions are key in exploiting the vulnerable code in the **Chessmaster** class.

Autocorrector Class

The Autocorrector class is responsible for providing a basic autocorrecting service for the chessmaster program. It is used when bypassing the vulnerable guard in the **Chessmaster** class.

4.2.2.2.1 Vulnerability code

Step 1: Bypass of pawn promotion guard

Pseudocode:

```
If the player's move is a pawn promotion:
While input is not valid:
Get player input
If player input is "king":
Continue
```

```
Otherwise:
Autocorrect player input
Make a move with specified promotion and return the new state
```

After advancing a pawn to the other side of the board, the player may promote the piece by entering the promotion coordinates, along with a piece to promote to (e.g. a7 a8 queen). As per normal chess rules, the program tries to disallow promotions to a king. However, the guard has been incorrectly implemented, resulting in a vulnerability. By allowing the player to input a misspelling of "king", the program will accept, and then autocorrect their input to "king" resulting in the time-complexity attack.

Step 2: Determine depth for adversary search

Pseudocode:

```
Boardstate score = |total white piece value – total black piece value|
Depth = 3 - 0.00041*score + 3.9e-8*score^2
```

Chessmaster adds piece values to a total score according the following heuristic:

- Pawn = 100
- Knight = 300
- Bishop = 300
- Rook = 500
- Queen = 900
- King = 20,000

Here, the depth for the adversary search is determined by a quadratic regression. For relatively even boardstates, where score approaches 0 in an even game, the depth is set to a relatively high amount, around 3. In a lopsided game, the depth is set to a lower amount, around 2.

In the case where there are three kings on the board, the score and depth values are set prohibitively high by the evaluate and depth functions, >10,000 and >15 respectively.

Step 3: Negamax search until timeout

Pseudocode:

```
function nextMove(State):
bestScore = −∞
currentScore = −∞
For each possible move from State:
depth = determineDepth
If it is white's turn:
currentScore = -negamax(nextState, depth, 1)
Else if it is black's turn:
currentScore = -negamax(nextState, depth, -1)
If the currentScore is higher than the bestScore:
bestScore = currentScore
bestMove = the current move
return the bestMove
```

```
// When given a sufficiently large depth, the exponential growth of the
recursive
// call will result in a timeout.
function negamax(State, depth, color):

if depth = 0 or the game is over
return color * state.eval()

bestScore = −∞
currentScore = −∞

For each possible move from State
// Recursive call
currentScore = −negamax(nextState, depth − 1, −color)
bestScore = max(bestScore, currentScore)
return bestScore
```

These two functions are called after the player makes a move. When the depth is set abnormally high, as in the case with 3 kings, the recursive negamax function is called a prohibitively large number of times, resulting in a timeout.

4.2.2.2.2 Use Cases

Chessmaster is a client for playing chess against an AI. The only interactions between the user and the program are starting new games, displaying the current game, and making moves, all after first inputting a username and password.

4.2.2.2.3 Overall code structure

Chessmaster classes and methods are depicted in Figure 14.

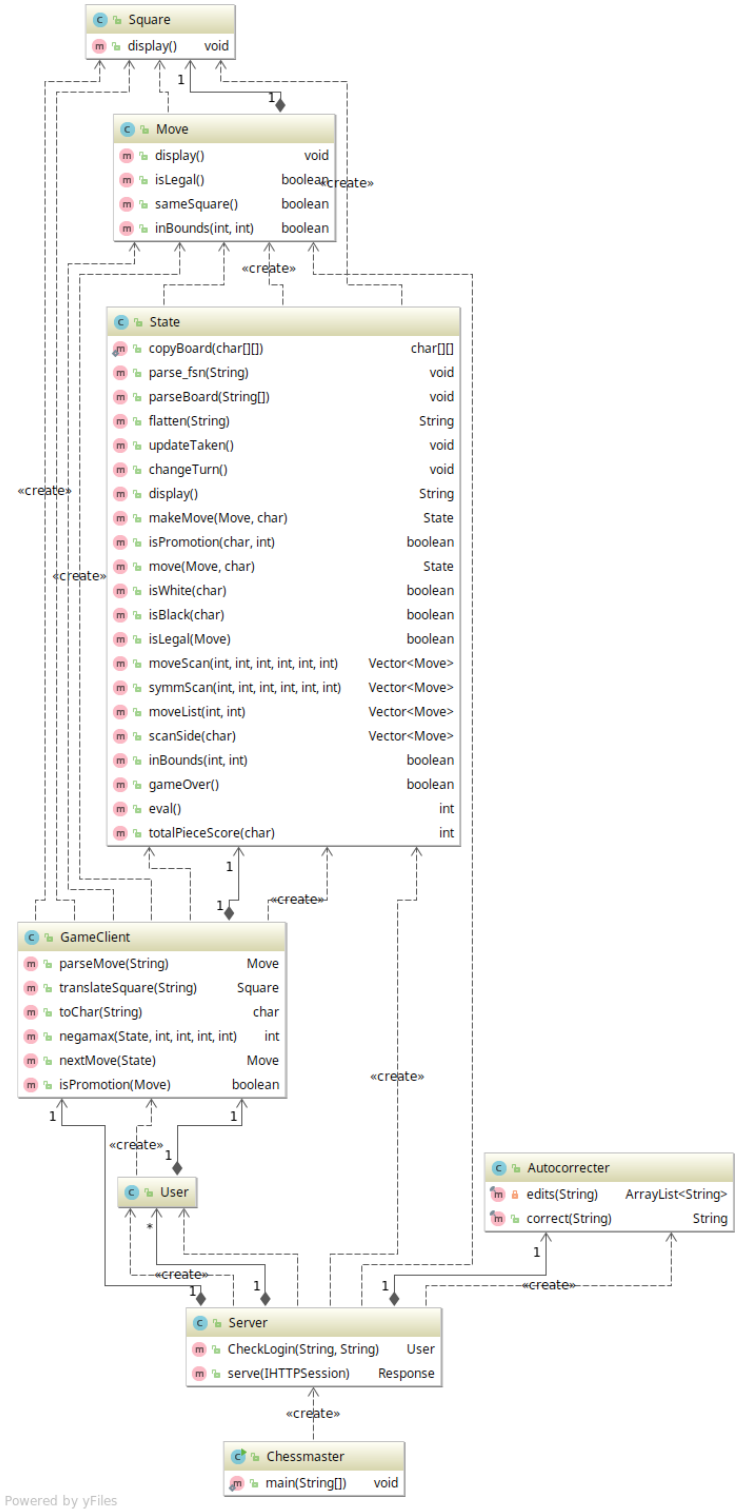


Figure 14. Class Diagram

- Chessmaster: Contains main; responsible for starting HTTP web server.
- Server: Instantiates a NanoHTTPD HTTP server and listens for incoming client requests. Manages user list.
- User: Instantiates game clients for all unique users (i.e. with different names). Contains name and password fields.
- GameClient: Responsible for administrating a game of chess. Translates input to the backend and runs the opponent AI.
- State: Keeps track of any given gamestate.
- Move: Contains information about an individual move.
- Square: Contains information about an individual square.
- Autocorrector: Does basic spelling correcting.

4.2.2.2.4 Inputs and Outputs

There are two types of accepted player input, creating a new game and making a move. Both require inputting a username and a password. If chessmaster does not recognize the username, the server creates a new user with the supplied username and password. If the player is making a pawn promotion, they must enter in the piece they would like to promote to along with the move coordinates. The input format is shown below. Example inputs and responses are also provided:

Template:

```
username
password
move / new game / display
move coordinates (req. if moving)
piece to promote to (req. if making a pawn promoting move)
```

Examples:

```
Input:
Bob
bobspassword
new game
```

Response:

```
New game created
  ABCDEFGH
8 rnbqkbnr
7 pppppppp
6 ........
5 ........
4 ........
3 ........
2 PPPPPPPP
1 RNBQKBNR
  ABCDEFGH
```

Input:

```
Bob
bobspassword
display
```

Response:

```
Displaying current game
  ABCDEFGH
8 rnbqkbnr
7 pppppppp
6 ........
5 ........
4 ........
3 ........
2 PPPPPPPP
1 RNBQKBNR
  ABCDEFGH
```

Input:

```
Bob
bobspassword
move
e2 e4
```

Response:

```
  ABCDEFGH
8 rnbqkbnr
7 pppppppp
6 ........
5 ........
4 ....P...
3 ........
2 PPPP.PPP
1 RNBQKBNR
ABCDEFGH
ABCDEFGH
8 r.bqkbnr
7 pppppppp
6 n.......
5 ........
4 ....P...
3 ........
2 PPPP.PPP
1 RNBQKBNR

ABCDEFGH
```

4.2.3 ClassScheduler

4.2.3.1 Description

The Class Scheduler is a web based non-vulnerable application that creates a school schedule based on the inputs of courses, number of sections for each course, list of teachers, list of rooms, list of students and maximum number of iterations through the algorithm.

It uses a genetic algorithm to search the very large space of possibilities to find a schedule that does not break any of the specified rules. We define one class (a section of a course with a teacher, a day/time schedule, a set of students and a classroom) as a 'gene' and combine each section of each course into a chromosome (combining the genes into one chromosomes) for a complete school schedule, i.e. all courses and their sections have been scheduled. We apply genetic operators (mutation or crossover) to generate new 'better' chromosomes until a correct match is found, i.e., a schedule that satisfies all the rules or the maximum number of iterations has been reached.

This challenge does not contain an intended vulnerability.

There is, however, a red herring in the code but no actual vulnerability. The interaction between the population size and the size of the selection pool (using a tournament style selection process to determine the schedules that will be operated on) can cause the application to run forever. These parameters cannot be modified by the adversary so the vulnerability can't be triggered.

4.2.3.2 Software Design

This program allows a school administrator to generate a schedule for all courses, including all sections of each course, given the teachers, students, time periods and rooms available for the school.

Generating a school schedule, like many scheduling type problems, is a nondeterministic polynomial time (NP) complete problem. The search space can be very large and finding an optimal solution in polynomial time is difficult. Genetic algorithms find approximate solutions where the search is not random but directed by a fitness function.

The class scheduler program uses a genetic algorithm to find the best schedule possible in the time given. The fitness function is a set of rules that are run over each proposed schedule adding a 'cost' to the overall fitness of the schedule when the rule fails. When a schedule is created that has a fitness value of zero success is declared. If after the maximum number of generations (set by the user but capped at 5000) has elapsed and no successful schedule has been found, the schedule with the lowest fitness value is selected as the best possible.

In the ClassScheduler a chromosome (used by the genetic algorithm) represents a complete schedule and is made up of individual genes. Each gene is one section of one course (a class) so one chromosome (schedule) has all course sections included in it. The size of the chromosome is dependent on the number of courses and sections per course. The location in the chromosome determines the course and section, i.e. the first gene is course 0, section 0, the second gene is course 0, section 1, and so on. A gene contains the teacher, room number, days the course meets, the time block for the specific course and a list of students taking this course.

The algorithm creates an initial population of chromosomes, or candidate schedules, where the contents of each gene is generated randomly. Each chromosome has all courses and sections but the remaining information (teacher, students, room, date, time) will be determined randomly from the set of inputs.

Each chromosome has to be evaluated to determine whether it is a schedule that meets all the criteria or something close to that. In terms of genetic algorithms the chromosome is evaluated to determine its 'fitness'. The fitness of each chromosome is computed using a set of rule. Each rule has an associated 'cost' that is added to the chromosomes fitness score when the rule is broken. For example, a rule might be that a teacher cannot teach two classes at the same time on the same day. Breaking this rule would add 250 points to this chromosomes fitness score.

Once the initial population is generated and their fitness scores computed the first generation is created. If there is a chromosome whose fitness score is considered a success (In the class scheduler's case this would be a score of 0 where no rule has been broken) the algorithm stops and a successful schedule is available for the user to download or look at using a web browser. If there are no successful chromosomes another generation of chromosomes is created. The algorithm continues to create new generations of chromosomes (candidate schedules) until a successful chromosome is found or the maximum number of generations has been created. If no successful chromosome is found the chromosome with the lowest fitness score after the maximum number of generations has been created is selected.

For genetic algorithms a new generation is created by performing two operations on a subset of the chromosomes in the population. These two operations are called crossover and mutation. Crossover takes two parent chromosomes and creates two children while mutation takes one parent and creates one child.

The first task in creating the next generation is to select a subset of the chromosomes in the population to operate on. The goal is to pick chromosomes that have very good scores, thereby creating better offspring. Picking the best n chromosomes will give us a good set of parents to work with but has the problem of potentially limiting the area of the search space. There needs to be a certain amount of diversity in the subset such that the algorithm doesn't get stuck in a local minimum. Too much diversity will however waste time in areas of the search space that will never yield a successful schedule. The classScheduler uses the tournament style selection method to create the subset pool. Tournament style selection has two properties, the size of the subset (n) and the size of the selection pool (m) which may sound like the same thing but actually allow us to add diversity into the next generation. The selection pool is created selecting at random m chromosomes from the previous generation. The best score in that pool is moved into the subset. This is done n times so we ultimately have a subset of chromosomes from the previous generation.

The next task in creating a new generation is to perform crossover and mutation on this subset from the previous generation. There are of course tuning parameters used for these two operations. For crossover we have the crossover rate which determines how many of the values in the gene are going to be crossed over. Crossover is performed on all the chromosomes in the subset. For every two chromosomes, two new chromosomes are created. The algorithm operates on every gene in the chromosome individually. It generates n random numbers between 0 & 1 where n is the size of a gene. It then looks at every value in a gene and if the random number generated for that value

position is less than the crossover rate, offspring 1 has the value from parent 1 and offspring 2 has the value from parent 2. If the value is greater than the crossover rate, offspring 1 has the value from parent 2 and offspring 2 has the value from parent 1. This effectively swaps random values from each gene between parent 1 and parent 2 to create offspring 1 and offspring 2.

Mutation has two properties, the mutation selection rate, the number of chromosomes from the subset list to operate on and the mutation rate, the number of genes in the parent to modify (i.e. a mutation rate of .7 will modify 70% of the genes in the chromosome chosen randomly). Once a gene is chosen for mutation a random number of values in that gene are actually modified. The modifications are chosen randomly from the permissible values for that values offset type, i.e. if the teacher offset in the gene is selected for mutation a new value for that field is selected at random from the list of teachers.

Once Crossover and mutation have been performed fitness values for the new offspring are calculated and the new offspring are merged into the existing generation and the top 'population size' chromosomes form the new generation. Note here that size of the population never changes, chromosomes with the highest scores are removed from the list after the new generation is merged in.

This process of creating and evaluating generations continues until a 'successful' chromosome is created or the maximum number of generations allowed is hit. The chromosome, schedule, with the lowest score is set as the final schedule.

As specified above, there are a number of tuning parameters used in the classScheduler. They are set to default values but can be overwritten by a new properties file. There is only one parameter that is exposed to the user, that is, the maximum number of generations. This can be modified using the web browser.

4.2.3.2.1 Use Cases

- Generate and download a School Schedule
- View newly generated schedule per teacher, per course, per room
- Download and upload a school data file

4.2.3.2.2 Overall code structure

The ClassScheduler application is a client server application that uses the Vaadin Framework. VaadinUI is the main point of entry and implements the look and feel of the user interface (UI).

Each scheduling item, teacher, room, course, etc. is represented by a separate class. The main UI interface reads the input data from an extensible markup language (XML) file and creates instances of each scheduling item (teacher, course, room and student). They are maintained in maps with unique identifiers for each instance of the item. The unique identifier is the value stored in the chromosome.

Chromosomes are represented by the Schedule class (Figure 15) which contains, among other items, an array of genes. It is this array that is operated on during crossover and mutation. There is one gene in the array for each possible class (a section of a course). Each gene is an array of

Integers where the offset into the gene defines what the Integer represents. For instance, offset 0 in a gene is the teacher offset. The value at this offset is the unique identifier for a teacher instance.

The class that does the bulk of the work is the population class. There is only one instance and it is responsible for creating the initial set of schedules (chromosomes) and creation of all subsequent generations using the crossover and mutation operations.

The fitness score for each chromosome is generated using a set of rules. Each rule extends the AbstractRule class and implements the doExecute method which evaluates the rule against one chromosome. Each rule has a 'cost' of failure and the cost is added to the overall fitness score.

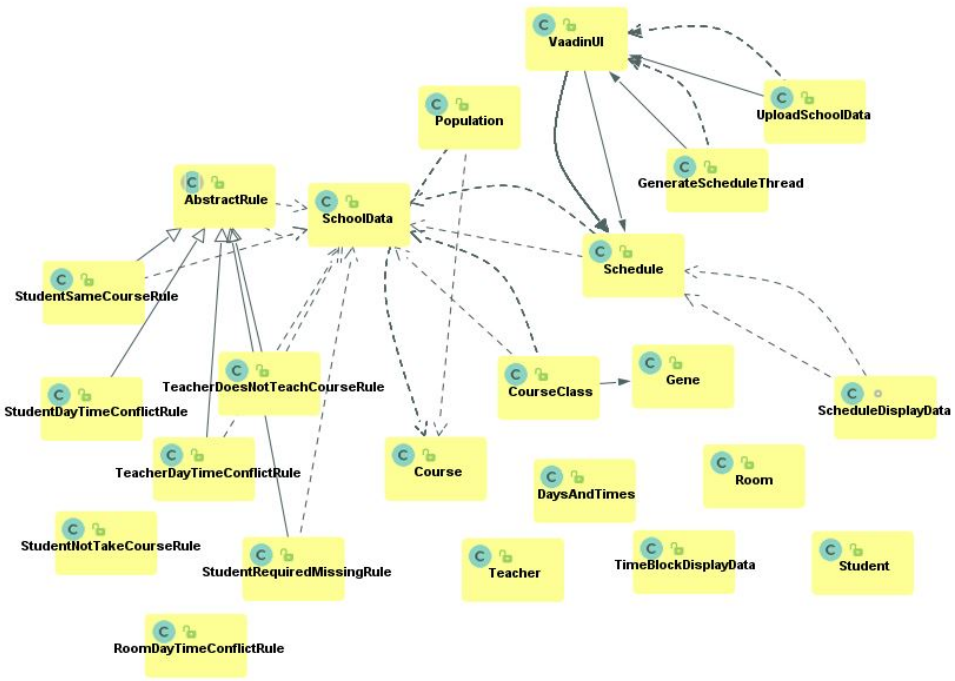


Figure 15. ClassScheduler Class Diagram

4.2.3.2.3 Inputs and Outputs

The application uses a standard HTTP protocol as implemented by the Spring library. However, the challenge uses Vaadin library which at its core is an HTTP Remote Procedure Call (RPC) server that operates on JavaScript Object Notation (JSON) objects. If the user navigates a web-browser to the challenge's address, they will be presented with a simple UI in which the user can press buttons and enter text in a text fiels. Each of these actions will generate a proper JSON data structure which will be wrapped in HTTP protocol and transmitted to the challenge. The challenge then will generate a response JSON and transmit it back over HTTP protocol.

The HTTP entry point into the web interface is http://address:port/ however, most of the data is being sent to http://address:port/vaadinServlet/UIDL/?v-uiId=<N> where <N> is a session number

that starts with a 0 and usually increments for each new session. See BT/sample_input/* scripts for more details on the communication.

The school data is contained in an XML file (schoolData.xml and is included in the jar. The user can download the file from the server using the 'Download School Data File' button on the graphical user interface (GUI), make modifications and then up load the new data using the 'Upload School Data File' button on the GUI. This is the data that forms the basis for the schedule. Every section of every course in the file is part of the chromosome.

The final schedule is created and can be downloaded by clicking the 'Download Generated Schedule' button or viewed in the GUI. The GUI view is done on a per item basis, i.e. the user can view the schedule per teacher, per room or per course. The downloaded schedule is an ascii text file.

The user can also change the maximum number of generations parameter. This parameter governs how many generations of the genetic algorithm can be run. If an optimal schedule is created in fewer generations the algorithm stops at that point. If not it will return the 'best' schedule available after that many generations have been created. This parameter is input using the GUI.

4.2.4 EffectsHero

4.2.4.1 Description

The challenge is a signal processing platform that provides a web interface for connecting various signal processing stages, uploading a sound file, and getting the result of passing the sound file through the different signal processing stages back.

The challenge does not contain an intended vulnerability but multiple red herrings.

4.2.4.2 Software Design

Class VaadinUI – Main point of entry of Vaadin UI library. Implements most of the UI look and feel. The Vaadin is fairly vulnerable library in that is assumes the client-side is not tempered with. This results in several challenge red-herrings. Side-channel red-herring: VaadinUI::runStage() updates a progress-bar on every loop iteration. It's easy to assume that the progress-bar's updated data is sent out immediately to the user and one can monitor the update packets timing and size to determine what stages are being run and how long the sound sample generated. However, Vaadin caches all of these updates and outputs only the final state update once the loop is finished, negating all of the side-channels.

VaadinUI::init() sets up the user interface, but does not set a limit on how many processing blocks are added to the stage. A user can potentially add millions of processing blocks and stall the application. This still will only result in a self-dos. A reasonably set input budget can also prevent this. Each stage takes very little time to run. Most blocks have O(n) complexity with the exception of the Equalizer block, which uses a fourier transform library.

BaseBlock::getLog() returns an accumulated run log from the block. At the end of the stage run, aggregated logs are shown to the user. This potentially presents another side-channel for determining what stages have been run, but the children of BaseBlock have very similar-sized output messages which puts the signal below the noise floor.

BaseBlock::fade() function uses a custom implementation of a Spline calculation, which uses a custom implementation of a Matrix inversion calculation. Over all of the subcalls, fade() has a time complexity in the order of O(n^n) however, fade() only operates on a vector of size 6, which negates the time complexity.

4.2.4.2.1 Use Cases

- - Add processing blocks to the stage
- - Connect processing blocks' outputs to other blocks' inputs
- - Generate and download a sound file that is the result of all of the processing blocks.

4.2.4.2.2 Overall code structure

All of the processing blocks are children of the BaseBlock abstract class which implements a ProcessingBlock interface. The rest – UI setup - is done in VaadinUI class.

4.2.4.2.3 Inputs and Outputs

The application uses a standard HTTP protocol as implemented by the Spring library. However, the challenge uses Vaadin library which at its core is an HTTP RPC server that operates on JSON objects. If the user navigates a web-browser to the challenge's address, they will be presented with a simple UI in which the user can press buttons, drag sliders, choose values in a drop-down lists and enter text in text-fields. Each of these actions will generate a proper JSON data structure which will be wrapped in HTTP protocol and transmitted to the challenge. The challenge then will generate a response JSON and transmit it back over HTTP protocol.

The HTTP entry point into the web interface is http://address:port/ however, most of the data is being sent to http://address:port/vaadinServlet/UIDL/?v-uiId=<N> where <N> is a session number that starts with a 0 and usually increments for each new session. See BT/sample_input/* scripts for more details on the communication.

4.2.5 RailYard

4.2.5.1 Description

This program is a network-based railyard management suite. It allows the operator to manage several rail platforms and attach/detach cars owned by the railway company to the locomotives on the platform via an API. Additionally, operators can view and modify the trains' schedules, inventories, and personnel assignments. Once the operator is satisfied with the trains, they can be scheduled for departure from the platforms, causing them to be written out to a log file '`./out.txt`' before being returned via the API.

This challenge program contains a single vulnerability that can be treated as either **an algorithmic complexity vulnerability in space or in time** (or both) depending on the question(s) asked.

The vulnerability lies in the output file generation code and a purposefully introduced bug. Each train car type has a class with no inheritance or interfaces, and has a reference to the next car in the train. When an input is received with the name of a train car type and a designation, a new instance of the car is created using reflection. Several instances of some types of cars can be added to a train with different, unique designations (e.g., box cars), while other car types (e.g., the coal

car and caboose) can only have one instance added and have no designation. When the train is to be output, the cars are 'linked' together with reflection by updating the value of a field '**`next`**' on each car. Then, starting with the first car and iterating over the train like a linked list, the train is printed to the output log before being returned via the API. The log file increases in size infinitely (space complexity) and the API request never returns (time complexity).

The actual vulnerability is in the class definition of one type of train car. Instead of an instanced reference to the next car in the train, a static reference is defined. When included more than once in a train, this car type's '**`next`**' will create a loop in the output generation process if the last car is an instance of the vulnerable car as shown in Figure 16. This causes the output routine to generate output until the program is terminated. In a case where the vulnerable train car type has not been added to the train previously, the '**`next`**' property of the last car will never be updated and will instead have its original default value of '**`null`**'.

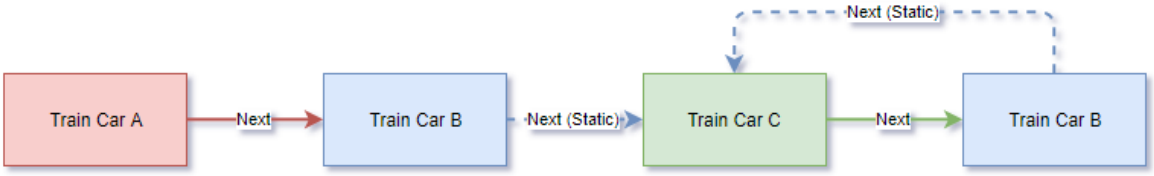


Figure 16. Vulnerable Train; Train Car B (blue) Contains the Vulnerable Static Field

Adding multiple instances of the same vulnerable car type to the train is made more difficult since the program enforces that only one of these cars exists in the railyard. However, a bug exists in this validation function in that the input car name is checked before being sanitized (' ' characters are converted to '_'). Therefore, '**`Coal Car`**' and '**`Coal_Car`**' both pass the validation (they are different strings) but refer to the same train car during the reflective lookup.

4.2.5.2 Software Design

Main Class

The vulnerable class is depicted in Figure 17.

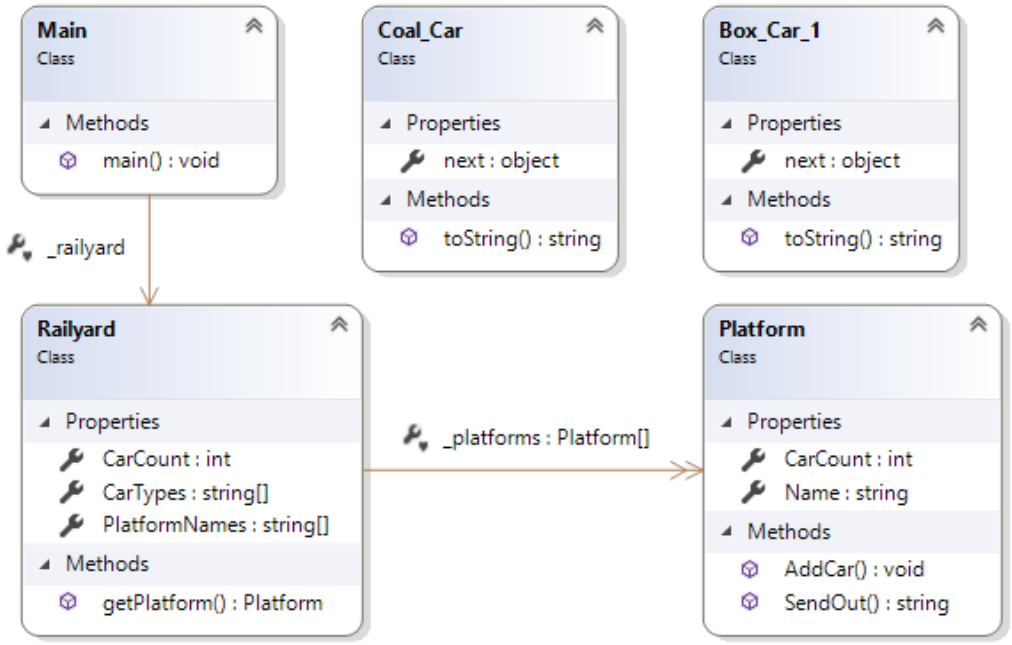


Figure 17. Vulnerable Class Diagram

The `Main` class is the program entry point responsible for handling API requests. The code here is minimal, and most functionality is handed off to individual classes.

Railyard Class

The **Railyard** class is the top-level data representation. It contains static fields **CarTypes** and **PlatformNames** which are used to describe what train car classes are available in the **com.ainfosec.railyard.cars** package (acquired via reflection) and what **Platform** names are available respectively. Aggregate field **CarCount** contains the sum of all **Platform.CarCount**s. No vulnerable code is located here.

Platform Class

The **Platform** class describes a train platform at which a train is being assembled. Multiple named platforms exist within a **Railyard** (e.g., **A**, **B**, **C**, **D**). Contains fields **Name** and **CarCount** which are used to describe the platform name and number of train cars at this platform respectively. Most of the vulnerable code for Railyard Manager exists in this class's **SendOut** and **AddCar** methods.

Box_Car_1 & Coal_Car

These classes describe individual train cars. Many more of these classes can exist, but for brevity only two are shown here. The classes merely need to exist in the **com.ainfosec.railyard.cars** package to be discovered by the **Railyard** class via reflection. The **Platform** class expects these classes to have at least a field **next** of type Object. A small vulnerability exists in the **Coal_Car** class in that the **next** field is implemented as **static**.

Vulnerability code

Step 1: Vulnerable validation code

Pseudocode*:*

```
def add_car(car_type, car_id):
'''
Add a `Car` to the `Platform`

:param car_type: the type of `Car` provided by the user
:param car_id: the `Car` identifier provided by the user

'''


this_car_class_name = None
for car_class_name in car_class_names:
if car_type.lower().replace(' ', '_') == car_class_name:
this_car_class_name = car_class_name
break

if not this_car_class_name:
raise Exception("car type '%s' does not exist" % car_type)

full_name = "%s (%s)" % (car_type.lower(), car_id.lower())

if car_type.lower() in ['coal_car', 'coal car', 'caboose']:
full_name = car_type.lower()

if full_name in cars_at_platform:
raise Exception("%s already in train" % full_name)

cars_at_platform.append(full_name)

try:
new_car = Railyard.car_classes.get(this_car_class_name).new_instance()
cars.append(new_car)
except Exception as ex:
raise Exception("%s could not be created: %s" % (full_name, str(ex)))
```

Logical Steps:

1. When a **Platform** is requested to add a new train car, it is provided with both a car **Type** and an **Identifier**. **Type** and **Identifier** are case insensitive, and in **Type** spaces are equivalent to underscores.
2. If the class **Type** (with spaces converted to underscore) exists in the **Railyard.CarTypes** array:
    a. Save the **Class<?>** from **Railyard.CarTypes** to variable **found**.
    b. Otherwise, abort.
3. If the class **Type** (converted to lowercase) is equal to "**coal_car**", "**coal car**", or "**caboose**":
    a. Variable **fullName** is set to **Type** (converted to lowercase).
    b. Otherwise, **fullName** is set to a string formatted as "**%s (%s)**" with **Type** and **Identifier** (both converted to lowercase).
    c. Essentially, **Coal_Car** and **Caboose** are treated as if they have no **Identifier**, meaning ideally only one coal car and caboose exist at a platform.
4. If **fullName** exists in a list **addedTypes** managed privately by **Platform**:
    a. Abort: Combinations of **Type** and **Identifier** must be unique per platform.
5. Create a new instance of the **Class<?>** found add it to the **stack _cars** managed privately by **Platform**.
6. Increment **Platform.CarCount** and **Railyard.CarCount**.

The vulnerability here lies in that **fullName** is set to the lowercase of **Type**, but spaces are not replaced with underscores (meaning that adding a "**Car Type**" followed by a "**Car_Type**" succeeds when it should fail).

Step 2: Improper class member

```
package com.ainfosec.railyard.cars;
public class Coal_Car {
static Object next = null;
/* ... */
}
```

The vulnerable code here is that the field **next** is defined statically. If our previous section's code worked appropriately, this would not be an issue. However, as more than one **Coal_Car** can be added to a platform, every instance of **Coal_Car** will share the value of **next**.

Step 3: Unintentionally infinitely looping code

Pseudocode:

```
def link_cars():
if len(cars) <= 1:
return

i = 0
while i < (len(cars) - 1):
car = cars[i]
next_car = cars[i + 1]
car_class = car.get_class()
next_field = car_class.get_field('next')
next_field.set_value(car, next_car)
i += 1

def print_to_stream(stream):
stream.write("<^=J")

if not cars:
return

current_car = cars[0]

while current_car:
stream.write(' ')
stream.write(str(current_car))
current_car_class = current_car.get_class()
next_field = current_car_class.get_field('next')
current_car = next_field.get_value(current_car)

def print_cars(file_stream, string_stream):
link_cars()

if file_stream:
print_to_stream(file_stream)

print_to_stream(string_stream)

def clear_cars():
cars.clear()
cars_at_platform.clear()

def send_out(file_stream):
string_stream = StringWriter()

print_cars(file_stream, string_stream)

clear_cars()

return string_stream.buffer
```

When **send_out** is called, iterate over the train cars at the **Platform**, first printing them out to a debug file, then to an in-memory string which is returned. The iterative code does not specify an upper bound limit (which would mitigate the vulnerability).

What is the sequence of adversary actions that will trigger the vulnerability?

To trigger the vulnerability, an adversary must at least:

1. Add a "**Coal Car**" to a platform
2. Add a "**Coal_Car**" to the same platform
3. Trigger the same platform to send out the train

4.2.5.2.1 Use Cases

Railyard is a client for train station management. Interactions include managing station platforms by adding/removing train cars, personnel, stops, cargo, and managing scheduling.

4.2.5.2.2 Overall Code Structure

Figure 18 contains overall challenge class diagram.

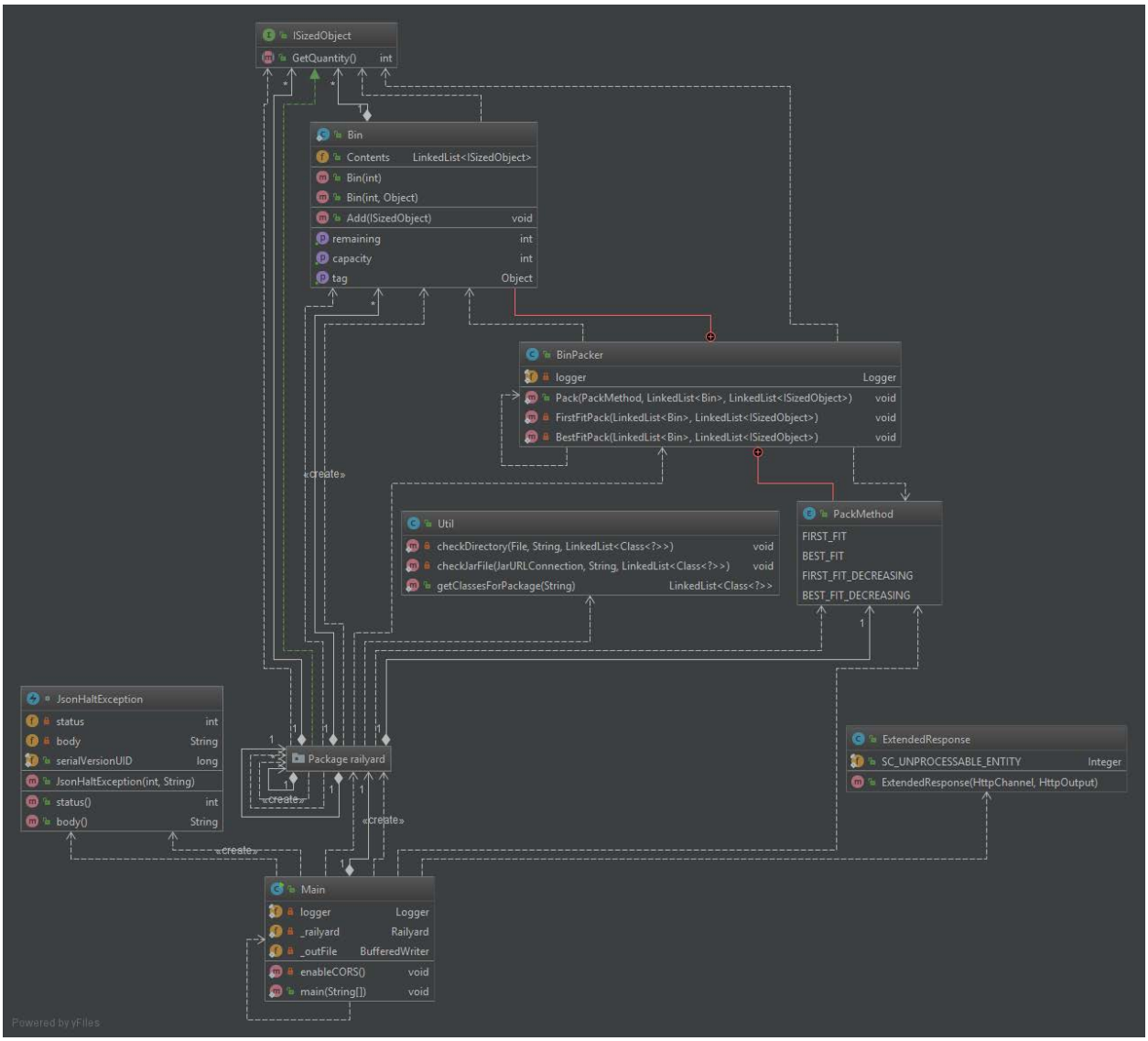


Figure 18 TrainYard class diagram

Main: contains API webserver and entrypoint logic.

- BinPacker: contains helper logic for bin-packing
- Bin: a container class for ISizedObject objects
- ISizedObject: an interface for objects being placed in bins
- Util: various helper functions

- PackMethod: enumeration of different packing methods
- ExtendedResponse: helper class to aid API webserver
- JsonHaltException: exception class to return JSON errors over API webserver

The following core classes are shown in Figure 19.

- Platform: class representing a train platform within the railyard
- Cargo: class for cargo management within a platform
- Materials: enumeration of cargo material types
- CargoItem: class representing a cargo item on a platform
- MethodAddCarsParameter: class used by Google's JSON parse and generator for Java (GSON) add car requests
- Schedule: class for schedule management within a platform
- AddStopParameters: class used by GSON to parse add stop requests
- Personnel: class for personnel management within a platform
- AddPersonnelParameters: class used by GSON to parse add personnel requests
- Railyard: class representing the railyard as a whole
- Various classes representing different types of train cars

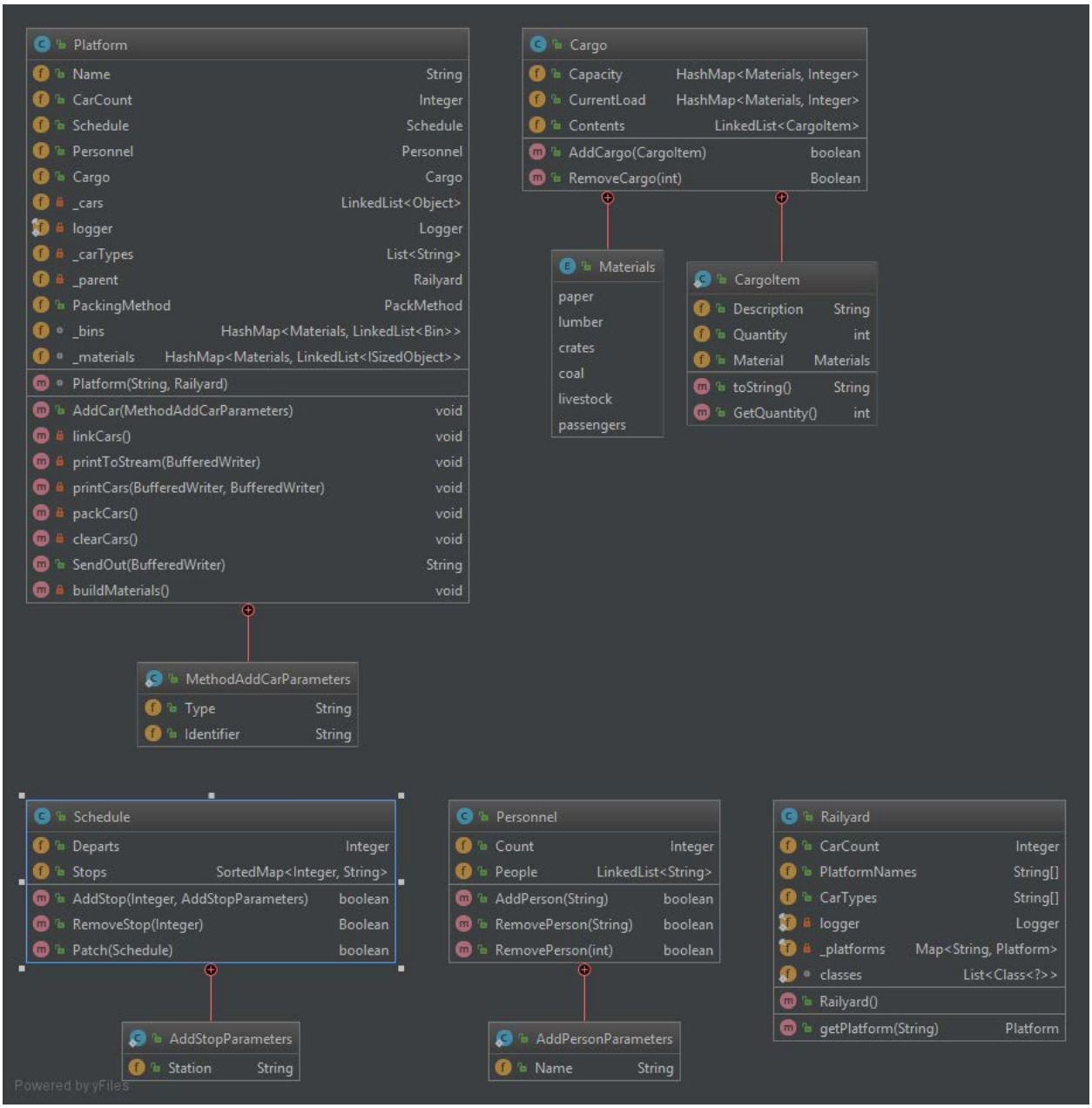


Figure 19 TrainYard core class structure

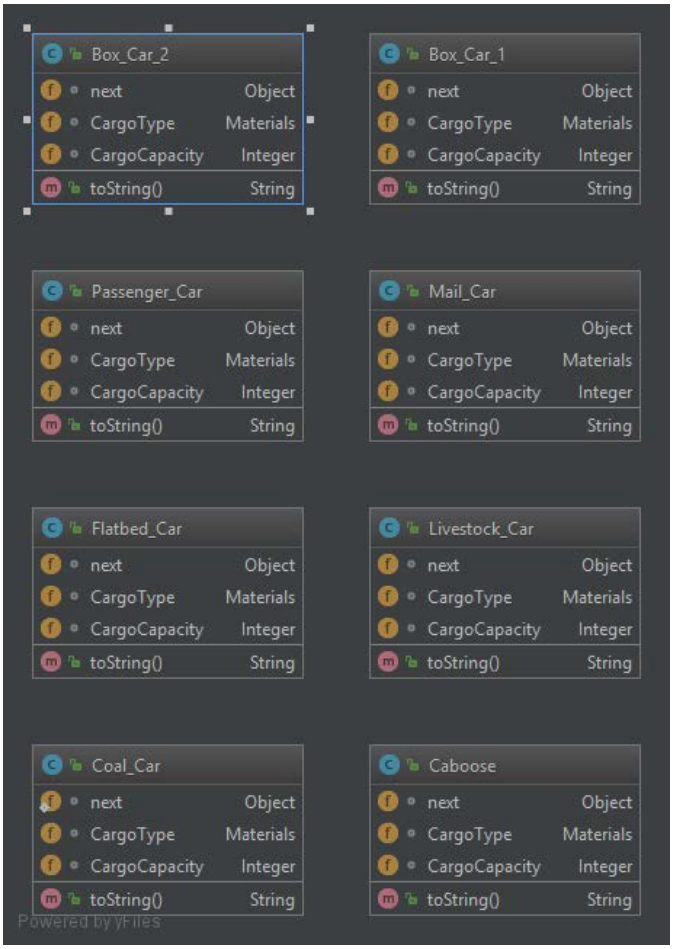


Figure 20 Train car types

4.2.6 STACCoin

4.2.6.1 Description

This purposefully non-vulnerable program acts as both the miner and the wallet for an associated blockchain-backed crypto-currency called STACCoin. The software and data structures of STACCoin is very similar to pre-existing crypto-currencies like Bitcoin [6].

This challenge problem is purposefully non-vulnerable for E6.

However, several algorithms (proof of work, blockchain verification, etc.) give the appearance of being potentially vulnerable to time- or space-complexity exploitation.

Moreover, the wallet functionality of the STACCoin Miner/Wallet application conceals the user's authentication password and private keys. These entities serve as good STAC secrets about which to ask null time- and space-side-channel questions.

STACCoin is a crypto-currency backed by a blockchain. A blockchain is a decentralized append-only data structure that is collectively maintained and updated by a network of applications called

miners. The blockchain consists of a linked sequence of blocks, where a block contains both a data payload and some metadata. To form the links among the blocks, each block has stored in its metadata a cryptographic hash of the preceding block in the sequence.

Every exchange of STACCoin currency is recorded as a ledger-entry transaction in a block on the blockchain. I.e., for one user to transfer some STACCoin to another user, they record the exchange in a transaction and submit it to the STACCoin blockchain miner network. The miners on the network aggregate new transactions into the data payload of a new block, then perform the proof of work necessary to append the block onto the blockchain.

The append-only property of a blockchain means that once a block is added to it, the contents of that block can never subsequently be changed. To ensure that the append-only property holds, the blockchain network will only accept and incorporate a new block if the value of its hash is less than some pre-determined difficulty threshold. To construct a new block that meets this criterion, a meta-data field of the otherwise-complete candidate block called the *nonce* is randomly varied repeatedly. These nonce trials continue until one is found where its insertion into the candidate block's nonce field results in the hash of the whole block being less than the target difficulty threshold. Once a miner finds such a satisfying nonce, it announces the completed block to its network peer(s), which collectively append it onto the end of the blockchain. This nonce hunting process, called *proof of work*, in conjunction with an agreed upon rule that the longest valid blockchain is to be considered the authoritative blockchain, makes it very difficult for anyone to modify the contents of a block once it has been buried by several others.

The STACCoin Miner/Wallet application combines two separate functionalities: Miner and Wallet.

The Miner functionality works together with another Miner on the network to listen for new transactions and to incorporate them into the blockchain by performing the necessary proof-of-work as discussed above. Unlike a traditional blockchain-backed crypto-currency in which the proof-of-work's difficulty varies with the amount of mining power connected to the network, we instead hardcode a fixed difficulty value. In this way, the time it takes to perform proof-of-work on the reference platform will be long, but will always follow a fixed distribution. The proof-of-work algorithm is the slowest part of the program, and thus acts as a red herring. It is not vulnerable, however, because the program's HTTP interface handler and the proof-of-work processor exist in separate threads.

The Wallet functionality allows the user to see his own STACCoin balance, to generate new STACCoin addresses, to import/export his addresses, and to send and receive STACCoins. To send a STACCoin, the owner uses his wallet to digitally sign the transaction with one of his private keys.

4.2.6.2 Software Design

4.2.6.2.1 Use Cases

STACCoin is a crypto-currency that supports two users. A STACCoin user can hold a balance of STACCoins at one or more STACCoin addresses, and they can spend their STACCoins by submitting valid transactions to the STACCoin network.

Each of the two STACCoin users runs his own instance of the STACCoin application. Of the two application instances, each should be configured to use the other as its network peer.

STACCoin largely works like Bitcoin. Each STACCoin block contains one or more transactions, the Secure Hashing Algorithm, 256-Bits (SHA-256) hash digest of the previous block in the chain, and a nonce. To be considered valid, and therefore to be incorporated into the blockchain, a block must satisfy the constraint that the SHA-256 hash digest of its JSON-serialization is prefixed with at least four zeroes. I.e., a block whose JSON-serialization hashes to 00004f0b5449…6af3b7f382cb0 is acceptable, whereas a block whose JSON-serialization hashes to 03c48f0b54d9…60e757f382cb0 is not acceptable. Just like in Bitcoin, the STACCoin application's miner functionality constructs new blocks by repeatedly varying the nonce field until a nonce value is found that causes the block to satisfy the hash digest constraint.

The two peered STACCoin applications work together to maintain the STACCoin blockchain, wherein all past STACCoin transactions are recorded. When one instance of the application receives a request from its user to spend some of his STACCoins, it relays that transaction (if it is valid) on to its peer. Both peers then race to mine a new block using the nonce varying strategy described above. Whichever STACCoin application instance first successfully mines the new block announces its result to its peer, then it appends its newly mined block containing the new transaction onto its copy of the blockchain. The peer receiving the new block announcement verifies whether the new block it received is valid. If the announced block is valid, then the peer also appends it onto its copy of the blockchain and ceases mining; otherwise it discards the announced block and continues trying to mine its own new block containing the transaction.

The STACCoin application always takes the longest valid STACCoin blockchain of which it is presently aware to be the official STACCoin blockchain.

The totally ordered set of transactions stored within the peers' blockchain acts a ledger. By replaying the ledger's transactions in order, the STACCoin application can compute the STACCoin balance associated with each STACCoin address.

A STACCoin address is a Base64-encoded public key of an EC 256 keypair. When a STACCoin address has an associated balance of STACCoins, anyone with that address' corresponding private key can spend those STACCoins (i.e., transfer them to another STACCoin address). To do so, the user wishing to spend some STACCoins submits a new transaction to the STACCoin network.

A STACCoin transaction contains a set of *outputs* and a set of *inputs*. A transaction output identifies an address to which STACCoins are being transferred and the amount of STACCoins being transferred to that address. A transaction input identifies and unlocks the STACCoins being transferred by the transaction. Concretely, a transaction input contains 1) a reference to an output of an earlier transaction, and 2) a digital signature generated using the private key that pairs with the recipient STACCoin address referred to in that earlier output (recall that a STACCoin address is just the public key of a keypair). In other words, that earlier output awarded some STACCoins to a particular STACCoin address (i.e., to a public key), and the new transaction unlocks and transfers those same STACCoins to yet another address by including a digital signature that could only be constructed by the owner of the STACCoin address specified in that earlier output.

All valid STACCoin blockchains share the same first block, called the *genesis block*. The genesis block is hardcoded into the STACCoin application, and any blockchain instantiated by the application will always have it as the first block. The genesis block contains a single transaction that awards Integer.MAX_VALUE STACCoins, which are all the STACCoins that will ever exist, to a particular hardcoded STACCoin address. The private key corresponding to that address is also hardcoded into the wallet functionality of the STACCoin application. This means that all the STACCoins in the universe are immediately accessible to any STACCoin application user upon instantiation of the STACCoin blockchain. In practice, then, before peering with another user, the person instantiating the STACCoin blockchain should first generate one or more new STACCoin addresses, distribute all of the STACCoins among those addresses as the situation warrants, and only then peer with the other STACCoin user's application.

The STACCoin application exposes two collections of RESTful HTTPS API endpoints. The wallet API endpoints only accept password authenticated requests and they expose user-facing functionality for holding and spending STACCoins. The miner API endpoints, which accept unauthenticated requests, are used by the two peered instances of the STACCoin application to maintain the underlying STACCoin blockchain.

The wallet API handles requests for:

- Spending STACCoins from the addresses held by this wallet;
- Getting the balance associated with a specified STACCoin address;
- Getting the cumulative balance associated with all the STACCoin addresses held by this wallet;
- Getting all the STACCoin addresses held by this wallet;
- Exporting the keypairs held by this wallet;
- Importing the keypairs exported from a wallet; and
- Generating a new STACCoin address.

The miner API handles requests for:

- Getting the length of the miner's copy of the STACCoin blockchain;
- Getting a JSON-serialization of the block at a specified depth in the miner's STACCoin blockchain;
- Accepting a new STACCoin transaction; and
- Accepting a new STACCoin block.

4.2.6.2.2 Overall Code Structure

The STACCoin application (Figure 21, Figure 22 and Figure 23) is multi-threaded. One thread uses the Java Spark library to listen over a collection of RESTful HTTPS API endpoints. Each request received by Spark is then handled in its own thread. Aside from the request handling threads, the miner functionality of the STACCoin application also runs within its own thread. When new transactions are received that should be incorporated into the STACCoin blockchain, they are added to a LinkedBlockingQueue that the miner thread polls in an infinite loop.

The external libraries used by STACCoin are com.google.code.gson, com.sparkjava, and org.slf4j. The external libraries are .jar files that get copied to the correct place by the create_cp.sh build script.

StacCoin: Contains main(). Responsible for starting the miner thread and setting up the RESTful API.

BlockChain: Implements a blockchain, consisting of a chain of Blocks where the JSON-serialization of each incorporated block has a SHA-256 hash digest prefixed with at least 4 leading zeroes. Each block also contains the SHA-256 hash digest of the preceding block in the chain.

Block: A block stored on the BlockChain; contains Transactions.

Transaction: A ledger entry reflecting the source, destination, and quantity of a transference of STACCoins.

Input: The source of STACCoins transferred via a Transaction.

Output: The destination of STACCoins transferred via a Transaction.

Ledger: A totally order sequence of Transactions, from which balances associated with specific STACCoin addresses can be computed.

SCHash: De novo implementation of SHA-256.

Miner: Implements the STACCoin miner, which is used to construct new blocks for addition onto the blockchain.

Wallet: Implements functionality for storing STACCoin addresses and keys, and for spending STACCoins held by those addresses.

Expenditure: Deserialization container for wallet spend API requests.

Persistence: Persists application state to disk, and reloads state at startup.

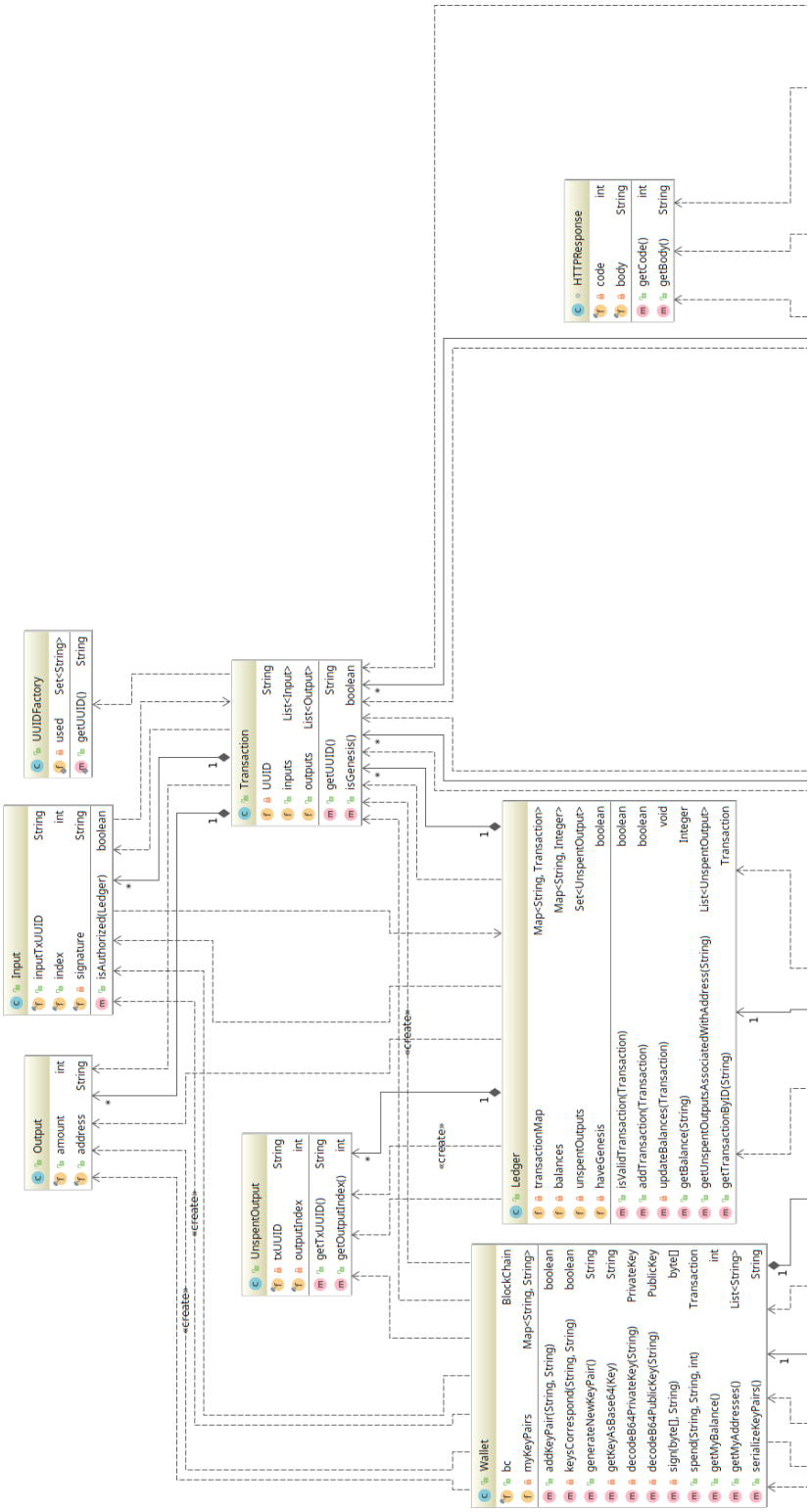


Figure 21. STACCoin Class Diagram

Figure 22. StacCoin class diagram (cont.)

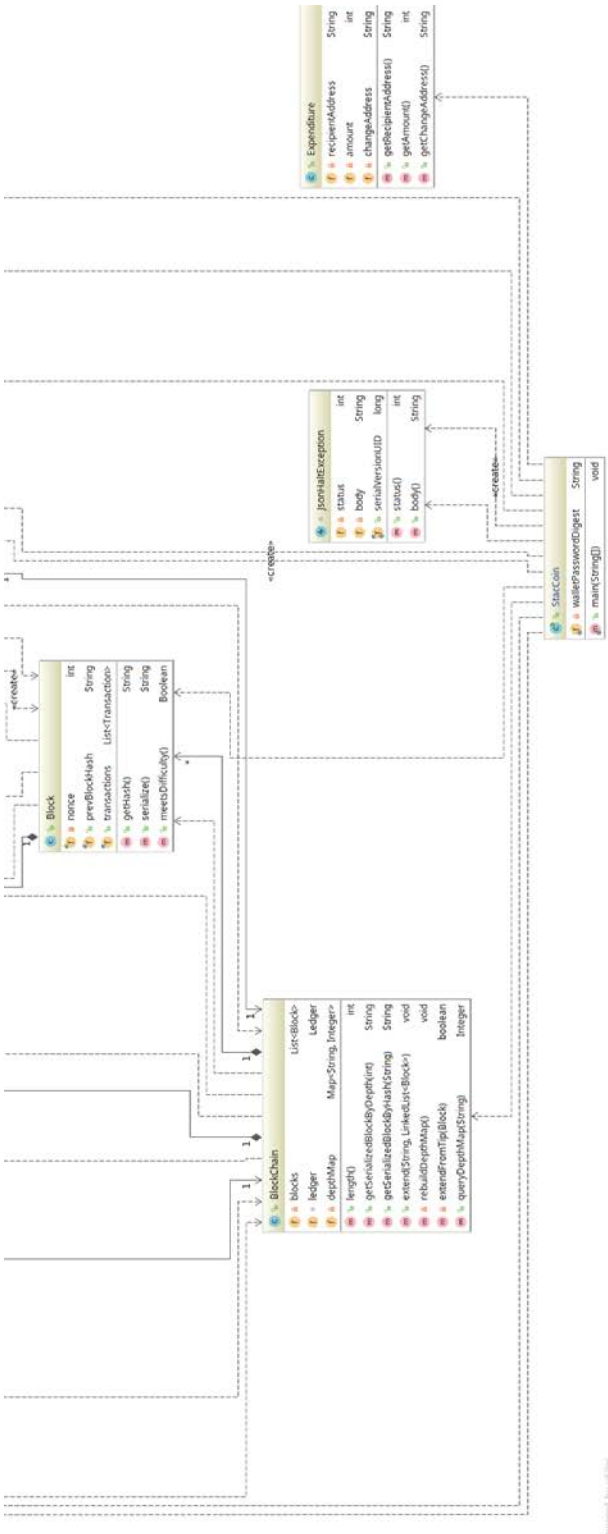


Figure 23. StacCoin class diagram (cont.)

4.2.7 Swappigans

4.2.7.1 Description

Swappigans is a consumer-to-consumer online merchant (e.g. ebay) that accepts bartered trades instead of electronic payment. Users are allowed to register accounts, list items they wish to sell, and purchase items from other users. Unlike a normal merchant though, monetary transactions are not allowed. Instead of paying directly for an item, a user must barter with a subset of their own items to pay for a transaction. The Swappigans web application allows users to: register accounts, post their own items, look through listed items, and initiate a check out request to purchase another user's item.

When initiating a check out request, the Swappigans service will attempt to find a subset of the buyer's items that forms an appropriate trade for the item they wish to purchase, meaning that the total value of this subset is approximately equal to the value of the item the buyer wishes to purchase. Finding said subset is analogous to the non-deterministic polynomial time (NP-complete) subset sum problem [7] and an exact solution would have exponential complexity in the size of the buyer's inventory. Instead of using the exact solution, a Fully Polynomial Time Approximation Scheme is used to find a subset of the buyer's items which is within a chosen epsilon of the purchased item [8]. If such a subset is found, and the purchase is completed then the user is given a receipt for the transaction.

Swappigans contains a Time-complexity attack.

When performing a bartered trade, the Swappigans service finds a subset of the buyer's items that is worth approximately the same amount as the item they wish to purchase. This problem is a manifestation of the subset sum problem and can be shown to be NP-complete; exact solutions are exponential in the number of items the buyer has to barter with.

There is a well-known fully polynomial time approximation algorithm that Swappigans will employ to avoid this complexity. This method is an epsilon approximation that can be shown to be quadratic in the number of items in the user's inventory. While the quadratic term would normally dominate the time complexity, the proper asymptotic for the algorithm is:

$$\frac{n^2 \log(B)}{\varepsilon} \tag{1}$$

Where n is the number of items the buyer has to barter with, and B is the price of the item the buyer wishes to purchase. Epsilon is the approximation parameter, meaning that returned approximate solutions will be within a factor $(1 + \varepsilon)$ of the optimal solution.

In most real world applications of this algorithm, epsilon would be fixed so that approximate solutions were within a small percentage of B. However, Swappigans is dealing with money and users of the service care more about absolute error than percentage error. For example, 5% error on $1000 is less acceptable to users than 5% error on $1. We therefore compute epsilon as a function of the item the user wishes to purchase. Given a maximum item price of M, we compute:

$$\varepsilon = \frac{M}{20\ B} \tag{2}$$

If this value of epsilon exceeds the maximum allowable error (0.15), then it is replaced by that value. Otherwise, it is left in place for the computation. For all benign values of B (i.e. those less than M) it is easy to see that this value of epsilon will be bounded below by 0.05, and above by 0.15. With epsilon and B in the allowed ranges, there does not exist a sequence of inputs within the allowed budget that will result in excessive time utilization.

However, a faulty guard triggered when users add items to Swappigans allows the attacker to exceed that threshold. The guard has a regex that does not properly eliminate Unicode characters from user supplied prices. If a supplied price contains Unicode characters whose least significant byte is in the American Standard Code for Information Interchange (ASCII) range for 0-9, those characters will be included in the price and not checked against the Swappigans maximum item price (M), thereby allowing an attacker to offer for sale items that have arbitrarily high prices.

Given the above equation, it's easy to see that if an adversary inserts an item with price $B \gg M$ then epsilon will tend to zero when purchasing this item. Further, in this case the algorithmic complexity is dominated by the $\log(B) / \varepsilon$ term and the time taken to process a request can seemingly be driven to infinity. Due to the recursive nature of the algorithm and Java's limited stack size though, there is a practical bound on epsilon. If epsilon is too small (much below .01), then for even modest size input baskets a stack overflow error is generated, and the algorithm terminates. The attacker must therefore insert an item that bypasses the guard, and results in an epsilon that is around .01 for the vulnerability to be achieved.

4.2.7.2 Software Design

Dependencies

- NanoHTTPD (External but automatically fulfilled by Maven)
- Google Guava (External but automatically fulfilled by Maven).

Included Data

This problem includes two pre-canned comma separated value (csv) files, one for user data and one for items for sale

Modules and Components

Swappigans has the following key classes. Their interrelationships are illustrated below in Figure 24.

- Swappigans: Contains main(); responsible for starting the HTTP server.
- WebServer: Instantiates a NanoHTTPD HTTP server and listens for incoming client requests.
    - ListItems: Handles item list request
    - Register/Login: Adds users to system or register respectively.
    - AddItem: Adds items to Swappigans (contains input parsing vulnerabilities)

    - ItemPurchase: Finds subset of users items that can be used to buy a good. Calls vulnerable item matcher.
- UserManager: Static class that reads user objects from stored data, and provides methods to add new users at runtime, check credentials etc.
- User: Object that holds user details
- SwappigansItemManager: Static class that reads item objects from stored data, and provides methods to add new items at runtime, remove them, etc.
- SwappigansItemMatcher: Uses approximation algorithm to find subsets of a users items that can be used to purchase another item.

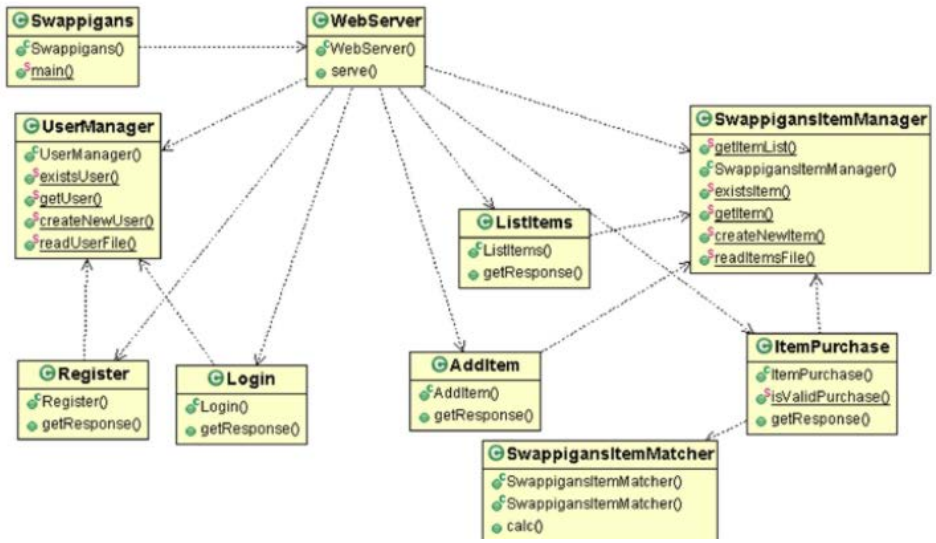


Figure 24. Swappigans Control Flow Diagram

Process Flow

Swappigans services five types of requests: register, login, addItem, listItems, and, purchaseItem.

Inputs and Outputs

Requests are made to Swappigans server using HTTP POST or GET requests. Only the URL path and URL arguments for the request are processed, all body data is ignored. All requests are made to http://server_address:8001/

Login and Registration Screen:

Aliases: http://server_address:8001/, any Swappigans page missing the correct arguments.

Required Arguments: None

Example requests:

- GET / HTTP/1.1

- POST / HTTP/1.1

Explanation: Any GET/POST request without appropriate arguments will display the Swappigans login/registration screen. Requests to the default root will also be redirected here.

Expected Output: An HTML form page that contains a box for username and password, as well as buttons to register or login. The action of the form will redirect the user to either the login or registration pages.

login:

Aliases: http://server_address:8001/login

Required Arguments:

- userName: String (4-16 bytes)
- password: String (6-32 bytes)

Example requests:

- POST /login?userName=<userName>&password=<password>
- GET /login?userName=<userName>&password=<password>

Explanation: The login request checks a user's credentials against the database. If they are correct, the user is redirected to the listItems page with an appropriate sessionKey and userName field. If not, they are redirected back to the login and registration page.

Expected Output: An HTML page redirecting users either to the item list or to the login and registration page.

register:

Aliases: http://server_address:8001/register

Required Arguments:

- userName: String (4-16 bytes)
- password: String (6-32 bytes)

Example requests:

- POST /register?userName=<userName>&password=<password>
- GET /register?userName=<userName>&password=<password>

Explanation: The register request attempts to create a new Swappigans user. It checks to exist the user doesn't already exist and that usernames and passwords meet requirements. If the request is successful, the user is redirected to the listItems page with an appropriate sessionKey and userName for their new username. If not, they are redirected back to the login and registration page.

Expected Output: An HTML page redirecting users either to the item list or to the login and registration page.

listItems:

Aliases: http://server_address:8001/listItems

Required Arguments:

- userName: String (4-16 bytes)
- sessionKey: String (32 bytes)

Example requests:

- POST /listItems?userName=<userName>&sessionKey=<sessionKey> HTTP/1.1
- GET /listItems?userName=<userName>&sessionKey=<sessionKey> HTTP/1.1

Explanation: The listItems request returns an HTML page that lists all items available for purchase, their price, and a hyperlink to purchase the items. There is also a form at the bottom of the page allowing a user to list their own items for sale, which will have form action redirecting to the addItem request.

Expected Output: An HTML page listing items for purchase with an add item box at the bottom of the page.

addItem:

Aliases: http://server_address:8001/addItem

Required Arguments:

- userName: String (4-16 bytes)
- sessionKey: String (32 bytes)
- price: String (represents a double, 5+ bytes).
- itemDescription: String (8-64 bytes)

Example requests:

POST

```
/addItem?userName=<userName>&sessionKey=<sessionKey>&price=<price>&itemDescri
ption=<itemDescription> HTTP/1.1
```

GET

```
/addItem?userName=<userName>&sessionKey=<sessionKey>&price=<price>&itemDescri
ption=<itemDescription> HTTP/1.1
```

Explanation: The addItem request checks the username and session keys to ensure the user is logged in correctly. It then adds the item to the list of items for sale, and tells the user whether or not the addition was successful.

Expected Output: An HTML page that indicates success/failure and also includes a link back to the listItems page.

purchaseItem:

Aliases: http://server_address:8001/purchaseItem

Method: POST, GET

Required Arguments:

- userName: String (4-16 bytes)
- sessionKey: String (32 bytes)
- itemId: String (36 bytes)

Example requests:

```
POST /purchaseItem?userName=<userName>&sessionKey=<sessionKey> HTTP/1.1
GET
/purchaseItem?userName=<userName>&sessionKey=<sessionKey>itemId=<itemId>HTTP/
1.1
```

Explanation: The purchaseItem request checks the username and session keys to ensure the user is logged in correctly, and ensures the user is not trying to purchase their own item. It then tries to find an approximate subset of the buyer's items that can be used to barter for the target item.

Expected Output: An HTML page that indicates the success/failure of the purchase, and also includes a link back to the listItems page.

Vulnerable Algorithm

When a buyer initiates a purchase request on Swappigans, the service attempts to find a subset of the buyer's items that has similar value to the item they wish to purchase. This subset is computed to be within a multiple of (1+epsilon) of item's purchase price using a well-known approximation algorithm to the subset sum problem. The choice of epsilon, and the price of the item purchased drastically affect the performance of the algorithm, and both can be implicitly controlled by an attacker.

By manipulating Swappigans to insert an item with an extremely high price, the attacker can create a degenerate case where the utilization threshold can be exceeded within the input budget. The general sequence of the attack is for the attacker to 1) create a new "seller" user, 2) add an expensive item to Swappigans from the "seller", 3) create a "buyer" user, 4) add many items to Swappigans from "buyer", and 5) attempt to purchase "seller"'s expensive item using "buyer". Successfully completing this attack requires the attacker to bypass a faulty guard in step 2, which leads to a worst case output of a function for step 5. The vulnerable algorithms are described below.

Faulty Guard 1 – Violating maximum item price

When adding an item to the Swappigans item list, the prices are sanitized and checked to make sure they don't violate the maximum item price, $1000.00. This sanitization process is broken and allows users to use Unicode characters to add items that are several orders of magnitude above the allowable price.

The price argument is passed to the sanitizer from the WebServer as a String, S.

The core of the input sanitization procedure is to:

1. Remove all ASCII characters from S other than 0-9, and "."
2. Keep all (including Unicode) characters before the decimal, and only the first two characters after the decimal.
3. Remove all decimals from S.
4. Try and convert S to an Integer P.
5. If P is in the range (MIN_PRICE, MAX_PRICE) return P, else return 0.

If the sanitization process throws an exception (which is only possible if a Unicode character is present) then a second method seemingly filters out all non 0-9 characters from S. The pseudo code for this procedure is to:

1. Convert S to a character array C.
2. Create an empty string F
3. For each character c in C
    - Cast c to a byte, b.
    - If b is in the range [48-57] concatenate it to F
4. Convert F to an Integer P and return P.

In appearance, this code will only allow characters in the integer range [48, 57] which is 0-9 in the ASCII table to be added to the returned String. However, if c is a Unicode character, then its byte representation, b, will only hold the low byte for c. If this value is in the range [48-57], then that value is concatenated onto the returned String. This exception handler will only get called if Unicode is present in the input string, making it unlikely to be discovered by fuzzers. This broken guard allows an attacker to create an item that has an arbitrarily large price (modulo machine limits) instead of the expected bound of $1000.

Faulty Guard 2 – No lower bound on epsilon

Swappigans picks epsilon as a function of the maximum item price M and the purchased item price B. The function that sets epsilon assumes that the value of B is strictly less than M and attempts to pick ε so that it approaches .05 as B approaches M, and approaches .15 as B tends to 0. The computation is performed as follows:

1. Let $\varepsilon = .05 * M / B$
2. If $\varepsilon > .15$ then $\varepsilon = .15$

For all $B < M$, we see that epsilon will be in the range [.05, .15] as expected. However, since the system expects B to never exceed M, there is no safeguard on the minimum value of epsilon, and if an attacker uses the above exploit they can effectively drive epsilon to zero.

4.2.8 TollBooth

4.2.8.1 Description

TollBooth is a peer-to-peer like mesh networking application meant to be run on vehicle toll booths to process automated transponder payments. All packets in the mesh network are transmitted over the reference network in User Datagram Protocol (UDP) packets. The system uses signing to authenticate every packet. Upon start, the first instance of the challenge assumes the role of the root hub that will keep track of the transactions. Each instance will also generate their key pair. Root node will present an interface separate from the mesh network to allow gathering of statistics about toll collections. New instances of the challenge broadcast discovery packets to build a mesh network. When a leaf node is presented with a car transponder's ID, it signs the message containing that ID and sends it to the root instance (first instance started). Upon receiving the message, root instance looks up the type of car by transponder's ID, calculates the charge amount, and responds back to the original leaf node with the confirmation of charge or with transponder not found message to activate license plate reading hardware (hardware not implemented in this challenge).

Tollbooth contains a **Time Complexity Vulnerability**

If a root node receives a properly signed message to decrease the balance associated with a transponder and receives a packet with the same value of the signature but incorrect data, the root node will drop the properly signed packet and never change the balance nor confirm the payment to the leaf node.

4.2.8.2 Software Design

Related Class Roadmap (see Figure 25**)**

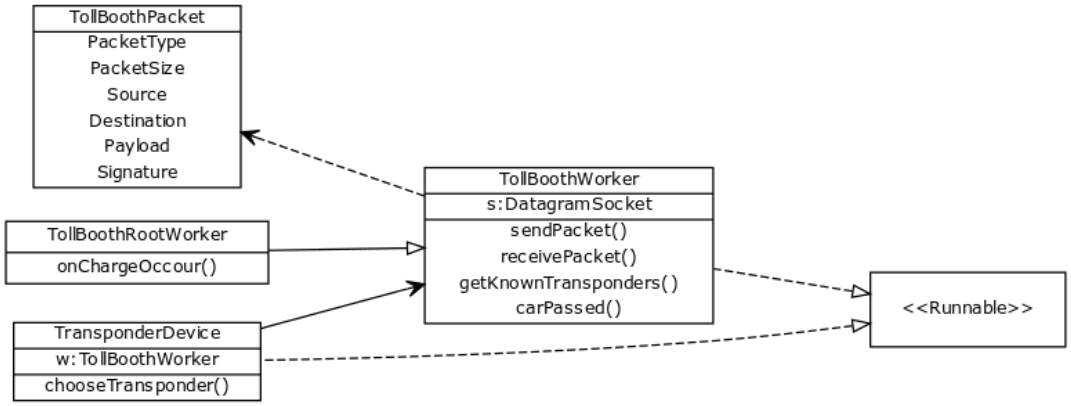


Figure 25. Tollbooth Class Diagram

What is the sequence of adversary actions that will trigger the vulnerability?

1. Listen on the network for a “charge_transponder” packet from leaf to root node.
2. Read the packet from (1.) and craft a packet such that its signature bytes equal signature bytes of the packet in (1.) but data bytes are not equal (the packet structure still has to be correct though).
3. Send crafted packet to the root node
4. Repeat 1-3 while wanting to avoid balance charges. Making sure to minimize time it takes to perform 1-3

4.2.8.2.1 Use Cases

The first instance of the program launched on the network becomes the root manager. Root manager provides a web interface for drivers to register their transponders, check transponder balances, and add value to the transponder balances. All of the customer interactions are done with the root manager. All other instances of the program launched on the network are for designated tollbooths with wireless transponder readers. Once a tollbooth launched, it will query the root manager for existing transponder IDs and wait until such a transponder passes under the transponder reader (simulated). When a car passes, the tollbooth will let the root manager know, and in turn the root manager will bill the transponder for the mount corresponding with their registered car type.

4.2.8.2.2 Overall code structure

Figure 26 depicts the TollBooth code structure.

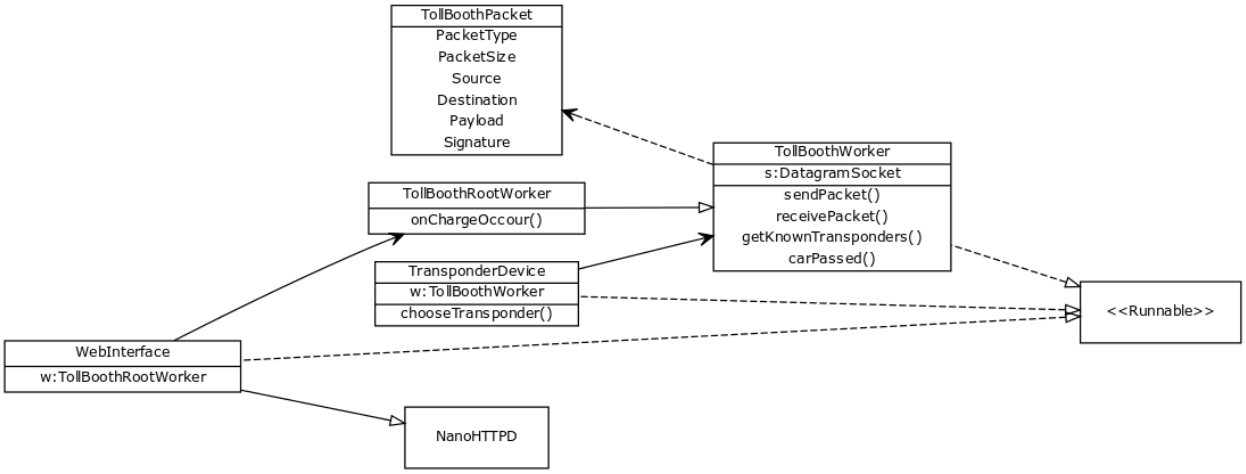


Figure 26. TollBooth Code Structure

4.2.8.2.3 Inputs and Outputs

The user interaction is happening over the HTTP protocol to the following end-points:

- GET Index.html
- GET register.html
- POST register
    - Variables: transponder, password, vehicle, amount
- POST login
    - Variables: transponder, password
- GET info
    - Variables: transponder, token
- POST add_value
    - Variables: transponder, value, token

4.3 Engagement 5 Challenge Programs

4.3.1 IBASys

4.3.1.1 Description

Image Based Authentication System (IBASys) is a network-based authentication server that uses image passcodes in place of textual passwords as authentication tokens. Authentication works by passing the image through a transformation that produces a binary passcode string from the image's contents. The server stores the binary passcode string to which each user's image maps in a database, and when a user attempts to login the server compares the binary string derived from the supplied image against the user's known-correct binary passcode string.

When a user attempts to log in by supplying a username and image token pair, the server replies with a response containing some plaintext metadata, an encrypted session token, and possibly an encrypted error message. The idea is that the session token could be used to interact with other services that rely on IBASys for their authentication needs. The token is encrypted using the user's binary passcode string as the key. This ensures that the user's session token remains secure in transit and even after being transmitted, since it can only be accessed by the user if he possesses his own image passcode. If the login fails due to a credential mismatch, the same type of response is returned but the token field is instead populated with a random value before being encrypted.

This challenge contains two vulnerabilities, **space side channel and space complexity** described below.

4.3.1.1.1 Space Side Channel

When a user requests to login to IBASys, a session object is instantiated for the authentication attempt that only persists for its duration. However, when multiple login attempts are made against the same username at overlapping moments in time, the session for the first incoming login request will be reused for the processing of the subsequent requests. Because the activity scope of any login instance is short, i.e., only so long as is necessary to perform authentication, unintentional reuse of a user-bound authentication instance would be unlikely during normal operation. However, a malicious user can take advantage of this race condition to learn a victim's passcode image.

Normally when a user tries to login, the session verifies 1) that the supplied passcode image data is at least as large as a hardcoded minimum size and 2) that the user's incoming data decrypts correctly. However, if an attacker submits a well-formed request immediately followed by a malicious request, then the piggybacked malicious request will inherit the session belonging to the earlier well-formed request in which those checks were already passed. As such, the attacker's piggybacked malicious request will not be subject to them. The authentication algorithm will instead proceed to directly verify the data supplied by the attacker (without even attempting to decrypt it) against just the corresponding prefix of the user's binary passcode string.

The attacker can therefore submit a sequence of login requests of increasing size, where the first request contains data that only gets checked against the first bit of the victim's binary passcode string, the second requests contains data to be tested against both the first and second bits of the victim's binary passcode string, and so on. When the attacker submits one of these requests that

matches a prefix of the victim's binary passcode string, the size of the packet returned from the server is different than if the data supplied didn't match the prefix. This size difference serves as the attacker's observable that allows him to learn the victim's binary passcode string.

Using this space observable, the attacker to determine each bit of the victim's binary passcode string in isolation of the others. The attacker can then use knowledge of IBASys' implementation to derive a passcode image that transforms to the victim's binary passcode string much more quickly than could be accomplished by brute force. With his passcode image, the attacker can login as the victim.

4.3.1.1.2 Space Complexity

When a user requests to login to IBASys, a session object is instantiated for the authentication attempt. This session object should only persist for the duration of the login attempt. However, when as part of their initial request the user passes in an image that is too small to meet the minimum size requirement enforced by a guard on the IBASys server, the session object is not properly cleaned up and remains active. Because the object is active, additional data passed in by the user with the same session ID as the active object will be processed using this object's state. This allows the user to submit additional requests that refer to the same session object of this image buffer. When this happens, and a malformed input is sent, IBASys catches the error caused by this malformed input and logs the error to disk and the image buffer along with it. The malformed input is a buffer of all zeroes of 10KB in size, it causes an ArrayOutOfBounds Exception.

The result of all this situation is that user is able to prime the buffer with data and then repeatedly send a small input that causes the buffer to be logged to disk. Because the input that causes the error can be very small (10KB), the result is a significant explosion in consumed disk space for a relatively small input, as is discussed in the budget section below. To make the logging of the buffer difficult to detect, the log does not write the buffer directly; instead, it logs a serialized version of the session object which will have a deep copy of the buffer. Each time the buffer is written, it takes up about 6MB on disk for an initial 338KB image due to extra data padding from serializing the error message to XML.

4.3.1.2 Software Design

4.3.1.2.1 Related Class Roadmap

The following is a description of the classes related to the vulnerabilities. The corresponding class diagram is shown in Figure 27.

IBASysServer: The IBASysServer class listens on the server port. This class is responsible for handling all incoming requests and handing them off to the LoginManager.

LoginManager: The LoginManager spawn off worker threads for each incoming authentication request.

LoginPendingQueue: This class manages the BlockingQueue used by LoginManager and its worker threads.

ImagePartMatcher: This class implements the worker thread logic for performing image authentication.

Image Database: This is the database of valid binary passcode strings that are associated with users. It is accessed by the ImageMatcher class. All passcodes are 32 bits in length.

LoginSession: The LoginSession object is used for communicating shared state between all the various classes described above. The LoginSession object holds information about the state of a session, including the username, the state (authenticating-pending, authenticated-success, authenticated-failure), the attempted passcode from the network, and the actual passcode from the database.

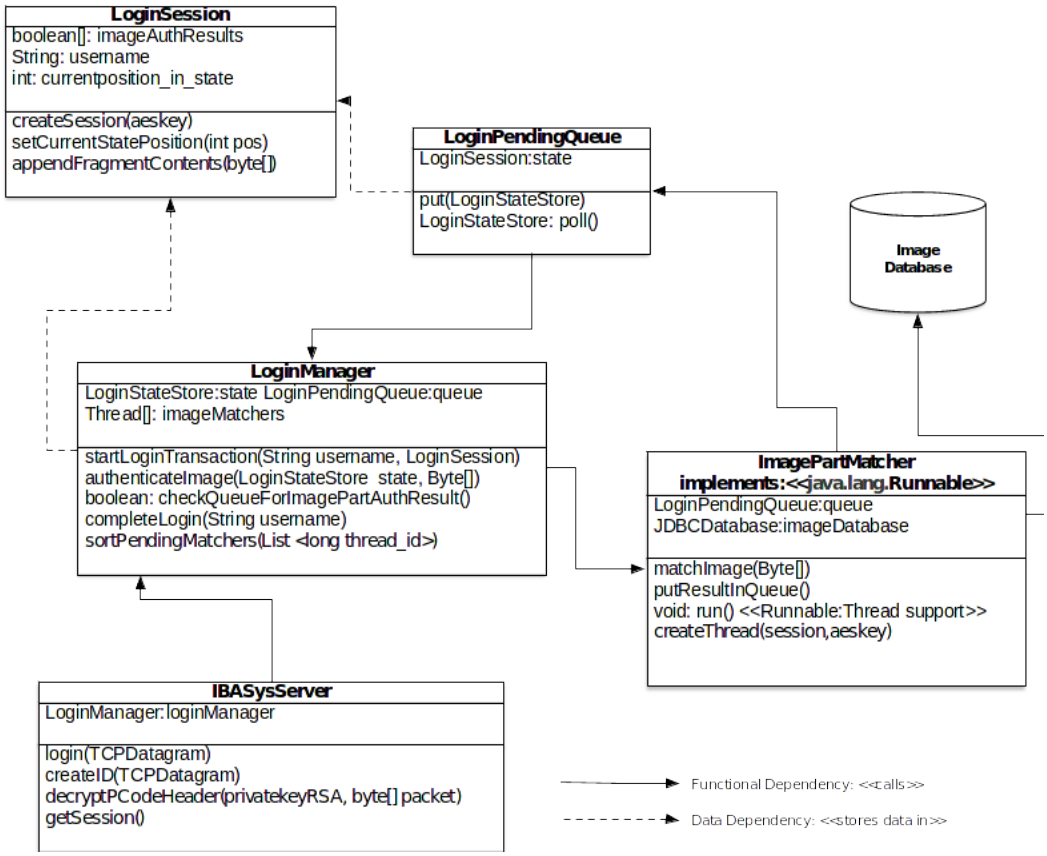


Figure 27. Class Diagram for IBASys Application

Inputs and Outputs

To send a login request to IBASys entails sending it the user's username and image passcode. The minimum size for an image passcode is 340,001 bytes. To perform the login request, the client must split the user's image passcode into chunks of 10,000 bytes and send each chunk in its own packet to the server. The packet for the final chunk, for which the available image data could comprise less than 10,000 bytes, must be padded out with zeros so that the length of the packet's image data payload field is still 10,000 bytes. The server knows that the final packet has arrived if the image data payload for a packet has five straight zeros at the end. If this five-zeros delimiter is

observed at the end of a packet's image data payload, then all the other padding zeros preceding the delimiter are also removed leaving just first byte of the field up through the final non-zero byte in the field. The server then reconstructs the original image by concatenating together all the image data payloads from the various packets it received.

The smallest possible well-formed login request thus entails sending 35 packets to IBASys. The first 34 packets each contain 10,000 bytes out of the 340,000 bytes of the user's image passcode, and the 35th packet is one byte padded with all zeros to make the packet 10k in size.

Passcode/User-Creation Request Packet:

All login request packets are 10,272 bytes in size. Table 8 shows the packet format.

Table 8. Format of Passcode Request Packet

16 bytes: Symmetric AES Session Key (RSA Encrypted with server Public Key)	16 bytes: username (RSA Encrypted with server Public Key)	224 Bytes of padding (anything ok)	16 Bytes: Initialization vector for AES Session Key	10000 bytes: Payload of passcode data. Encrypted with AES Session Key. Note that the delimiter and any padding 0's are not encrypted.

Success/Failure Message for a Well-Formed Login Request:

The server will send back a message in response to a well-formed login request. The response contains an encrypted session token. If the login request was successful, then the token is that of the user; otherwise, the token is random. In either case, the returned token is encrypted. There is also an optional variable length error field, which is never populated in the response to a well-formed request. Table 9 shows the packet format.

Table 9. Message Format for Well-Formed Login Request

6 Bytes: Message type, static string "result"	32 Bytes: token (Encrypted using AES session key provided by client, with image derived passcode as initialization vector).	Variable Length: username plus error message (Encrypted using AES session key provided by client). No error usually occurs for a well-formed login request, and never for one that does not succeed,so this field is 0 bytes in practice.

Success Message for a Malicious (Piggybacked) Request:

When the data in the attacker's malicious login request does not match the victim's binary passcode string, then the attacker receives back this result. Table 10 shows the packet format.

Table 10. Format of Success Message for Malicious Request

6 Bytes: Message type, static string "result"	46 Bytes: Error message (Encrypted using AES session key provided by client, with image derived passcode as initialization vector). An error message with size 46 bytes always occurs in this case.

Failure Message for a Malicious (Piggybacked) Request:

When the data in the attacker's malicious login request does match the victim's binary passcode string, then the attacker receives back this result. Table 11 shows the packet format.

Table 11. Failure Message Format for a Malicious Request

6 Bytes: Message type, static string "result"	32 Bytes: token (Encrypted using AES session key provided by client, with random (fake) passcode as initialization vector).	username plus error message. No error occurs for a well-formed login request that does not succeed, so this field is always 0 bytes plus length of username.

The three different response cases can therefore be differentiated based solely on their sizes. A response to a well-formed request is 38 bytes long plus the username, a failure response to a piggybacked malicious request is 38 bytes long plus the username length, and a success response to a piggybacked malicious request is 52 bytes long. To differentiate between the well-formed response and the 'a failure response to a piggybacked malicious request', supply different length usernames for the well-formed request and the malicious one. In this case, we use user 'davidbowie' for the malicious request, resulting in length 48, and we use 'hansolo' for the well-formed request, resulting in length

4.3.2 Medpedia

4.3.2.1 Description

Medpedia is a medical encyclopedia website where a user may search for and read articles about various medical conditions, medications and procedures. There are a total of 48,165 medical articles from Wikipedia which are served over HTTPS to prevent an attacker from being able to discern which page a user has requested.

Medpedia contains a **Space side channel** where the secret is specific article loaded by the victim's browser.

The set of sizes of objects comprising a particular web page – HTML, stylesheets, images, and scripts – allow an adversary to identify a page with high accuracy. This by itself is a space side channel, but to make the challenge more difficult, we will pretend to mitigate this side channel by adding a random length of padding to each object. However, the amount of padding added will be insufficiently random to successfully eliminate the side channel.

When a request for an HTML page arrives, out HTTP serve creates a SecureRandom instance to add the randomized padding to each object. The same random number generator instance will be used for all the objects associated with a particular page load by storing the SecureRandom instance in a short-lived session. The objects are padded deterministically in the order they appear in the HTML page.

A SecureRandom instance requires an initial high-entropy seed in order to generate a genuinely unpredictable sequence. It is during the seeding process that we will introduce a vulnerability. We

include a custom pseudo-random number generator (PRNG) that mixes high-entropy input from /dev/urandom using a complicated-looking algorithm, but actually only generates eight possible output values. This algorithm is a combination of bit operations, table lookups, random-looking constants, and matrix multiplications that we generate programmatically.

The end result is that for each possible secret symbol (article URL), there are eight possible observable symbols (sets of object sizes) formed by the sum of original object sizes plus one of only eight possible pseudorandom sequences of padding lengths.

We will include a Java security provider in which to hide our broken PRNG, implementing just enough of the algorithms to use it for the web server's HTTP operations. We register out vulnerable PRNG under the name of a legitimate one, SHA1PRNG, the same as in the Open Java Development Kit (OpenJDK) crypto provider.

4.3.2.2 Software Design

Medpedia's third-party dependencies are listed in Table 12.

Table 12. Medpedia Third-Party Dependencies

Dependency	Internal/external?	URL
Spring Framework	External	https://projects.spring.io/spring-framework/
Spring Boot	External	http://projects.spring.io/spring-boot/
zimreader-java	Internal	https://github.com/wikimedia/openzim
ZIMreader (native code)	External	https://github.com/wikimedia/openzim
MapDB	External	http://www.mapdb.org/

The above table does not include transitive dependencies; it can be assumed that they will be external to the challenge program. Note that ZIMReader, the library used for reading the Zeno IMproved (ZIM) file, contains both Java and native code.

ZIM File

Medpedia's master data source is a ZIM web archive file. This file format was designed for archiving large number of web pages, especially Wikipedia or a subset thereof, for offline access. It includes both HTML pages and embedded resources such as images, stylesheets and scripts. We provide a ZIM archive with the challenge application that contains 48,165 articles on medical topics.

Resources in a ZIM file are grouped into a number of single-character *namespaces*, namely:

- -/ (hyphen): CSS, JavaScript, and images not related to individual articles (e.g. a Wikipedia icon embedded in every page)
- A/ : article .html files
- I/ : images and similar files embedded in the .html pages
- M/ : additional ZIM metadata (unused by Medpedia)

Resources are expected to be found at URL <namespace>/<url> relative to some common root: e.g., an article *A/Foo.html* will refer to its embedded image file as ../I/<image.ext>.

Database

Medpedia also maintains a MapDB key-value database as an index and cache for some of the information in the ZIM file. This database contains the following collections:

- A MapDB *BTreeMap* that maps article titles to their URLs (namespace included). *BTreeMap* provides prefix lookups, so this data structure can be used for autocomplete. This data structure is built at startup the first time the ZIM file is read.
- A MapDB *HTreeMap* of article URLs to the content of master pages and sections of pages. A master page will have every top-level section (i.e., an *<h1> or <h2>* element and all content up until the next heading element of the same level, or the end of the parent element) replaced by an HTML import (*<link rel="import" href="…")* of an HTML fragment containing the original content of the replaced section. This map also contains the extracted fragments.

Unlike HTML pages, other embedded, static resources (stylesheets, images, and JavaScript files in the ZIM -/ and I/ namespaces) do not need to be modified so they are served directly from the original ZIM archive.

Components

The Medpedia application consists of the following major components, also depicted in Figure 28:

- **ContentController**: A Spring controller that serves all article content (both HTML and embedded resources). Uses *ArticleService* to obtain article content and uses the vulnerable *SecureRandomSpi* to choose padding sizes for served objects.
- **StaticResourceController**: A Spring controller that serves unmodified resources such as stylesheets, images, and JavaScript straight from the ZIM file via *ZimReader*.
- **TitleSearchController**: A Spring controller that implements the */titles* endpoint and provides a list of article titles matching a specified prefix. It will also pad these responses to avoid a second space side channel (see section “Autocomplete”).
- **TitleIndex**: Wrapper around the MapDB *BTreeMap* of article titles to their URLs. This component is also responsible for initializing the map from the ZIM file at first startup.
- **ArticleService**: Convenience wrapper around *ArticleCache*. When *ContentController* requests a given article, this service first tries to retrieve the corresponding master page from *ArticleCache*. If it is not there, the service loads the original HTML page from *ZIMReader*, stores the resulting page, and then returns the originally requested master page.
- **ArticleCache**: Wrapper around the MapDB *HTreeMap* of master pages and section fragments.
- **ZIMReader**: Modified *zimreader-java* source code. Provides methods to:

- Iterate over all resources (articles and embedded resources).
- Retrieve resource contents by title or URL.
- **Custom crypto provider suite**: A Java crypto provider implementing just enough algorithms to support a single HTTPS cipher suite. Most of these algorithms are simply copied from existing crypto providers. See section “Crypto provider” under “Red herrings” for more details.
- **SecureRandom**: A custom SecureRandom instance containing a vulnerability. See section “Vulnerability Design” for more details.
- **BitArray, BitMatrix, BitVector**: Classes used in random number generation.
- **SeedGenerator, NativeSeedGenerator**: Copied from OpenJDK.
- **Stac, StacEntries**: Classes required to implement Java crypto provider.

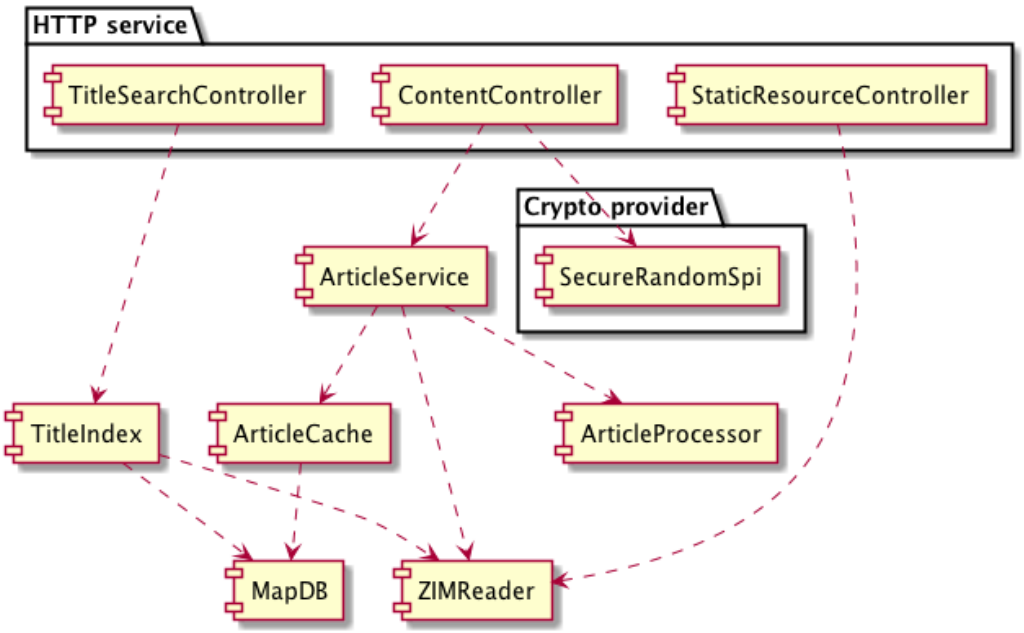


Figure 28. Components and Their Dependencies

4.3.2.2.1 Inputs and Outputs

4.3.2.2.2 HTTP endpoints

The server exposes the HTTP endpoints listed in Table 13.

Table 13. HTTP Endpoints Exposed by Server

Method	Path	Description
GET	*/titles?prefix={prefix}*	Return a JSON list of all article titles matching the specified prefix.
GET	*/A/{article}.html*	Return the master HTML page for an article.
GET	*/{namespace:[-I]}/***	Return static resources.

4.3.2.2.3 JSON messages

The */titles* query endpoint returns a list of messages with the format:

```
[
{
"title": "...",
"url": "...",
},
...
]
```

4.3.2.2.4 Vulnerability Design

4.3.2.2.5 Algorithm

ContentController

ContentController has a private method getMasterPage(HttpSession session) with the following logic:

- Invalidate the current HttpSession if any and create a new session for the page request.
- Get the vulnerable *SecureRandom* instance:
- *SecureRandom random = (SecureRandom) session.getAttribute("random");*
- If *random* is null, set it to our instance of a PRNG.
- *random = srInstance;* where

  srInstance = SecureRandom.getInstance("SHA1PRNG", "STAC");
- Read 20 bytes (160 bits) from /dev/urandom and feed it to random's setSeed() method. See the description of our SecureRandomSpi below for how this triggers a vulnerability.
- Add it to the session: *session.setAttribute("random", random);*

- Call *random.nextInt(n)* to obtain a random integer amount of padding between some MIN_PADDING_BYTES and MAX_PADDING_BYTES.
- Add an "X-Padding" HTTP header to *HttpResponse* consisting of any arbitrary string of characters of the chosen "random" length.
- Call *ArticleService.writeMasterPage(articleName)* to write the master page content to the HTTP response. This call either retrieves the existing content from *ArticleCache*, or *ArticleCache* will load and return the page from the ZIM file if it does not already have the article.

4.3.2.2.6 SecureRandomSpi

The *SecureRandomSpi* in Medpedia's custom crypto provider is a modified copy of OpenJDK's implementation. OpenJDK's implementation iteratively applies Secure Hash Algorithm 1 (SHA-1) to a 160-bit internal state initialized from a random seed.

We want our *SecureRandomSpi* to be secure when used for HTTPS, but insecure when used for object padding in *ContentController*. The key to this is that the HTTPS implementation will let *SecureRandom* seed itself while *ContentController* will manually feed it a seed read from */dev/urandom*. Self-seeding happens the first time our *SecureRandomSpi*'s *engineNextBytes()* method is called and finds its *state* is null. We implement out vulnerability in *engineSetSeed()*, which is not called unless the wrapping *SecureRandom's setSeed()* method is called.

OpenJDK's existing *engineSetSeed()* method operates as follows:

- If the PRNG's internal state is null – i.e., if this PRNG instance has never been seeded yet – then set the internal state to the SHA-1 digest of the provided seed argument.
- If the state is non-null, then "mix" the new seed material into the existing state by updating the state to the SHA-1 digest of the existing state concatenated with the provided seed argument.

When Medpedia's crypto provider is instantiated, a static 160x160 bit matrix of rank 3 called A is generated. Medpedia then repeatedly generates properly random seeds from the system entropy pool and computes a 20-byte SHA-1 hash of those seeds. These hashes are converted into 160-bit vectors and multiplied by the bit matrix A. Since A is rank 3, we know the image of y(x) = A*x has dimension 3. This means that A*x for any x can be written as c1*e1+c2*e2+c3*e3, for some fixed basis vectors (e1, e2, e3) and some constants c1, c2, and c3. Since the basis vectors are fixed and (c1, c2, c3) are all in GF(2) where there are only two possible values (i.e. 0 or 1) there are 2^3 = 8 possible choices of (c1, c2, c3). Thus, any seed generated by the system entropy pool will be transformed to one of these 8 vectors when multiplied by A. This causes *ContentController* to pad an article's object sequence with one of only 8 possible padding sequences.

4.3.2.2.7 Red herrings

4.3.2.2.8 Autocomplete

Medpedia's autocomplete functionality guards against a second side channel by properly supplying random padding to a response. Since the mapping from number of characters in a search prefix to response size might have relatively few collisions for certain search queries, a second

side channel could result when an adversary views a benign user making certain requests. TitlePrefixController makes this side channel too noisy by limiting the number of search results to a maximum of 10 for all queries and adding random padding. To make analysis more interesting, TitlePrefixController uses the same provider for pseudo-random number generation but, critically, does not trigger the vulnerable random number generation. Instead, it creates its own instance of the provider and does not forcibly seed it. This means that it uses a much stronger random amount of padding than the version used by ContentController.

4.3.2.2.9 Crypto provider

To avoid drawing attention to the vulnerable *SecureRandomSpi,* we include it as a small but complete Java cryptography provider – one that implements enough algorithms to implement at least one widely supported Transport Layer Security ciphersuite. (It should be supported by at least one standard browser, by whatever tools the proof script uses, and preferably by *curl*). The web server is then configured to support only that ciphersuite, and use the algorithm implementations from our custom provider.

4.3.3 Poker

4.3.3.1 Description

This challenge is a Texas Hold ’Em poker server that hosts online poker games. Users connect to the server to play heads up (i.e., one-on-one) Texas Hold ‘Em poker against one another. The poker server exposes an API that the provided player client uses to start a new poker table and to play hands of poker against another user. The server handles the shuffling and dealing of the cards, and the evaluation of poker hands. The server is also responsible for repeatedly rendering each player’s view of the table as a Portable Network Graphics (PNG) image, which is then transmitted to the player’s client for display.

The challenge contains a **Time side-channel**. A user connected to the server can send a command at any time to change the current card skin. The selected skin determines how the cards look when they are rendered. When the server is initially started, it pre-renders and caches all the cards using the default skin. However, the cache is cleared whenever a player sends a request to change the current skin. After such a request, subsequently dealt cards are rendered under the new skin and cached on a just-in-time basis.

When a card is dealt for the first time after the skin selection has been changed by a player, it must be re-rendered under the new skin before it can be drawn onto the table. Conversely, if that same card is dealt once again under that same skin selection, the previously rendered representation of the card from the cache will be used. These two modes of operation exhibit an observable timing difference: A card which has never been dealt previously under the selected skin will take longer to be dealt than will a card that has been dealt during a previous hand.

To take advantage of this vulnerability, the attacker would start a new poker table and change the skin to clear the rendering cache. He would then populate the table with two phony players that he controls, and play poker against himself until exactly 26 unique cards (half of the standard playing deck) have been dealt. Let this set of 26 previously dealt cards be called *B* and the complement of the set containing the deck’s other 26 cards be called *C*.

Now, the attacker disconnects one of the clients under his control and waits for a victim player to join. Let the number of operations performed by the attacker up to this point be *N*.

When the victim connects, the server starts the first hand of the game by dealing two hole cards to the attacker and two hole cards to the victim. By observing how long it takes for the table to be rendered as each of the hole cards are dealt, the attacker can ascertain which set, *B* or *C*, respectively contains each of the victim's two hole cards. Let the attacker's two known hole cards be called *x* and *y*, and the victim's two unknown hole cards be called *j* and *k*.

Recap:

- *B:* The set containing the 26 cards that the attacker knows were previously dealt.
- *C:* The set containing the other 26 cards that the attacker knows were not previously dealt.
- *x* and *y*: The attacker's two hole cards.
- *j* and *k*: The victim's two hole cards.

Facts:

- *x*, *y*, *j*, and *k* are all different cards.
- *B* and *C* are disjoint.
- Either (*x*,*y* ∈ *B*), (*x*,*y* ∈ *C*), *x* ∈ *B* and *y* ∈ *C*, or *x* ∈ *C* and *y* ∈ *B*.
- Either (j,*k* ∈ *B*), (j,*k* ∈ *C*), j ∈ *B* and k ∈ *C*, or j ∈ *C* and k ∈ *B*.
- The attacker knows from direct observation which set, *B* or *C*, that each of his two hole cards, x and y, respectively belongs to.
- The attacker knows from the time side-channel which set, *B* or *C*, that each of the opponent's two hole cards, *j* and *k*, respectively belongs to.

Table 14 shows the number of possibilities for the victim's two hole cards under various scenarios, where the victim's hole cards are treated as an unordered pair of differing cards. For example, the upper-left data cell says that if the attacker's two hole cards are among the 26 cards of *B* and the victim's two hole cards are also among the 26 cards of *B*, then there are only possible combinations for the victim's two cards, since they must be among *B*'s 26 cards but neither of them can be the same as either of the attacker's two cards. The worst cases have 625 possibilities for the victim's hole cards.

Table 14. Number of Possibilities for j and k

	x,y* ∈ *B	***x,y* ∈ *C***	***x* ∈ *B* and *y* ∈ *C***	***x* ∈ *C* and *y* ∈ *B***
j,k* ∈ *B	= 276	= 325	= 300	= 300
j,k* ∈ *C	= 325	= 276	= 300	= 300
j* ∈ *B*, *k* ∈ *C	24 * 26 = 624	26 * 24 = 624	25 * 25 = 625	25 * 25 = 625
***j* ∈ C, *k* ∈ B**	26 * 24 = 624	24 * 26 = 624	25 * 25 = 625	25 * 25 = 625

If the attacker had no information other than knowing his own two hole cards, he could be assured of guessing the victim's two hole cards correctly using no more than = 1225 guesses (i.e., oracle queries). But with his added information, the attacker can guess the victim's two hole cards with 100% probability using no more than $N + 625$ operations (where N was the number of operations to set up the attack). This amount of information gain would provide a major advantage to a poker player, since taken together with the shared cards on the table it would allow the attacker to rule out many potential poker hands that the victim might otherwise possess.

4.3.3.2 Software Design

This program can run in one of two modes, client or server. The client mode is graphical and requires a running X server. The server mode is a command-line application suitable for running on a STAC reference platform NUC. The vulnerability exists solely in the server mode of the program, and the client mode is only provided to the blue teams to show them how interactions with the server should work.

The following is a description of the classes that are utilized by the server mode, and a corresponding class diagram is shown in Figure 29.

STACPoker: Contains main(). Depending the command line arguments, starts the program in either client or server mode.

PokerServer: Listens for incoming connections from user clients, forking off a new PokerServerCommsThread as needed to handle communication with each new client connection. It also handles requests incoming from connected clients. The main logic for the progression of a poker game lives here.

PokerServerCommsThread: Maintains a bidirectional and persistent transmission control protocol (TCP) sessions with a single poker client. Passes requests received from its client to the synchronized handle() method of PokerServer.

HandOfPoker: Represents a hand of poker, i.e., one round of poker with two players both receiving cards and playing against one another.

Deck: Represents a deck of Cards. Provides functionality for dealing and shuffling.

Card: Represents a playing card that may be dealt during a hand of poker. The Card class contains members for the playing card's suit and value, and a getter for the graphical representation of the card.

Renderer: Responsible for repeatedly rendering the poker table as a PNG image for each Player. Part of this rendering functionality includes the ability to skin a card. The imagery for a card is stored as an overlay with a transparent background. To skin a card, the Renderer draws the card's overlay over the skin image. All card overlays and skins are provided as resources in the challenge program's .jar package. When a card needs to be rendered, the Renderer first checks an in-memory cache to see if the rendering has already been generated with respect to the currently selected skin. If so, the pre-existing rendering for the card is used. Otherwise, the Renderer needs to reskin the card under the new skin. Testing showed that reskinning just the required card did not provide a large enough timing discrepancy compared to the cache-lookup to outweigh network and scheduling noise, hence an extra delay was added by making the Renderer reskin the entire deck

whenever an individual card needs to be reskinned. However, even though the entire deck goes through the reskinning process, only the requested card is cached.

TablePosition: Stores information about where cards should be drawn on the table imagery.

Poker{Input,Output}Stream: Provide the ability to receive and send long Strings over a Socket.

MessageHelper: Helper class full of static methods that construct and send various types of messages to a specified client.

ServerMessage: An internal representation of a message from the server that is to be delivered to a client. Before sending a message, the server constructs a ServerMessage. The ServerMessage ensures that the message conforms to the poker protocol (cf. Inputs and Outputs section), and provides the serialization routine to transform the message into a JSON String that is suitable for transmission over the connection socket.

ClientMessage: An internal representation of a message from a client that was received at the server. When data is received from the socket, it is deserialized from JSON into a ClientMessage. ClientMessage provides the routine to ensure that the message conforms to the poker protocol. If it the message conforms to the protocol, it is handled by the server; otherwise, the message is dropped.

FiveCardHand: Represents a five-card poker hand held by a player during a hand of poker. Upon construction, the FiveCardHand determines what kind of hand it is; e.g., a “Full House” or a “Pair”.

Evaluator: If a hand of poker is played to completion (i.e., neither player folded), then the Evaluator is used to determine if the hand was a tie or if there was a winner. If there was a winner, the Evaluator also determines which player won. To determine the result of the hand of poker, for each player the Evaluator constructs a FiveCardHand for each of the five card combinations that can be made from the player’s seven available cards. It then finds and compares the highest ranked FiveCardHand held by each player. The result of the hand of poker is returned as a HandResult object.

HandResult: Stores the result of the hand of poker.

HandRanking: Stores information about a player’s FiveCardHand, including what it is (e.g., a two pair) and which of the five cards in the hand contribute to the ranking. E.g., in a two pair, four of the cards contribute and one doesn't.

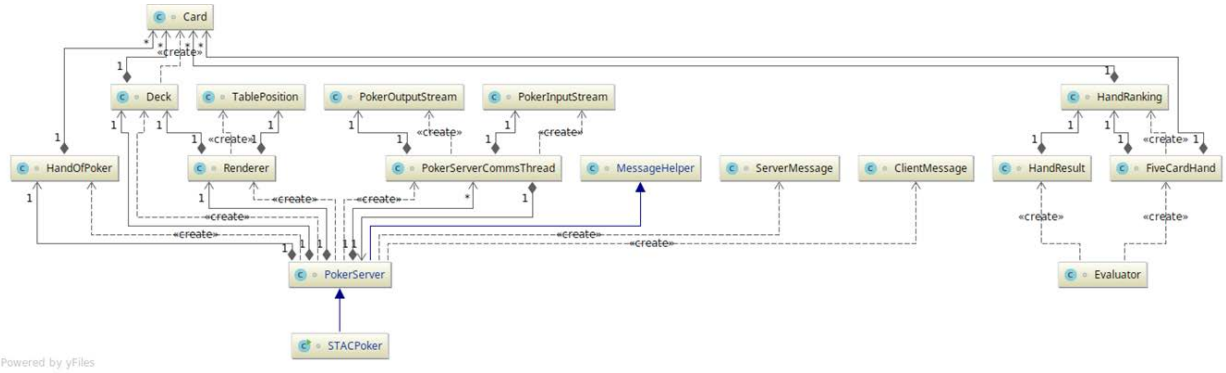


Figure 29. Poker Class Diagram

4.3.3.2.1 Vulnerability Description

The observable for the timing side-channel is only available after the card rendering cache is cleared. This happens when a player sends a valid setSkin message (cf. Inputs and Outputs section) to the server. Upon receiving the setSkin message, the changeSkin() method of Renderer.java is called:

```
/**
* Change the skin under which the cards are rendered.
* @param skinName The name of the skin to change to.
*/
void changeSkin(String skinName)
{
try
{
currentSkin = ImageIO.read(STACPoker.class.getResourceAsStream(
"/skins/" + skinName + ".png"));
skinnedBackOfCard = skinCardImage(Card.getBackImage());
currentSkinName = skinName;
skinnedCardsMap.clear(); // clear the cache here
}
…
}
```

After the card rendering cache has been cleared, the timing observable actually manifests in the drawTable() and reskinCard() methods of Renderer.java.

```
BufferedImage drawTable(…)
{
…
for(Card card : opponentsHoleCards)
{
if(card != null)
{
// If the skin was changed, then re-render the card under the new skin
if(!skinnedCardsMap.containsKey(card.toString()))
{
```

```
reskinCard(card.toString());
}
table = drawCardImageAtPos(table, skinnedCardsMap.get(card.toString()), pos);
pos++;
}
}
…
}

/**
* Re-render a card under the current skin if it's not already in the cache.
* @param c The name of the card to be re-skinned
*/
private void reskinCard(String c)
{
BufferedImage reskinnedCard = null;
// Reskin all the cards of the deck, but only cache the requested one.
for(Card card : deck.getCards())
{
BufferedImage skinned = skinCardImage(card.getImage());
if(c.equals(card.toString()))
{
reskinnedCard = skinned;
skinnedCardsMap.put(card.toString(), reskinnedCard);
}
}
}
```

At the start of a new hand of poker when the server deals the four hole cards to the players, the cards are dealt and transmitted to the clients in a specific order that is also important to the STAC vulnerability. The four hole cards are dealt to the two players, Player0 and Player1, in the following order: Player0, Player0, Player1, Player1. For each card that is dealt, the updated table is rendered and sent first to Player1 then to Player0. After each hole card is dealt and the updated table transmitted to both players, the server waits to receive an *Ack* message (cf. Inputs and Outputs section) from the Player0 client before it deals the next card. The provided well-behaved client always sends back an *Ack* message upon receipt of an updated table rendering irrespective of which player the client belongs to, however the server only cares about the *Ack* message from Player0. After the fourth hole card is dealt, no *Ack* message needs to be sent. Rather, at this point in the game, the player to whom the current turn belongs is expected to either send a *fold*, *call*, or *raise* message which itself serves as a form of acknowledgement.

The complete ordering of important events is therefore:

1. Deal a card to Player0
    a. Server renders the updated table for Player1 and sends it to him.
    b. Server renders the updated table for Player0 and sends it to him.
    c. Server receives *Ack* message sent from Player0.
2. Deal a card to Player0

a. Server renders the updated table for Player1 and sends it to him.

b. Server renders the updated table for Player0 and sends it to him.

c. Server receives *Ack* message sent from Player0.

3. Deal a card to Player1

    a. Server renders the updated table for Player1 and sends it to him.

    b. Server renders the updated table for Player0 and sends it to him.

    c. Server receives *Ack* message sent from Player0.

4. Deal a card to Player1

    a. Server renders the updated table for Player1 and sends it to him.

    b. Server renders the updated table for Player0 and sends it to him.

Assume Player0 is the attacker and Player1 is the victim. After the attacker is dealt his second hole card, he receives the updated table rendering showing both his hole cards (1b in the above list). The time observable the attacker uses to learn about the victim's first hole card is the elapsed time between when he sends his next *Ack* message (2c in the list of important events) and when he receives the updated table rendering resulting from the card dealt to the victim (3b). In between those two events, the victim is dealt his first hole card and receives the re-rendered table including that card (3a). If the hole card just dealt to the victim was already in the rendering cache, meaning that the same card was dealt previously under the current skin, then the time delta between 2c and 3b is small; otherwise the time delta is large.

To learn about the victim's second hole card, the attacker similarly monitors the time delta between sending his next *Ack* message (3c) and when he receives the updated table rendering resulting from the victim being dealt his second hole card (4b). In between, the victim is dealt his second hole card and receives the re-rendered table including that card (4a). If the second hole card dealt to the victim was already in the rendering cache, meaning that the same card was dealt previously under the current skin, then the time delta between 3c and 4b is small; otherwise the time delta is large.

The ordering in which the cards are dealt to the players is fixed by the runBlinds0Round(), runBlinds1Round(), runBlinds2Round(), and runBlinds3Round() methods of PokerServer.java. The order in which the updated tables are sent to the clients is fixed in updateClientViews().

```
/**
* Run the blinds betting round for the current hand, dealing the first card.
*/
private void runBlinds0Round()
{
…
currentRound.giveP0Card(deck.dealCard());
updateClientViews(false);
needAnAckFrom = PLAYER_0;
}
```

```
/**
* Run the blinds betting round for the current hand, dealing the second card.
*/
private void runBlinds1Round()
{
currentRound.giveP0Card(deck.dealCard());
updateClientViews(false);
needAnAckFrom = PLAYER_0;
}

/**
* Run the blinds betting round for the current hand, dealing the third card.
*/
private void runBlinds2Round()
{
currentRound.giveP1Card(deck.dealCard());
updateClientViews(false);
needAnAckFrom = PLAYER_0;
}

/**
* Run the blinds betting round for the current hand, dealing the fourth card.
*/
private void runBlinds3Round()
{
currentRound.giveP1Card(deck.dealCard());
updateClientViews(false);
…
}

/**
* Send a rendering of the table as it currently stands to both clients
* @param revealOpponentsHoleCards If true, the hole cards for the opposing
player are drawn face up. If false, the opposing player's hole cards are drawn
face down.
*/
private synchronized void updateClientViews(boolean revealOpponentsHoleCards)
{
sendTableToClient(PLAYER_1, revealOpponentsHoleCards);
sendTableToClient(PLAYER_0, revealOpponentsHoleCards);
}
```

4.3.3.2.2 Overall Code Structure

All communication between the clients and the server take place over secure SSL sockets. Clients only communicate with the server, never among each other. All messages between the client and server are in JSON.

The only library used is Google's GSON. It is used to serialize messages to JSON prior to transmission. The GSON library is an external .jar that gets copied to the correct place by the create_cp.sh build script.

4.3.3.2.3 Inputs and Outputs

Messages from a client to the server and from the server to a client are both stored internally as a Java Map<String, String>. Prior to transmission, they are serialized by GSON to a JSON object associative array. For example, an *action* message contains two key/value pairs, where the key "serverMessageType" maps to "action" and the key "whatHappened" maps to a string that is to be displayed by the client, e.g., "Player 1 folded.". Such an *action* message is then serialized to JSON as follows:

{"serverMessageType":"action","whatHappened":"Player 1 folded. "}

The server can send 12 types of messages to a client:

1. action - Notifies the client that some action was performed, usually by the opposing player.
    a. "serverMessageType" : "action"
    b. "whatHappened" : The message to be displayed on the receiving client.
2. gameStatus – After a card is dealt, this message conveys a new rendering of the poker table to be displayed by the receiving client and comma separated lists of the cards held by the two players. A message of this type is sent to both players after each card is dealt.
    a. "serverMessageType" : "gameStatus"
    b. "rendering" : Base 64 encoded PNG image of the table view for the receiving client.
    c. "holeCards" : Comma separated list of the hole cards held by the receiving client; e.g., "Four of Clubs, King of Spades".
    d. "opponentsCards" : Comma separated list of the hole cards held by opponent of the receiving client. If the hand has not yet ended, each of the opponent's cards is represented by "HIDDEN"; otherwise if the hand has ended then the value is like that of the value corresponding to the holeCards key.
    e. "tableCards" : Comma separated list of the shared cards dealt onto the table during the flop, turn, and river rounds.
    f. "youAre" : Either "Player 1" or "Player 2" depending on the recipient.
3. yourTurn – Sent to a player's client when it is that player's turn.
    a. "serverMessageType" : "yourTurn"
    b. "yourMoney" : The number of chips held by the recipient.
    c. "opponentMoney" : The number of chips held by the recipient's opponent.
    d. "howMuchToStayIn" : The number of chips the recipient must call in order to stay in the current hand.

e. "potSize" : The number of chips that were bet in previous betting rounds during the current hand. E.g., if the river was just dealt, then this is the number of chips that were bet during the blinds, flop, and turn betting rounds.

f. "youIn" : The number of chips bet by the recipient in the current betting round.

g. "opponentIn" : The number of chips bet by the recipient's opponent in the current betting round.

h. "youAre" : Either "Player 1" or "Player 2" depending on the recipient.

i. "numerOfAllowableCommands" : This yourTurn message is telling the recipient that it is his turn to act. The value associated with this key is the number of actions (i.e., commands) that are available to the recipient that he can send back to the server to continue the hand of poker. The only commands that are ever possible, though in various combinations, are *fold*, *check*, *bet*, *call*, and *raise*.

j. "0" : The first command available to the recipient; e.g., "fold".

k. "1" : The second command available to the recipient.

l. …

m. n where n is the value associated with the key "numerOfAllowableCommands" in this message. : The n^{th} command available to the recipient.

4. wait - Sent to a client when it is the other client's turn.

    a. "serverMessageType" : "wait"

    b. "yourMoney" : The number of chips held by the recipient.

    c. "opponentMoney" : The number of chips held by the recipient's opponent.

    d. "waitingOn" : Comma separated list of the actions currently available to the recipient's opponent.

    e. "potSize" : The number of chips that were bet in previous betting rounds during the current hand. E.g., if the river was just dealt, then this is the number of chips that were bet during the blinds, flop, and turn betting rounds.

    f. "youIn" : The number of chips bet by the recipient in the current betting round.

    g. "opponentIn" : The number of chips bet by the recipient's opponent in the current betting round.

5. notYourTurn – Sent to a player's client when it has sent an otherwise valid poker command but it is not that client's turn.

    a. "serverMessageType" : "notYourTurn"

6. cantAffordCurrentBet – Sent to a player if they tried to make a bet that they could not afford.

    a. "serverMessageType" : "cantAffordCurrentBet"

7. endOfHand – Sent to both clients when a hand concludes, either when one of the players folds their hand or when betting concludes at the end of the river round.
    a. "serverMessageType" : "endOfHand"
    b. "winner" : "yes" if the recipient won the last hand, otherwise "no"
    c. "yourMoney" : The number of chips held by the recipient.
    d. "opponentMoney" : The number of chips held by the recipient's opponent.
    e. "tie" : "yes" if the last hand ended in a tie, otherwise "no". If "yes", the client should disregard the "winner" key of this message.
    f. "winAmount" : The number of chips won by the winner of the last hand.
    g. "yourHand" : If the hand was played to completion, then this holds the rank of the recipient's hand (e.g., Full House) and the cards that he held. Otherwise, empty string.
    h. "opponentsHands" : If the hand was played to completion, then this holds the rank of the recipient's opponent's hand (e.g., Full House) and the cards that he held. Otherwise, empty string.
8. badCommand – Sent back to a client that sends an otherwise valid poker command but which is not allowed right now. In other words, it is the client's turn but they sent a command that wasn't among the allowable commands specified in the yourTurn message that was previously sent to them by the client.
    a. "serverMessageType" : "badCommand"
9. blinds - Sent to both clients at the start of each hand to notify them that blinds for the current hand were posted.
    a. "serverMessageType" : "blinds"
    b. "which" : "big" or "small" depending on whether the recipient posted the big or the small blind.
10. outOfMoney – Sent to a client if they cannot afford to post the blinds at the start of a new hand. Upon receipt of this message, a client should automatically disconnect itself from the server.
    a. "serverMessageType" : "outOfMoney"
11. otherClientDisconnected – Sent to a client when the opponent's client disconnects from the server.
    a. "serverMessageType" : "otherClientDisconnected"
12. listOfSkins – Sent in response to a client that sent a listSkins message to the server.
    a. "serverMessageType" : "listOfSkins"

b. "numberOfSkins" : The number of skins to which the server has access and from among which the player may choose.

c. "0": The name of the first available skin.

d. "1": The name of the second available skin.

e. "…" (next available command, repeated).

f. n where n is the value associated with the key "numberOfSkins" in this message. : The name of the n^{th} available skin.

A client can send 8 types of messages to the server:

1. fold – Sent when the player using the client wishes to fold his current hand.
    a. "clientMessageType" : "fold"
2. call – Sent when the player using the client wishes to call his opponent's previous bet.
    a. "clientMessageType" : "call"
3. bet – Sent when the player using the client wishes to bet. The betting amount is hardcoded to 2 chips.
    a. "clientMessageType" : "bet"
4. raise - Sent when the player using the client wishes to raise. Raises always double the current bet on the table.
    a. "clientMessageType" : "raise"
5. check - Sent when the player using the client wishes to check.
    a. "clientMessageType" : "check"
6. listSkins – Sent when the player using the client requests the list of available card skins on the server.
    a. "clientMessageType" : "listSkins"
7. setSkin – Sent when the player using the client wants to change the skin under which cards are rendered.
    a. "clientMessageType" : "setSkin"
    b. "skinName" : The name of the skin desired by the player. The name should be among those the server sent in its listOfSkins message in response to the client's listSkins message.
8. ack – Automatically sent by a client upon receipt of a gameStatus message from the server.
    a. "clientMessageType" : "ack"

4.3.4 SearchableBlog

4.3.4.1 Description

This challenge application is a blogging website that ranks its blogs according to their importance. When a new user submits a blog, the application determines the internal links between blogs and uses this information to update its site wide importance ranking. The ranking algorithm itself is run automatically after a user submits a blog.

The challenge contains a **time complexity vulnerability**. The importance ranking will use an algorithm similar to Google's PageRank [9], which at its core is an iterative method to find an eigenvector of a matrix. The problem is that these iterative eigenvector methods can fail to converge quickly or at all, if certain criteria of the matrix are not met. Namely, if the two largest eigenvalues (the *dominant* and the *subdominant*) are equal or very close in magnitude, the algorithm will have potential convergence issues. The difference between potential and actualized bad convergences lies in the initial vector chosen to begin the iterative process.

Unlike general eigenvector methods, for the PageRank algorithm the raw matrix is first processed (in particular by adding a damping factor α) in a way that ensures a sufficient gap between the eigenvalues and hence good convergence of the algorithm. So when a new user submits their blog, a new column is concatenated to the existing raw matrix and then is processed. However, if we modify the algorithm to separately process the existing matrix and the user's column first and then concatenate them, then it is still possible for the resulting matrix to have a much worse convergence rate then the PageRank algorithm usually guarantees. The second modification needed will be to set the damping factor α to a number much closer to 1 than the factor 0.85 used in Google's algorithm. The modifications and their consequences will be discussed in more detail in the Description of the PageRank Algorithm section.

4.3.4.2 Software Design

4.3.4.2.1 Data

There is a folder of HTML files representing the preexisting network of blogs. Each of these is a valid HTML file containing links to other blogs as well as some decoy content (Markov bot word vomit). Before these HTML files were generated and packaged with the program, the links between them were determined from a matrix that ensured the desired algorithm runtime. If we stipulate N blogs in total (not including the user's), there will be N HTML files provided. This will make the initial matrix NxN, but after incorporating the user's blog vector, the matrix on which PageRank runs will be (N+1)x(N+1).

In order to increase the severity of the worst case, a specific initial vector may also be provided for the iterative algorithm. This vector of doubles is size N+1.

The current value of N is 729.

4.3.4.2.2 Inputs and Outputs

The format of the input is an HTML file. The output is just a notification, either of successful or invalid input. Upon successful input the matrix is processed and passed to the vulnerable algorithm, and then the user is notified. The results of the algorithm itself are not returned directly

to the user but searches for blogs by title keyword will be returned in the order of their page rankings.

User inputs are listed in Table 15.

Table 15. Searchable Blog User Inputs

HTTP Method	Path	Description
POST	/submit	A user may post a valid HTML file as a blog to be ranked.
GET	/title?keyword={}	A user may search for blogs by entering a keyword and the server will return JSON of titles containing that keyword and the blog filename corresponding to that title. They will be ranked in order of their page rank score.

Outputs from the server are listed in Table 16.

Table 16. Searchable Blog Server Outputs

HTTPs Method	Path	Description
GET	/blog/{blogname}.html	Allows a user to view a blog where blogname is the filename of any valid blog, i.e. blog0.html, blog1.html, etc.
GET	/submit	Shows a page where a user may submit a blog by POSTing a file to the server.
GET	/keywords	Server returns JSON containing a list of all keywords currently found in blogs' titles.

4.3.4.2.3 Vulnerability Design

The SearchableBlog application, depicted in Figure 30, consists of the following major components:

- **BlogContentController:** Spring Model-View-Controller (MVC) controller that serves all blog content (HTML files). Additionally, it implements the /submit endpoint where a user may submit a blog for ranking by making a POST request.
- **BlogSearchController:** Spring MVC controller that implements the /keywords and titles?keyword= endpoints. The /keywords endpoint returns JSON data of the form {keyword, [blogname, …]} where keyword is a valid title keyword (any word found in a blog title that is greater than 3 characters in length) and blogname is a list of blog names whose title contain this keyword. The /titles?keyword=query endpoint allows a user to submit a keyword of their choice as query and obtain JSON data of the form {title, blogname, …} where title is the full title that contains the queried keyword and blogname is the name of the blog with that title.
- **MatrixRoutines:** Methods that manipulate the matrix M and includes the core vulnerability. For further discussion of these methods, see "Description of PageRank Algorithm" and "Discussion of processing and concatenation routines".

- **ParsingRoutines:** Methods that assist in parsing the blog files and submitted user blog, including collecting a vector representing the links from a blog to other blogs.
- **RankingService:** Calls initial preprocessing routines and processing routines. After a user submits a blog, the method performRanking() is called to rank the newly submitted blogs against the canned data (blogs found in data/blogs/).
- **TitleService:** Supports searching for blogs based on title keywords. On starting the application, it parses the titles of all the blogs found in data/blogs/ and adds their file names and titles to a HashMap so that a querying for a keyword will return blogs whose titles contain that keyword.
- **WebConfigurer:** Simply adds the data/blogs/ directory to a template resolver so that BlogContentController can serve the blogs found in this folder to the user.

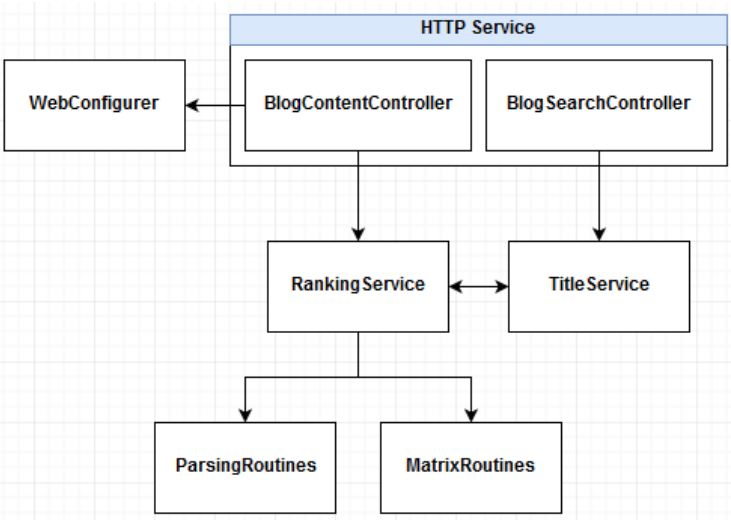


Figure 30. SearchableBlog Major Components

Description of PageRank Algorithm

The intent of the PageRank algorithm is to provide a ranking of a collection of webpages by importance, using nothing but the structure of the directed links between webpages. Similar algorithms are used in other contexts, i.e. to determine the influence of scholarly articles using the citations between them or to study the intercommunication of the Linux kernel using its procedure calls. The precise interpretation of a page's importance is the proportion of time a random web surfer will spend on this particular page while surfing the web, using a simple probabilistic model of a surfer. The probabilistic model is intended to model the way a surfer would behave but, more importantly, is constructed so that the PageRank is the unique solution to a linear algebra problem and the iterative methods used to calculate it will have good convergence properties.

At first approximation, we might say the random surfer, once on a webpage, has an equal probability of following any link from that page to another page. As an example, 4 outgoing links mean that there is a 0.25 probability a surfer will follow any one of them. In the case where a page

has no outgoing links, the surfer is equally likely to jump to any other page. However, this neglects the reality that there is always a chance the surfer will jump directly to a page that is not linked by the current page. After all, some pages are never linked to, yet still get surfers. This issue is addressed by the inclusion of the *damping factor* α, a number between 0 and 1 which is the probability that the surfer will indeed follow some link from the current page instead of typing in a different URL. The complement probability, (1- α), is the probability that the surfer will pick the next page at random – the "jumping factor". In the first approximation, the damping factor was just $\alpha = 1$, but Google has historically used a factor of 0.85. The significance of the damping factor is that it will equal the subdominant eigenvalue and so directly controls the upper bound on convergence of the algorithm.

This probabilistic model can be described with a discrete Markov process, which can be thought of as a graph with a weighted, directed edge from each node to each other (so a node has an edge to itself, and there is an edge in each direction between distinct nodes), such that the weights of all edges originating at a particular node must sum to 1. Here, there will be a node for each webpage, and we interpret the weight of an edge from page x to page y as the probability that a surfer currently on page x will next move to page y. According to the above description, if page x has n outgoing links and there are N pages in total, there will be a probability of $\alpha(1/n) + (1- \alpha)(1/N)$ to visit to one of the linked pages, and a probability of $\alpha(0) + (1- \alpha)/(1-N)$ to visit a non-linked page. Each of these probabilities is written as the sum of two terms, in order to illustrate how α interpolates between the model where the surfer randomly chooses among the links on the current page and the model where the surfer chooses randomly among all pages on the web. In the parlance of network flow on the directed graph, the damping factor eliminates the sources and sinks, and this is strongly tied to the irreducibility of the matrix and the convergence of the algorithm.

A discrete Markov process (with finite state space) can be represented using a *column stochastic* matrix, i.e. a square matrix in which each entry is a real number between 0 and 1 such that the sum of the entries in any column is exactly 1. It has a close connection to the adjacency matrix of the directed graph. The value at entry (i, j) corresponds to the weight on the edge from node j to node i. The entries in column j correspond to the weights of the edges *going out of* node j. The entries in row i correspond to the weights of the edges *coming into* node i. Whereas the sum of the outgoing weights from node i is always equal to 1, the sum of incoming weights to node i is proportional to the probability that the next page the surfer visits will be page i, and so have some relation to the importance of page i. To find the actual importance value, we need to consider powers of the matrix. If the matrix satisfies certain conditions (irreducibility, aperiodicity) then as we take higher and higher powers of the matrix, the entries across a row i will all converge to the same number, which is the probability that the surfer ends up on page i after an arbitrary amount of time on the web, and equivalently the PageRank of page i.

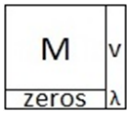
It turns out that the PageRank values are also equal to the entries of the dominant right stochastic eigenvector of this matrix, which means the right eigenvector for the largest eigenvalue, scaled so that the sum of the entries is 1. By the Frobenius-Perron theorem all entries of this eigenvector will be non-negative. The simple iterative method described in the pseudocode is a convenient, computationally feasibly method for calculating the dominant right eigenvector, as long as the matrix is irreducible and the second largest eigenvalue is significantly smaller than the largest (0.9 is small enough compared to 1, but 0.999 is starting to get too close).

Discussion of processing and concatenation routines

The primary processing functions are: makeVectorStochastic, makeMatrixStochastic, addDampingFactor, and concatenateColumn. makeVectorStochastic and makeMatrixStochastic take in a vector and a matrix, respectively, assumed to be an adjacency vector or matrix (i.e. to consist of 0s and 1s), and return a stochastic vector or matrix, according to the previous description of the Markov model. If the sum of the entries is nonzero, makeVectorStochastic returns the vector with each entry divided by that sum. If the sum of the entries is nonzero, makeVectorStochastic returns the vector with each entry divided by that sum. If the sum of the entries is zero, it sets every entry to 1/n. makeMatrixStochastic takes a matrix, assumed to be an adjacency matrix, and converts every entry x to the quantity $\alpha x + (1- \alpha)(1/N)$. The preprocessing consists of running makeMatrixStochastic on the adjacency matrix, running makeVectorStochastic on the user's adjacency vector and then running addDampingFactor on each. The methods processVector and processMatrix perform these operations. Finally, concatenateColumn is run on the matrix together with the user's vector.

concatenateColumn takes a matrix M and a column vector v, where M is assumed to be square of size NxN, and v is assumed to be length (N+1). This is because M consists of the N column vectors of length N, containing link information to the N preexisting blogs and v is a column vector containing link information to the N preexisting blogs in addition to the user's new blog.

TableconcatenateColumn returns an (N+1)x(N+1) matrix M formed, as Figure 31 illustrates, by putting M in the upper left corner and appending v to the right side. Since v is one element longer than M (and that element is λ), we must introduce a new row underneath M, which we do by filling in zeros. If M and v were simply adjacency matrices, the interpretation of those zeros would be that none of the preexisting blogs has a link to the new blog.



λ	self-link	no self-link
n > 0	$\alpha/n + (1-\alpha)(1/N)$	$(1-\alpha)(1/N)$
n = 0	not possible	1/N

Figure 31. Return Matrix

If we had appended v to M and then performed the processing routines on the resulting augmented matrix, the elements on the bottom row would not be zeros. The crucial consequence of performing the concatenation last instead of first is that the resulting matrix is block upper triangular, with the top block being the original NxN matrix M and the bottom block being the 1x1 matrix consisting of the last element in the user's vector, denoted in Figure 31 as λ. The eigenvalues of are all the eigenvalues of M together with the additional eigenvalue λ. The preprocessing performed on v ensures that λ will be no larger than α, so the algorithm must converge in finitely many steps. Values for λ are given in the table.

Keeping in mind that N will generally be large and α close to 1, this formula shows that if there is a self-link and n = 1 then λ is roughly equal to α, if n = 2 then λ is roughly equal to α/2 and so on. Therefore, the only time that λ can be close to 1 is when n = 1 and there is a self-link, in which case λ is the subdominant eigenvalue and bad convergence is possible. If there is no self-link, λ will always be close to 0.

The Frobenius-Perron theorem together with facts about stochastic matrices guarantee that if a square matrix has strictly positive entries with the columns summing individually to 1, then 1 is the eigenvalue of largest magnitude and all the other eigenvalues have strictly smaller magnitude. Since does not have strictly positive entries due to the row of zeros at the bottom, the Frobenius-Perron theorem does not directly apply. However, the theorem applies to the original matrix M and the eigenvalues of block upper triangular matrix are the union of the eigenvalues of the block matrices on the diagonal. Together this implies that all eigenvalues of except 1 and λ are strictly less than 1 but, as described above, λ is always less than 1 also.

4.3.4.2.4 Algorithm Pseudocode

The ranking algorithm starts with an initial vector b and then repeatedly applies the matrix to b. When the amount of change resulting from the matrix operation becomes very small, the vector has homed in on the desired eigenvector r. The initial vector can be thought of as a guess for the ranking vector r, also called the stationary vector, but any choice of b will give (approximately) r as the result, as long as b has nonzero component in the direction r, i.e. the dot product of b and r is nonzero. In the case where b does have zero component in the direction of r, the algorithm will still converge (probably faster) but will produce the zero vector as output. In our algorithm, only the convergence behavior and not the result is important, so our initial vector b is chosen to be a small perturbation of the subdominant eigenvector. This choice ensures the desired convergence behavior but, in most cases, the result will be the actual ranking vector anyway.

Though the algorithm is very simple, it relies heavily on the type of matrix being input. For a general matrix, the termination condition has no guarantee of being met. However, the structure of the links in the provided HTML files with our matrix preprocessing routine ensure that the condition will be met in a finite number of steps.

```
Vector estimateStationaryVector( Matrix A, Vector b, double precision) {
If( b.isZeroVector() )
return b // Avoids the exception caused by scaling a zero vector to norm one.
b.scaleToNormOne()
c = A * b // For the termination condition, we always want c to be the vector
that
// results from multiplying by A one more time than b.
while( distance(b,c) > precision ) { // See if the matrix operation still has
significant effect.
b = c // Take c to be the current vector.
c = A * b // Find the result of applying A once more.
if( c.isZeroVector() ) // The algorithm has converged to zero,
return c // but escape early to avoid exception in scaleToNormOne
c.scaleToNormOne() // Normalize length of vector to prevent size from getting
too
// large or small (plays well with floating point operations, etc.)
}
return b
}
```

4.3.5 Tawa-fs

4.3.5.1 Description

Tawa-fs is a file sharing service that stores all files of a given user in a separate file system. Tawa-fs uses HTTP to let each user manipulate files in his own file system. To decouple from the operating system's file system drivers, Tawa-fs uses a regular file as a container in which to store its own internal file system, with a separate container-file per user. Tawa-fs's internal filesystem is similar to that of FAT-32 – it has a file allocation table with some extended metadata fields specific to file sharing tasks. The internal file system of Tawa-fs is susceptible to fragmentation so the service implements its own defragmentation algorithm as well.

Tawa-fs contains both a space complexity vulnerability and a time complexity vulnerability.

The *space complexity* vulnerability comes from several non-local sources.

1. The implemented file system has a single reader/writer object associated with it. All read/write operations are done through it.
2. Due to the nature of an HTTP service, all user request spawn separate threads for processing. Most of those threads will put a lock on the reader/writer object above.
3. Defragmentation will only consider moving fragmented files, so the Blue Teams need to transition the file system into a fragmented state.
4. The user of Tawa-fs has a UI Button to start the defragmentation process if they feel that their file system needs defragmentation.
5. Like many file systems, Tawa-fs implements a hashing function to make sure that its blocks can be copied correctly. After a block copy, the before and after hashes are checked using the faulty implementation: "Integer.valueOf(hash1) == Integer.valueOf(hash2)". Because of the way Java chooses to cache these "boxed" integers, this check will fail for hash values above 127. Therefore, the hashes will be deemed mismatched for some sectors even when no mismatches happen. At this point, the defragmenter will start copying the block it deemed erroneous, byte by byte to a separate location. After copying that block, the defragmenter will compare the copy byte-by-byte. If bytes match, the defragmentation process will continue as intended.
6. Unlike most user-called code, the defragmenter's logic takes longer than a few milliseconds, so the user gets locked out from changing the file system while the defragmenter is running.
7. When the defragmenter fails to copy the block, it moves it to a "backup region" which can grow indefinitely (limited by physical hard drive space)
8. The frontend prevents the user from writing the files when defragmenter is running. However, nothing stops the user from sending a custom write request for a given file to the backend. This will change the block content and even the byte-by-byte comparison will fail, causing the defragmenter to keep backing up a block to the backup region until the whole hard drive is occupied.

9. If an adversary sends that write request often enough (before defragmenter finishes copying sector) a racing condition in (6) will form, causing the defragmenter thread to read the sector over and over again while writing a continuous stream of repeated data.

For all of this to happen the adversary needs to understand the condition for which the defragmenter will think that the sector copy is mismatched (sector's hash is above 127), create a file such that one or more of its sectors' hashes is above 127, initiate the defragmentation process, and finally send in a loop of custom GET requests to that file.

The *time complexity* comes from a missing failsafe for creating a file/folder with name '/' and a path joining function. For path joining, '/Documents/' + '/' == '/Documents/'. The defragmentation process recursively finds all of the files in a file system to check if the file system needs defragmentation. Then it path joins current name to the current directory and if it is a file – checks its fragmentation, or if it's a directory – calls itself over it. If a file or folder with name '/' exists inside another folder, the recursive function will call itself on the same folder until the JVM stack gets full and JVM kills the thread with an exception. Tawa-fs will spawn a maximum of 2 threads for client handlers. This means that if an attacker creates the '/' file/folder and calls the defragmentation process two times at the same time, it will use all of the available threads and block benign inputs until JVM kills the first two threads. For an increased benign delay, the attacker can spawn more than 2 threads of defragmentation process instead. The client handler uses a first in-first out (FIFO) scheduler, and the benign request will have to wait until all but the last thread are free.

4.3.5.2 Software Design

Entry point is Tawa-fs class (Figure 32). It will create a new BlockDevice per registered user. BaseBlock.equals() contains the CRC8 check trigger. FileTable.moveBlock() triggers erroneous BaseBlock.equals(), and fails back to FileTable.backupBlock() that does a byte-by-byte comparison. If the separate thread triggers Filesystem.write() while FileTable.backupBlock() is running, the BlockDevice.dataFile will grow in size without a limit. RandomAccessFile is a Java class: java.io.RandomAccessFile;

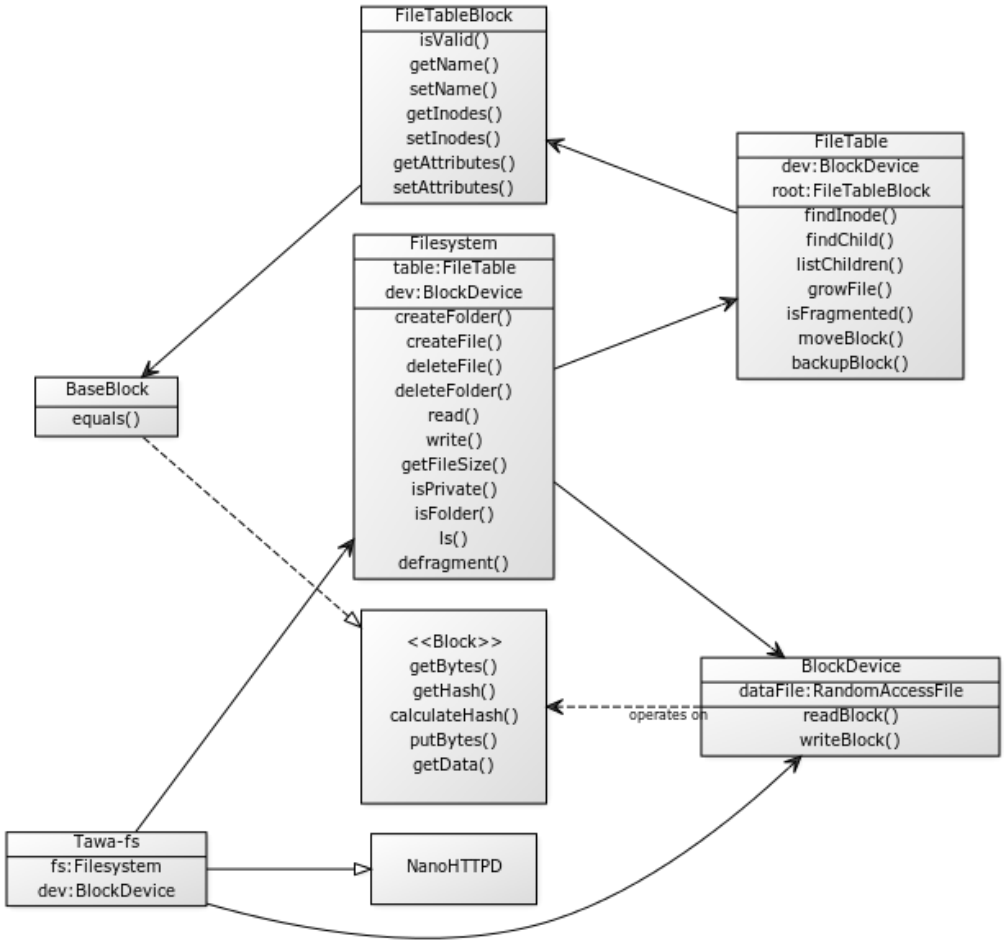


Figure 32. Tawa-fs Software Software Components

4.3.5.2.1 Vulnerability code(SC vulnerability)

Block hash comparison:

```
public boolean equals(BaseBlock block) {
return (Integer.valueOf((int) block.calculateHash()[0] & 0xFF) ==
Integer.valueOf((int) calculateHash()[0] & 0xFF));
}
```

Block backup logic:

```
public short backupInode(short parent, short blockId) throws IOException {
FileTableBlock parentBlk = getFile(parent);
if (parentBlk.usedInodes() < blockId || blockId < 2) // can not move unused
blocks or structure nodes
throw new CanNotMoveBlockException("Bad source block ID");
short fileInodes[] = parentBlk.getInodes();
Block srcBlk = dev.read(fileInodes[blockId]); // read the block to be moved
byte oldHash = srcBlk.getHash()[0]; // get the hash as stored
```

```
if (oldHash != srcBlk.calculateHash()[0]) { // recalculate hash, compare to
storage
throw new DataWasLostException("File "+parentBlk.getName().trim()+" is
corrupted :(");
}
// if we are here, the hash matches, let's relocate
byte[] sourceBytes = srcBlk.getBytes();
long backupRegion = dev.getSize() - Block.BLOCK_SIZE; // backup region is
after the block device's original size.
// Since our "BlockDevice" is a file, underlying file system lets us write
past its current size
// This also accounds for any previously backed up blocks - those have
already caused the file size to grow
int errorByte = Block.BLOCK_SIZE;
do {
backupRegion += Block.BLOCK_SIZE; // try to go to the next block in the
backup region
for (int cByteIndex = 0; cByteIndex < sourceBytes.length; ++cByteIndex) {
dev.writeByte(backupRegion+cByteIndex, sourceBytes[cByteIndex]);
}
// slowEquals re-reads the block content. If the block content is overwritten
in another thread,
// slowEquals will return false
errorByte = slowEquals(fileInodes[blockId], (short)
(backupRegion/Block.BLOCK_SIZE));
} while (errorByte < Block.BLOCK);
// if we are here, we successfully backed up the block. Adjust its owner's
index, update the parent inode
parentBlk.setInode(blockId, (short) (backupRegion/Block.BLOCK_SIZE));
dev.write(parent, parentBlk);
return (short) (backupRegion/Block.BLOCK_SIZE);
}
/**
* Performs byte-by-byte slow comparison
* @param inode1
* @param inode2
* @return
* @throws IOException
*/
private int slowEquals(short inode1, short inode2) throws IOException {
Block block1 = dev.read(inode1);
Block block2 = dev.read(inode2);
byte[] b1bytes = block1.getBytes();
byte[] b2bytes = block2.getBytes();
for (int pos=0; pos<b1bytes.length; ++pos) {
if (b1bytes[pos] != b2bytes[pos])
return pos;
}
return Block.BLOCK_SIZE;
}
```

File writing code:

```
/**
* Write data to a file, growing the allocation as needed
* @param fullPath of the file to write into
* @param content to be written
* @return the inode of the modified file
* @throws IOException
*/
public short write(String fullPath, byte[] content) throws IOException {
short fileInode = table.findInode(fullPath);
table.growFile(fileInode, content.length);
FileTableBlock fileEntry = table.getFile(fileInode);
short[] pInodes = fileEntry.getInodes();
BaseBlock fileDataBlock = new BaseBlock();
for (int i=2; i<pInodes.length && pInodes[i] != -1; ++i) {
fileDataBlock.put(Arrays.copyOfRange(content, (i-2)*BaseBlock.DATA_SIZE, (i-
1)*BaseBlock.DATA_SIZE));
dev.write(pInodes[i], fileDataBlock);
}
return fileInode;
}
```

4.3.5.2.2 Vulnereability code (TC vulnerability)

```
/**
* List all of the data files this file system has (i.e. not folders)
* @param path to start search at
* @return a list of strings, each is a full path to a file
* @throws IOException
*/
private List<String> findAllFiles(String path) throws IOException {
List<String> fileList = new ArrayList<String>();
String cFolder = path;
Set<String> cChildren = table.listChildren(table.findInode(cFolder));
FileTableBlock cFileBlk;
for (String child: cChildren) {
if (!child.equals(".") && !child.equals("..")) {
String newPath = new String(cFolder);
if (!newPath.endsWith("/"))
newPath += "/"+child;
else
newPath += child;
cFileBlk = table.getFile(table.findInode(newPath));
if (!cFileBlk.getAttributes().isSet(BitAttrs.FOLDER)) {
fileList.add(newPath);
} else {
fileList.addAll(findAllFiles(newPath+"/"));
}
}
}
return fileList;
}
```

4.3.6 StacSQL

4.3.6.1 Description

The StacSQL challenge program is a simplistic implementation of an SQL database. It supports basic SQL statements for creating, modifying, and deleting databases, tables, and values. The challenge presents a telnet-like interface on a network port which reads SQL statements. The parser then builds a series of actions, logs them to a file, and executes them. Each entry follows a schema that allows data types similar to SQLite. Each field in the entry is compressed using Huffman coding.

StacSQL contains a **space complexity vulnerability**. All fields in the database are compressed using a modification of the standard Huffman coding algorithm. The code length in bits gets determined on the first information insertion. However, all but the blob type recalculate the symbol distribution for the tree building upon value update. Fields containing blob data use the existing Huffman tree as a starting point when adding new symbols, and the updated tree is used to encode updated data blobs. This allows the attacker to hand-craft the coding tree, through a series of field updates, in such a way that a one-byte input symbol will be substituted by a code of multiple bytes. By substituting a single byte of data in multiple rows with a large amount of data containing the symbol with the largest Huffman code, one can create a huge database file using a relatively small amount of input data.

4.3.6.2 Software Design

4.3.6.2.1 Related Class Roadmap

Figure 33 depicts the related classs roadmap.

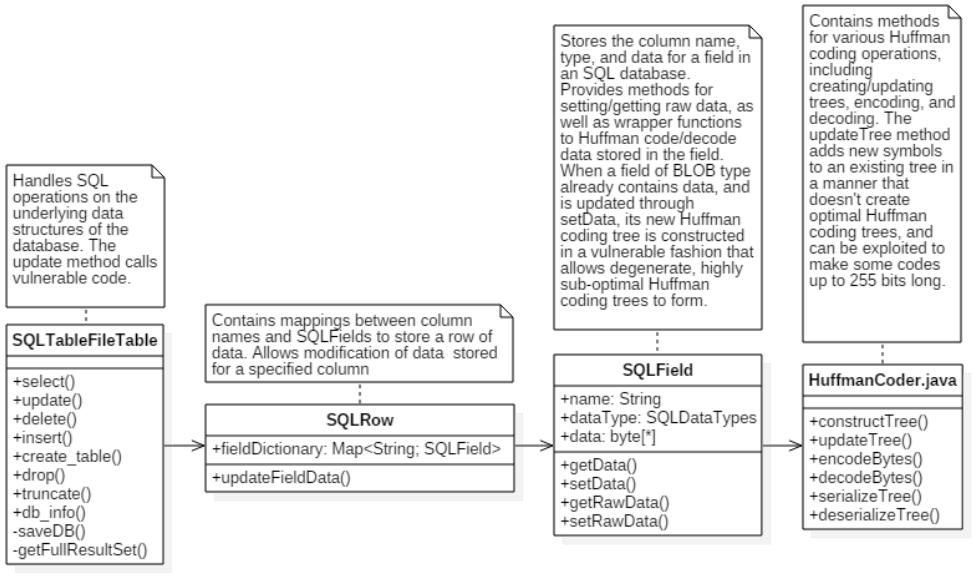


Figure 33. Classes Related to Vulnerable Execution, along with Their Key Members and Descriptions

- **HuffmanCoder:** Contains routines for building Huffman coding trees for an input data set, encoding data with a tree, decoding data with a tree, and updating a tree with new symbols. The method to update the tree produces sub-optimal Huffman coding trees that can be crafted to contain codes up to 255 bits long, which is the intended vulnerability of this challenge.
- **SQLField:** Contains information about a field stored in the database. Contains routines for reading/writing data to/from the field and handles Huffman coding the data to save space. For fields containing blob data, it simply updates the old stored tree if new symbols are found in new data being written to the field, which can be used to produce malicious Huffman coding trees.
- **SQLTableFileTable:** Contains routines for performing standard SQL commands on the database. Performs the update operations by setting field data using the vulnerable methods in the SQLField class.

4.3.6.2.2 Vulnerability code

The vulnerability present in updateTree is simply that the algorithm is not guaranteed to produce optimal trees (which is not the case in the standard Huffman coding tree construction algorithm). The updateTree method simply constructs a tree from the new symbols, and then combines the old tree and new tree as children under a new root node, which causes the max code length to increase by 1 every time a new symbol is added to the tree by an update. This allows an adversary to construct a Huffman coding tree where a symbol of 1 byte will encode to a code of 255 bits (regardless of its frequency in the updated text), which (including padding) will take 32 bytes of storage per occurrence of the symbol.

4.3.6.2.3 Use Cases

The user is able to connect to the service by connecting a socket to the correct host:ip (on port 8394). The client must support SSL in order to establish a connection to the server. Once connected, the client can send standard SQL commands to be executed by the server:

- CREATE – Allows the user to create a table in the database (CREATE DATABASE isn't supported).
- DROP – Allows the user to drop a table from the database.
- INSERT – Allows the user to insert a row containing some data into a table.
- SELECT – Retrieves the specified database contents and prints them.
- TRUNCATE – Deletes all rows from a given table.
- UPDATE – Updates the contents of a row.
- DELETE – Deletes rows that match a given condition.

In addition, the client can also use the following special keywords:

- HELP – Prints a list of all available commands.
- DBINFO – Prints information about the database including its tables, their column types/names, and the size of the database file.
- EXIT – Drops the connection between the client and server.

The user can also type "help" after any of these keywords (e.g. "CREATE HELP") to get more information about the command, including its proper syntax.

4.3.6.2.4 Overall code structure

Figure 34 summarizes the StacSQL class.

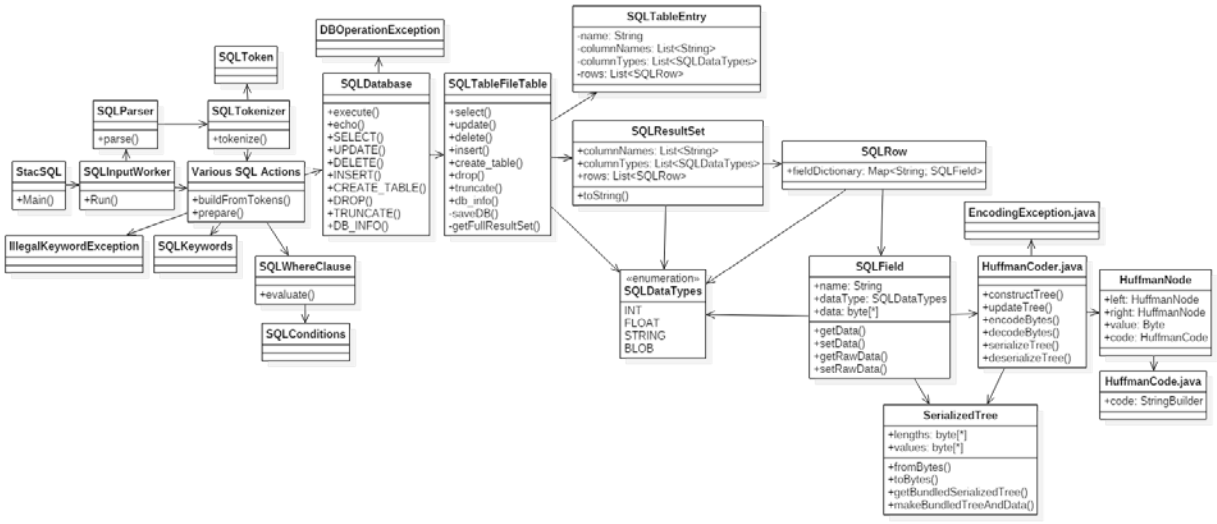


Figure 34. StacSQL Classes - The Most Important Members of Classes are Shown (Less Important Members are Omitted for the Sake of Brevity)

Non-Vulnerable Classes:

- **StacSQL:** Sets up the networking for the service, and handles requests in a while(true) loop.
- **SQLInputWorker:** Reads client input, and passes it to the tokenizer then to the database.
- **DBOperationException:** Exception thrown on error while performing a database operation.
- **SQLConditions:** Enum for supported conditions in WHERE clauses. (=, <, >, != supported)
- **SQLDataTypes:** Enum for supported SQL data types (INT, FLOAT, STRING, BLOB).
- **SQLDatabase:** Provides methods to perform standard SQL commands on the database, and get the resulting output.
- **SQLResultSet:** Stores rows and other info returned as a result of a WHERE or SELECT statement. Also provides toString representation of this data.

- **SQLRow:** Stores a mapping between column names and fields in a given row in the database.
- **SQLTableEntry:** Stores information about a database table, including its name, column names/types, and rows.
- **SQLWhereClause:** Stores the components of a WHERE clause, and provides a method to evaluate the clause against the value of a SQLDataType.
- **HuffmanCode:** Stores a code for use in Huffman coding.
- **HuffmanNode:** A node in a Huffman coding tree. Stores children nodes, the value encoded by the node, and the code represented by the node's position in the tree.
- **SerializedTree:** Stores arrays of code lengths and symbol values for use in building canonical Huffman trees. Provides methods for reading this info out of an array of bytes, or storing it into an array of bytes.
- **SQLParser:** Handles transformation of an input sql command string to a list of tokens, and finally to a list of SQL actions to be executed by SQLInputWorker.
- **IllegalKeywordException:** Exception thrown when an illegal keyword or illegal syntax is found while parsing SQL commands.
- **SQLKeywords:** Enum for the SQL keywords used by this program.
- **SQLToken:** Stores information about a token in an SQL statement. Provides methods for determining if the token is a keyword or literal, and stores the token's position in the input string.
- **SQLTokenizer:** Preprocesses a string to get it into an easy format to tokenize, then splits tokens using spaces as a delimiter.
- **SQL*Action:** Various parsing routines for turning SQLTokens into SQLActions that can be executed by SQLInputWorker.

4.3.6.2.5 Inputs and Outputs

The connection to the server is a normal connection to a socket, and is encrypted using SSL. The connection should be made to port 8394.

The supported commands, their syntax, and their response formats are as follows:

- CREATE
    - INPUT: CREATE TABLE *table_name* (*column_1 type_1 [, column_x type_x, …]*)
    - OUTPUT: "Successfully created table *table_name*."
- DROP
    - INPUT: DROP TABLE *table_name*
    - OUTPUT: "Successfully dropped table *table_name*."

- INSERT
    - INPUT: INSERT INTO *table_name [*(*column_1 [, column_x, …])]* VALUES (*value_1 [, value_x, …]*)
    - OUTPUT: “Successfully inserted values.”
- SELECT
    - INPUT: SELECT * | *column_1 [, column_x, …]* FROM *table_name [*WHERE *column_y condition value_y]*
    - OUTPUT:

      column_1 column_2 …

      (*type_1*) (*type_2*) …

      row_value_1 row_value_2 …

      ……
- TRUNCATE
    - INPUT: TRUNCATE TABLE *table_name*
    - OUTPUT: “Successfully truncated table *table_name*.”
- UPDATE
    - INPUT: UPDATE *table name* SET *column_1 = value_1 [, column_x = value_x, …] [*WHERE *column_y condition value_y]*
    - OUTPUT: “Successfully updated *updated_row_count* rows.”
- DELETE
    - INPUT: DELETE FROM *table_name [*WHERE *column_y condition value_y]*
    - OUTPUT: “Successfully deleted *deleted_row_count* rows.”
- HELP
    - INPUT: HELP
    - OUTPUT: A list of all available commands each on its own line.
- DBINFO
    - INPUT: DBINFO
    - OUTPUT:

      “Info for DB:

      Tables:

      [table_1_name [*column_1, [column_2, …]*] [*type_1, [type_2, …]*] (*table_num_rows* rows)\r\n

, …]

Total size: *db_file_size* bytes."

- EXIT
  - INPUT: EXIT
  - OUTPUT: "Have a good day."

Supported conditions:

- >
- <
- =
- !=

Supported column types:

- INT – Must only contain numeric values and optionally a negative sign.
- FLOAT – Must only contain numeric values and optionally a negative sign and/or a decimal point.
- STRING – No restrictions on contents.
- BLOB – Must be sent as base64 encoded data.

4.3.7 AccountingWizard

4.3.7.1 Description

Accounting Wizard is an accounting support service. The wizard simplifies the creation of a purchase-statement that accountants use for writing checks. The wizard records time-sheet hours and item purchases for a number of projects, and can produce a report on the hours charged and items purchased that occur in the current billing cycle.

There are four vulnerabilities embedded within the wizard:

- **Time Complexity**: Crafted employee input to the manager-only components of the wizard can force the application into a slower processing mode resulting in denial-of-service for managers.
- **Space Complexity**: Unauthenticated users can force the wizard into a more verbose logging mode resulting in denial-of-service for authenticated users.
- **Time Side-Channel**: Crafted employee input can leak sensitive **expenditure data**. The time interval expended for processing reveals expenditure data.
- **Space Side-Channel**: Crafted employee input can leak sensitive **purchase data**.

4.3.7.1.1 Time Complexity

Accounting wizard uses a bi-modal solver to work out the set of items that should be settled during the billing cycle. All hours must be paid out for each billing cycle, but items may be deferred to later cycles. The input to this algorithm is the collection of outstanding items and a project-internal expenditure limit (less the cost of paying the employees). The assumption, made by the programmers, is that the collection of outstanding items is unlikely to breach the expenditure limit. Once this assumption is broken, the application enters a complex processing stage that will attempt to quickly find a near-optimal settlement, but this side of the algorithm runs in exponential time when the unsettled items are made up of a single high-cost outlier and a number of low-cost items.

4.3.7.1.2 Space Complexity

The wizard contains a logging mechanism that allows administrators to perform basic audits and look for problems as they occur, however, the mechanism switches to TRACE level logging if a specific exception occurs. This exception is thrown in an unauthenticated route allowing anyone to force the server to begin consuming disk space. To make matters worse, the logging mechanism isn't configured to rotate logs or erase old messages. Without rotating logs or a method to limit the logging capacity of the application, the wizard will quickly run the server out of disk space.

4.3.7.1.3 Time Side-Channel

For the purposes of this side-channel, the secret is the project expenditure limit. When an employee submits hours to the wizard, the wizard checks the hours to make sure they don't exceed the budget before running a complicated task. The time difference caused by the check allows a malicious employee to guess the project expenditure limit.

4.3.7.1.4 Space Side-Channel

For the purposes of this side-channel, the secret is the number of items and employees currently expensed on any project. There is a database, stored on disk, of all of the projects, submitted hours, and items. A determined attacker with access to the file sizes and the ability to add items and submit hours will be able to figure out the secret. This side-channel occurs in the same way no matter which secret is targeted.

As an item is purchased, the wizard adds an identifier to the on-disk project file and creates a new item file with details about the purchase. The net effect is a single file will increase in size and a new file will be created, which will leak the location of the project file and the number of "children" of the project. The children include both the items and the hours that are charged to the project, so the leak only exists with 100% accuracy at the beginning of a billing cycle when there are no hours submitted.

4.3.7.2 Software Design

4.3.7.2.1 Time Complexity

In order to solve the problem of maximizing the usage of an expenditure limit when expensing required and deferrable expenses, this tool implements a two-stage calculation that looks like a near-optimal knapsack solver. This mechanism lives in the class BudgetSolver.

4.3.7.2.2 Stage One

1. Make sure that the number of charged hours will fit under the expenditure limit.
    a. Throw an exception if there are hours that cannot be paid for during this cycle.
2. Make sure the solution isn't trivial, such as is the case when all hours and items are affordable.
    a. Upon trivial solution, return every expense to the manager who requested the report.
3. Work out what the *adjusted expenditure limit (AEI)* is, i.e. the limit less the cost of spent hours.

4.3.7.2.3 Stage Two

1. Order the items in descending order.
2. Select lowest cost items, from the right, until the sum of the costs breaches the AEI.
    a. Finds a sub-optimal solution fast, but introduces error by ignoring expensive items.
3. Generate combinations of the selected items until the cost doesn't breach the AEI.
4. Return the collection of items whose expense receded below the AEI to the requesting manager.

4.3.7.2.4 Combinations

In order to "select" items (see Stage Two, Part 2), a Boolean vector, with a value for each position in the ordered items collection, is used to indicate which items are "selected". This can be interpreted as a set selector, a binary string, and an integer. The vector [true, false, false, true] is both the integer 9 and a set selector for selecting the first and last element of the items collection.

Generating combinations of items in a collection can be seen as incrementing (or decrementing) an integer of the same number of bits as there are elements in your collection. There are no repetitions and no fixed combination widths, so we don't have to worry about correcting for those issues. The selector, created in the second step of stage two, always matches the regular expression: "0+1+". During the generation phase, the vector will be decremexiented as if it represented an integer. For example dec(7) = 6 = dec([1, 1, 1]) = [1, 1, 0].

Summing the currently selected set is a costly operation, so accounting wizard minimizes this cost by utilizing a simple memoization strategy that groups chunks of the collection and stores the sum produced by each configuration of the selector so that the sum is not repeated for that configuration.

4.3.7.2.5 Space Complexity

Log messages from the challenge contain basic play-by-plays about what is occurring in the system at any given point in time. In the state the system is in when the application starts, there are relatively few messages aside from a few per API call such as "authentication occurred" messages,

which occur when a user is authenticated, and “database read” messages, which occur when a user reads information from the database. During any normal execution, an unauthenticated user can attempt to change the locale of the system, which mimics unprotected and unmaintained feature bloat that occurs when developers overlook the security impact of ‘dead-code’. If the system encounters a locale change request, but doesn’t have a locale to complete the switch, then the system will throw an exception that the user will see and change the logging message level to TRACE.

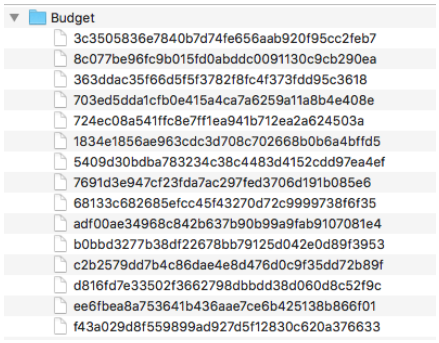
This vulnerability is a classic example of unexpected user input that causes a denial of service. To make matters worse, the attacker can actually change the language the system is logging in to one that is more verbose than English before triggering the TRACE level logging. The route that can trigger this non-local vulnerability is the “/lang/:tag” route (shown above in *Challenge Provided Interactions)*. The vulnerability code is any log producer that logs at a level lower than the default level of INFO, and the exception behind the switch is called *LocaleNotSupportedException*.

4.3.7.2.6 Time Side-Channel

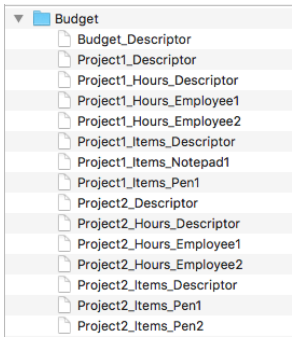
A basic guard exists in the code that throws an exception if an employee tries to submit a number of hours that totals greater than the project expenditure. Because this is so naïve, the attacker can submit a set of hours that far exceeds the expenditure-limit, which will take less time than attempting to submit a reasonable set of hours. This requires some item purchases to consume enough time to notice the difference, so an attacker might order nine low-cost pens and one high-cost pen to consume some time. This is not a blind side-channel because an exception is returned to the user, but the server will not give away exactly what the concern is. In this case, the server will inform the user that their submission is concerning and they need a manager.

4.3.7.2.7 Space Side-Channel

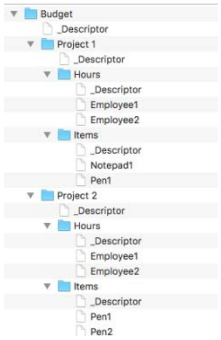
The budget, projects, items, and hours are serialized into a structure *like* this first image in Figure 35. The more an attacker plays with the program state and observes the file sizes, the more it can be pictorially restructured as the images progress to the right. Both the center and right images leak sensitive data.



Extracted view as seen by an attacker



View if you knew the names of the objects



View if you grouped the objects by name

Figure 35. Structure and Re-structure of Object View

The structure of each of these files begins with a serialization ID, similar to that of the Java Development Kit serializable interface, followed by object-unique content serialized as simply as possible. Children are serialized as an array of filename strings matching the filename of the child objects. The serialization code lives in BaseObjectMarshaller and the database code lives in FileStore.

4.3.7.2.8 Challenge Design

4.3.7.2.9 Architecture

The wizard is a basic API using the Java 8 *Spark framework* for all of the networking code. The API endpoints, are supported by a custom *role based access control* framework and a number of manager objects, which attach users' sessions to the code that their requests want to execute. Manager objects provide the view methods with access to the database, which is called FileStore in the code-base, and the budget management component, BudgetSolver. Figure 36 depicts how requests flow through the codebase.

The challenge is a JSON web API so inputs and outputs should be valid HTTP and JSON. If you don't provide valid HTTP or JSON, where applicable, to valid challenge inputs, then you will just get regular HTTP malformed request responses.

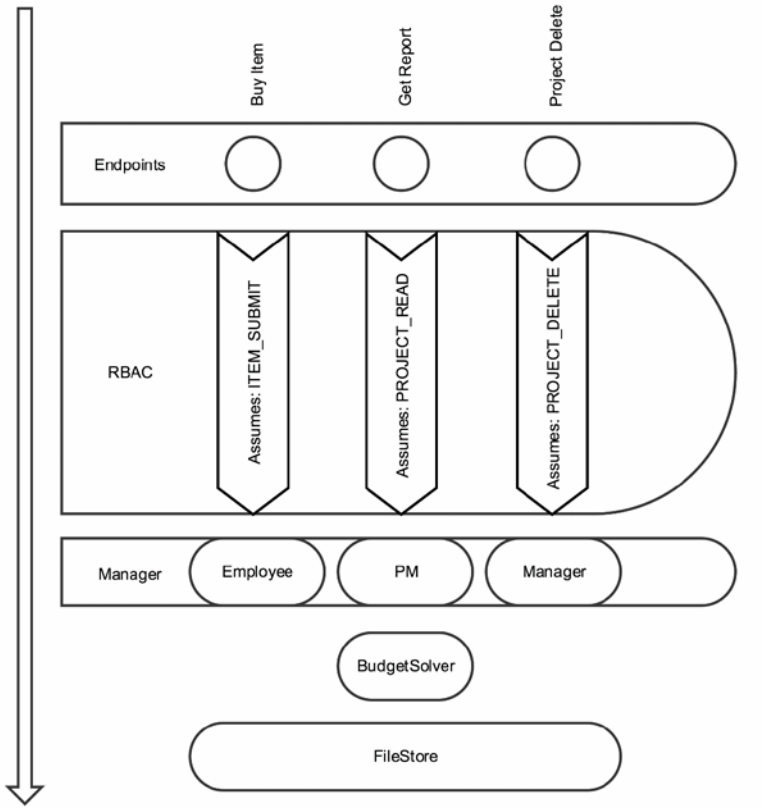


Figure 36. How Requests Flow through the Codebase

At the core of the security and role-based access control system, there is a **predefined** employee set. Each employee has a type and each type comes with a collection of assumable roles (Table 17). This gives the system the information it needs to provide user sessions with access to the correct manager objects.

Table 17. Predefined Employees and Their Corresponding Types and Roles

Employee	Type	Assumable Roles
Marie Callender	Employee	TimeSubmit, ItemSubmit
Ronald McDonald	Employee	TimeSubmit, ItemSubmit
Dairy Queen	Project Manager	Employee + TimeRead, ItemRead, ProjectSubmit, ProjectRead
Burger King	Manager	Project Manager + ProjectDelete, ProjectCreate

Manager objects are objects that contain segregated code that only a particular level of employee may have access to. For example, when submitting an item purchase, every employee session will have an attached EmployeeManager with methods that allow them to charge a project with purchases. This item purchase code only exists on the EmployeeManager object, which begins by checking to make sure that the currently logged in user has the ability to assume the role of item purchaser (ItemSubmit). This double-edged approach prevents users from escalating their privileges after their session has started. If an employee was somehow able to impersonate a program manager by changing their employee type, then they would still be missing the required ProgramManagerManager object and thus their impersonation would be fruitless. This particular construct is also an attempt to confuse tools into believing that all authenticated routes might have access to manager code that the security framework will never give them. These managers contain code that interacts with the FileStore and the BudgetSolver. BudgetSolver implements the vulnerability.

FileStore objects are forced into the hierarchy in Figure 37 by the challenge. On disk, there may only be one budget object, and violating this constraint will cause a fatal exception to be thrown.

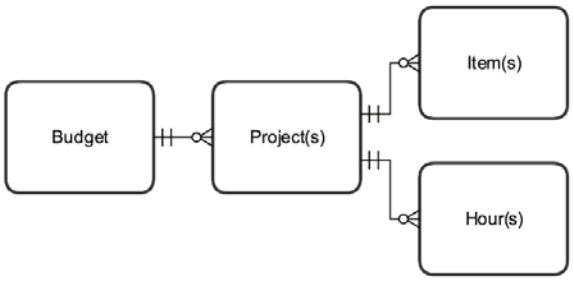


Figure 37. Challenge Database Hierarchy

Figure 38 depicts the programmatic relationship between the serialized structures. The only member of BaseObject that is always serialized is the set named *children*, otherwise members are only serialized when a subclass also declares a member of the same name and type.

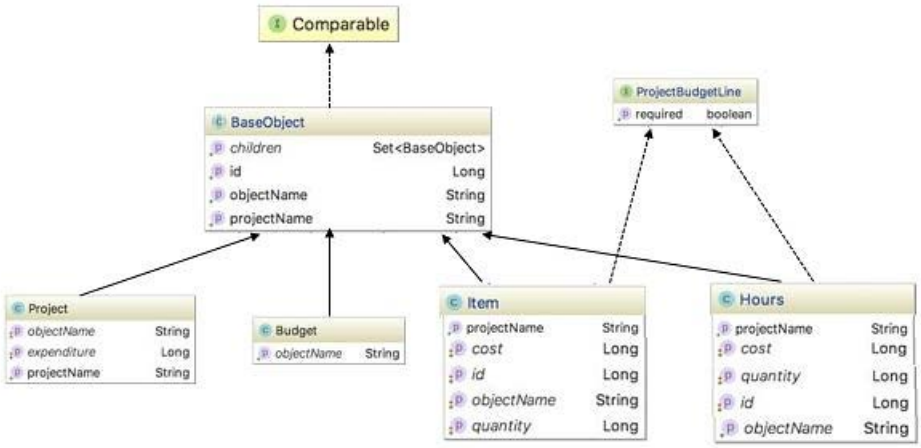


Figure 38. The Programmatic Relationship between Database Objects

The BaseObjectMarshaller orders the serialized members on disk via rules for BaseObject members and an annotation parameter on subclass members. All of integer-like types are stored in big-endian format and strings are stored as an integer length followed by the string. An on-disk object follows this order:

- Long – Serial ID
- String – Object name
- Object… – Subclass members...
- Integer – Number of children
- String… – Child SHA-1

The subclass members are serialized in the same way as BaseObject members, but their order is defined by an index parameter to the Field annotation (i.e. @Field(index=x) private Long member.)

4.3.8 Stegosaurus

4.3.8.1 Description

Stegosaurus is a messaging service that uses steganographic techniques to hide messages in random images to hide message presence. The sender can log into the messaging service, choose an image or let the server find a random image, and send the message to be embedded into an image over an encrypted connection to the server. The embedded message is supposed to be encrypted using a client-provided Rivest–Shamir–Adleman (RSA) public key, however, due to implementer's error, the RSA public key is only used for walk-finding logic. Once the message is embedded, the server lets the sender to choose a method over which the image needs to be delivered.

The Stegosaurus challenge exhibits **a timing side channel vulnerability** that leaks the message the client is trying to hide in an image.

Stegosaurus gets the message from the client over a SSL protocol, generates a walk over image subpixels using the client-provided public key RSA, and embeds it into either a provided or a randomly chosen image. Since the embedding process is relatively slow, the client polls the server every 5 seconds to see if the server is still working and that it is n% done. To alleviate any timing side-channels, the heartbeat packet is sent from a second thread on the server, however, due to a racing condition, the heartbeat thread sends a packet only once each bit is embedded.

The steganographic process uses the provided RSA public key to calculate the next bit offset similar to how RSA generates cyphertext. In this case, the “cyphertext” simply determines the location at which the next plaintext bit is embedded; the plaintext bits are retained within the image. On its own, this calculation is not vulnerable to side-channel leaks, however, offsets are dependent on each message bit. The offset seeking implementation is side-channel vulnerable – since the RSA-computed values are much larger than the image size, modulus needs to be done to calculate the actual offset. However, a subtraction in a loop was used. Larger values will result in longer execution of the offset-calculating loop. Since offsets are generated per 1 bit of the plaintext message, RSA-computed values for a high bit will be much larger than for a low bit, resulting in a noticeable time difference between heartbeats depending on the value of the plaintext bit.

4.3.8.2 Software Design

4.3.8.2.1 Architecture

Stegosaurus uses NanoHTTPD library to present its interface through any web-browser. This library will be included in the challenge app. Figure 39 shows challenge class diagram.

The web-server (*Stegosaurus*) is responsible for establishing the SSL communication with the user's web-browser, sending the data-input page to the user, updating the steganographic process' progress bar, and sending the final image to the user. *RequestHandler* decides if the user wants to request the data-input page or of the user is sending the data to be used by *Steganographer*. If the latter, the request-processor *RequestHandler* will pass the data to *Steganographer* and it will check if the message is at least 1-character long, if the public key is in a proper format, and if the image is sufficiently large to have the given message embedded. If the checks pass, *RequestHandler* will spawn a *HeartbeatThread* to keep tally on *Steganographer*'s progress, otherwise it will notify the user of the corresponding error. *Steganographer* takes the user data and embeds the encrypted message into an image, it stores the current and total bits to be embedded in variables visible to the heart-beat thread. The side-channel is originated in *Steganographer*. *HeartbeatThread* polls *Steganographer*'s progress and decides when it's the time to update user's progress bar through the web-server. Due to a racing condition with *Steganographer*'s thread, the *HeartbeatThread* lets the side-channel information through. Refer to the vulnerability pseudo-code for racing conditions outline.

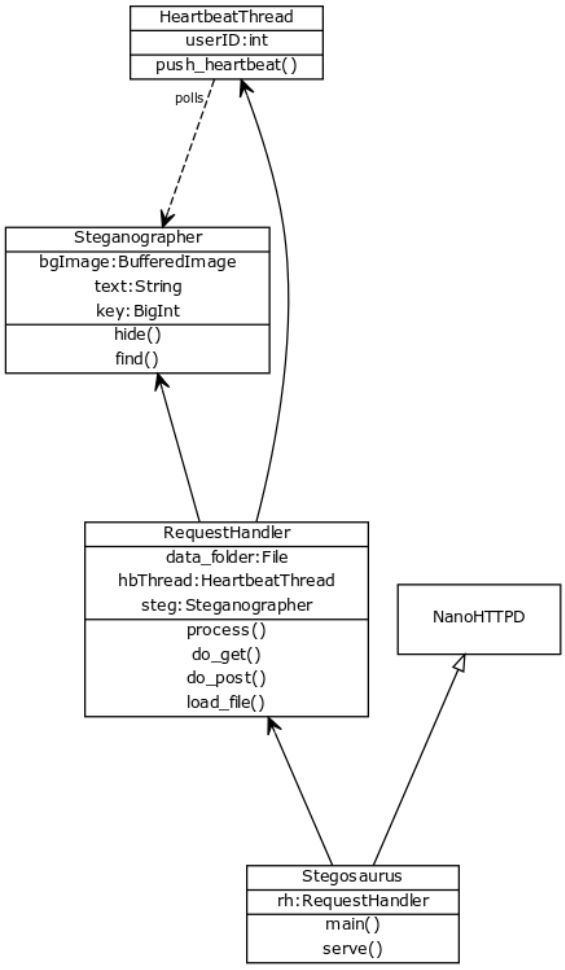


Figure 39. Stegosaurus Class Diagram

4.3.8.2.2 Inputs and Outputs

The output of Stegosaurus are HTTPS packets. The data-input page is a dynamically generated page with java-script to enable Asynchronous JavaScript and XML (AJAX) -based progress-bar updates. Dynamic generation is used to let the user know the maximum embeddable message length and whether there were any errors. The progress bar update is a JSON structure wrapped in an HTTPS response. The final image is a PNG image wrapped in an HTTPS response.

The input from the user is a possible image file path into which the message will be embedded. If the file path is empty, Stegosaurus will choose a random image from its internal library. The public key is hexadecimal encoded RSA key. The embeddable message is any set of printable characters, whose length is larger than 1 and smaller or equals to the maximum embeddable length for a given image or 512, whichever is smaller. The user's web-browser will take these inputs and form an HTTPS request that will be sent to Stegosaurus.

Inputs from the user are listed in Table 18.

Table 18. Stegosaurus User Input

HTTPs Method	Path	Description
GET,POST	/, /index.html	Main interface for user to provide possible target image, message, and key
POST	/process.html	Posts data from the index.html (image, text, key) to the backend. Backend starts processing the data. Returned process.html will already contain JavaScript code to perform AJAX calls to /status.html and to draw a progress bar based on the data returned from /status.html The image is in any format that javax.Imageio.IomageIO can open. This includes JPEG and PNG images. The key is a hexadecimal string value without the '0x' prefix. The string must be of 32 characters long, and the integer value of the key must be at least 122 bits long (last character of the string must be 1 or greater) The message is any printable string smaller than 512 characters.
GET	/status.html	AJAX call from process.html by the user to receive status of the steganographic process. The response to these packets is the one carrying the timing side-channel.

Outputs from the server are listed in Table 19.

Table 19. Stegosaurus Server Output

HTTPs Method	Path	Description
GET,POST	/, /index.html, /process.html, /status.html	Proper pages or data structures as expected by the user
Everything else	Everything else	Error 404: Page not found.

AJAX calls to status.html will return an HTTPS packet containing the amount of bits currently embedded in the image. This value is used to fill the progress bar of the client's page.

The collection of random images and html pages that the server can utilize will be placed in the data/ folder that will be located in the same folder as the challenge *.jar file. When needed, the server will randomly choose 1 image that is large enough to contain the data, and add noise and decorations to the image to further obfuscate the hidden message. The image files will be located in the data/images/ folder.

The content of the web pages aside from the dynamically generated content, will be stored in data/www/ folder. The HTML wiles will contain the proper sequences of characters to be replaced at runtime to include the dynamically generated data.

4.3.8.2.3 Vulnerability Design

4.3.8.2.4 Related Class Roadmap

Classes related to the implementation of the vulnerability:

- *Steganographer* – class that includes the actual side-channel leak and the algorithm pseudocode mentioned below.

- Class member message_bit – bit offset that's currently being embedded. *HeartbeatThread* uses this information to decide when to release the heart-beat packet.
- Class member cyphertext – takes a value of n, when embedded bit is 0, and n+k when embedded bit is 1, where n, k are large integers. Specifically, n correlates to the size of the public key modulus, and k correlates to the public key modulus raised to the public key power.

- *HeartbeatThread* – potential red-herring class, as it is meant to remove the timing information, however due to a racing condition, the timing information is preserved. Proper implementation would not have the 'if' statement in the run() function and have a longer sleep interval.

4.3.8.2.5 Algorithm Pseudocode

```
function HeartbeatThread.run(steganographer):
last_update = 0;
while (steganographer.is_working):
if (steganographer.message_bit > last_update):
sendHeartbeatPacket(steganographer.message_bit * 100 /
steganographer.message.bit_length)
last_update = steganographer.message_bit
Thread.sleepms(1)

function Steganographer.hide(image, message, public_key):
data_offset = 0
data_offsets_used = boolean[max_offset+1]
set all data_offsets_used to false
heartbeat_thread.progress = 0
heartbeat_thread.max_progress = message.bit_length
for message_bit in message.bits:
// message_bit is either 1 or 0
// clear out image bit
image[data_offset+i] &= 0xFE
// write message bit
image[data_offset+i] |= message_bit
// wrong 'RSA' implementation
cypher = (message_bit*public_key.remainder)^public_key.power +
public_key.remainder

while cypher > 0:
data_offset += 1
cypher -= 1
if data_offset > image.data_size:
data_offset = 0
while data_offsets_used [data_offset] {
data_offset++; // up to 3 * imageX * imageY iterations
if data_offset >= max_offset
data_offset = 0
}
data_offsets_used[data_offset] = true;
```

4.3.9 StuffTracker

4.3.9.1 Description

StuffTracker is a network service that allows users to upload, retrieve, and search product inventory records. To support these capabilities, StuffTracker provides a backend that records inventories in a XML-based markup language called the Stuff Description Language (SDL). Users write inventory records in SDL and upload them to the service via a RESTful[1] interface. When an inventory record is uploaded to StuffTracker, the backend parses the document to identify terms that are similar to those in other inventory records. To facilitate this, elements are extracted from each uploaded document and inserted into the backend's storage engine. This capability enables users to deconflict the inventory by finding similarities among textual product descriptions and the SDL schema tags representing product categories. Since inventories can grow large, StuffTracker also implements a specially crafted compression scheme that allows SDL inventory documents to be both uploaded and searched in a compressed form.

StuffTracker contains a **space complexity** vulnerability. The space vulnerability in the StuffTracker application is the result of subtle flaws in a custom compression algorithm. The algorithm is designed specifically for use with the SDL format, and works by replacing SDL's potentially verbose schema strings with smaller numerical representations (called compression codes) that uniquely map to the language's schema strings while taking up fewer bytes. The flaws manifest in the input guards that check the correctness of the compression codes and which handle how the codes get replaced with the SDL strings that they represent during decompression.

A pair of flaws have been introduced in the input guard which must both be exploited by the attacker to trigger the vulnerability. One of the flaws is that the guard fails to catch tags that are not part of the SDL format but affect how it processes inputs, and another where a compression code can be maliciously prefixed with additional unnecessary bytes and yet still can successfully pass through the guard. In order to pass through the guard, the malicious compression code must be prefixed with bytes representing a zero so that it will be recognized as an integer (by Integer.parseInt) with the same value as some valid non-prefixed compression code. When this occurs, the code will be treated as valid even though the overall length of the bytes representing it has increased by one. When such a missized code is passed to the serializer, the delimiter marking the end of a dictionary entry will get overwritten because the write position in the serialized dictionary buffer will not be incremented correctly to account for the additional byte in the code.

With the delimiter(s) overwritten, the program will be in a faulty state since the resulting buffer contents will later be used by the decompression algorithm. This can result in a space explosion that is orders of magnitude larger than the input needed to trigger it because the previously mentioned flaws allow an attacker to make the size of each dictionary entry arbitrary large, within the limits of the provided input budget. The space explosion occurs when numerical references to these malformed dictionary entries are replaced with their arbitrarily large text during decompression.

To make the program more difficult to analyze, we will structure the vulnerability so that the attacker will have to submit two different types of requests, the upload request and then the search request. The vulnerability can be considered 'non-local' (i.e., distributed throughout the challenge

program's code) since the data flows that connect the requests' code sections are separated by an interpreted query language used for search requests. This will require the attacker to formulate dynamically interpreted queries that trigger the attack, which we expect will make their taint analysis more difficult.

4.3.9.2 Software Design

The following is a description of the classes related to the vulnerability. The corresponding class diagram is shown in Figure 40.

StuffTrackerService: The StuffTrackerService class listens on the server port. This class is responsible for monitoring for all incoming requests and forwarding those requests to instances of the SDLParser or the SDLDataStore.

SDLDataStore: coordinates actions between the DocIndexer, DocStore, and GlobalDictionary.

SDLParser/ItemListHandler: These classes parse and validate SDL files. As such, they are responsible for extracting tags and associated values. The ItemListHandler also constructs a local dictionary for every SDL doc it receives and prepares a compressed version of the SDL doc for storage; as such, it the compression handler object for SDL.

DocIndexer: This class maintains indexes of documents and processes queries that search those indexes. It allows users to search for a document by tag values.

IndexDecompressionHandler: Decompresses indexed SDL contents during search on behalf of DocIndexer.

DocStore: This class stores SDL documents alongside their local dictionary file.

StoredSDLDecompressionHandler: Decompresses stored SDL documents on behalf of DocStore.

GlobalDictionary: Stores an index of actual tag values to integer ids. The integers' ids are the compression codes used by user when uploading compressed documents.

SDLDoc: Holds a reference to a particular document and has fields to hold its contents and its local dictionary.

Explanation of Local vs Global Dictionary:

The global dictionary holds information about all tags and their frequency. It has a mapping of an integer ID to the value it represents. There is one global dictionary for all documents. A user can compress any document they upload to the server using terms from the global dictionary.

The local dictionaries exist on a per document basis. After a document is uploaded, the server creates a local dictionary that only contains codes for the tags in that particular document.

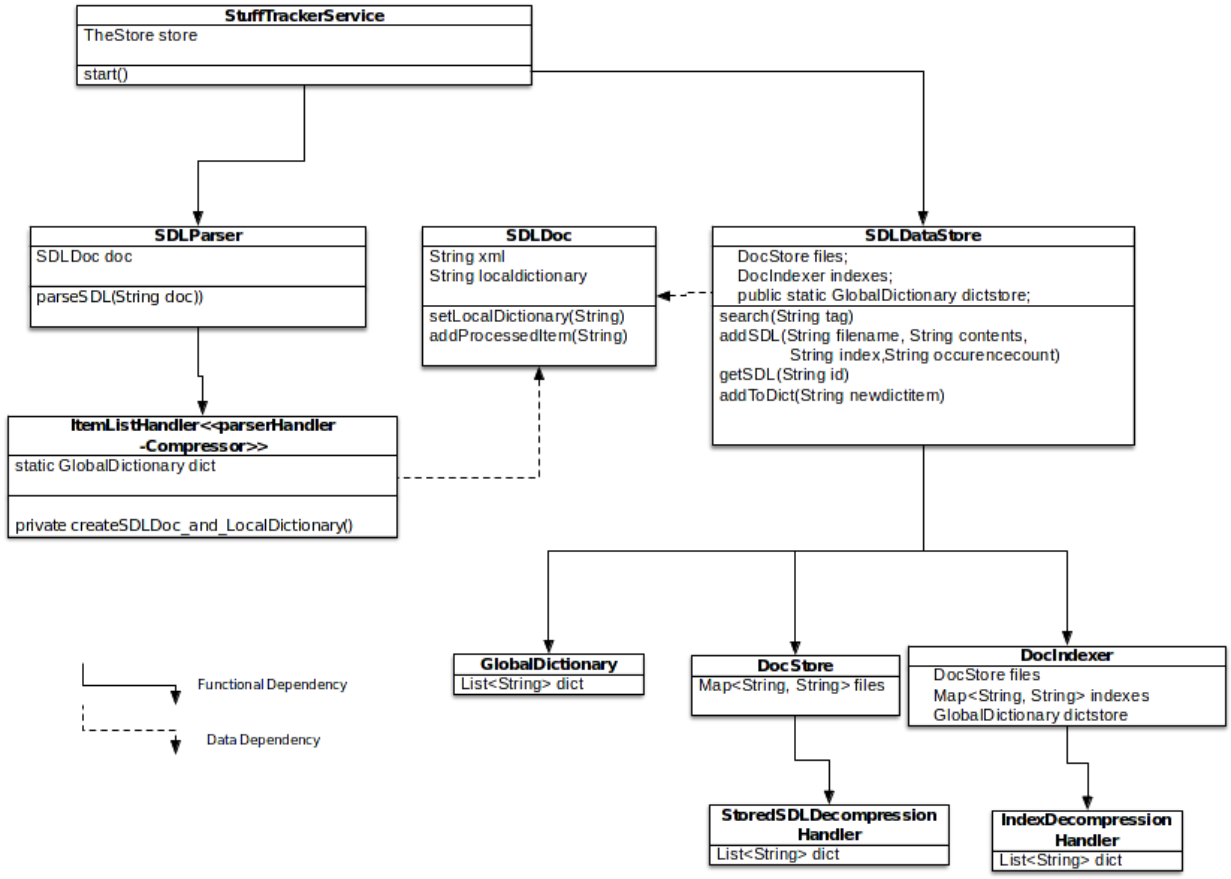


Figure 40. Class Diagram for StuffTracker Application

Vulnerability Code Flow

The following step descriptions correspond to the numbered steps in the Vulnerability Sequence Diagram in Figure 41. These are the steps required to trigger the StuffTracker space complexity vulnerability.

Step 1: Attacker Uploads Malformed SDL Document

Attacker makes a malicious request by supplying a text document, written in the Stuff Description Language (SDL) XML format, that contains additional, unnecessary characters which prefix valid tags from the SDL. These additional characters bypass guards checked during parsing. Their inclusion results in a situation whereby an improperly formatted document is stored on the server in a compressed form that, when uncompressed during a subsequent step, will trigger the space vulnerability.

Step 2: StuffTrackerServlet Parses Uploaded Document

The SDLParser class parses the uploaded document by extracting the XML tags, their attributes, and the contents located between these tags. The tags describe the types of items whose records

are to be uploaded into the inventory system. These tags may be compressed or uncompressed. The following is an example of an uncompressed entry in a SDL document:

```
<list><item id='Imperial Destroyer:Destroyer:1.2:Empire Inc.'>Darth
Vader</item ></list>
```

In this entry the 'list' tag is the topmost level tag that contains all of the 'item' tags. Each 'item' tag represents a single entry into the inventory system. An 'item' tag must have both an 'id' attribute, which describes the type and version of the item, and text between its tag brackets that describes who the owner of the item is. The text in the 'id' tag must correspond to an existing value located in the StuffTracker Global Dictionary.

Step 3: Check for Compressed Tags

For each tag, the parser must check if the tag's id value has been replaced with a compressed value. A compressed value is a numerical value that substitutes the string located in the 'id' attribute when the 'c' attribute is also included in the tag, as shown below. Each numerical value acts as an index into a global dictionary represented by the GlobalDictionary class. This dictionary is a one-to-one mapping between each numerical value and the tag it represents. On the wire, the numerical value is represented as a byte for each character of the particular numerical code. That is, 1-9 would require one byte; 10-99 two bytes; 100-999 three bytes; and so on. The following is a depiction of how the example from Step 2 would appear when compressed using numerical values:

```
<list><item id='1' c='x'>Darth Vader</item ></list>
```

The Parser is the first location where the vulnerability manifests. The parser has a flaw that allows illegitimate/unexpected tags to still be processed without error, if the illegitimate tag is located inside a legitimate one. The following is an example of an input that will pass the parser and trigger a part of the vulnerability:

```
<list><item id='01' c='x'>A<x></x>A <x></x>A<x></x>A<x></x>A</item></list>
```

Note that the mixing of the illegitimate tags, *<x></x>,* tags with a 'A' character. The 'A's correspond to the name of the item owner in a properly formatted tag. In this case, however, there are unexpected illegitimate tags between the 'A' instances that cause each 'A' to be treated as a separate name. The result is that the parser processes the legitimate tag repeatedly, once for each illegitimate tag added. Each time the parser does this, it passes another instance of the legitimate tag to further steps.

Step 4: Decode Compressed Items

When Step 3 finds a compressed value, the value is decompressed by the parser. The decompressor is part of the ItemListHander class and is responsible for decoding the compressed value by converting it from its Java integer form to a text string. Then the decompression routines work by looking up the integers in the GlobalDictionary to retrieve the actual text string they represents.

Step 5: Create a Local Dictionary (prepare document for storage)

The string value of each tag retrieved from the global dictionary is now added to a new local dictionary file that is specific to the uploaded SDL document. The local dictionary has a simple format, listing each tag used in the SDL document in order of frequency of occurrence. The most

common tag appears first, the second item second, and so on, and each entry is separated by a delimiter. This local dictionary allows the specific SDL file to be compressed based only on its own contents instead of by the GlobalDictionary, which may contain hundreds or even thousands of tags. The function that implements this step is vulnerable because, when data is passed to this function that has a tag prefixed with a '0', the delimiter character that separates the tag values is overwritten by that '0'.

The following is what a normal dictionary file should look like after this step, where the '|' represents the delimiter (in the actual code the delimiter is a null character):

```
sometag|someothertag|somethirdtag
```

The following is what a vulnerable dictionary file missing its delimiters looks like after this step:

```
sometag0someothertag0somethirdtag
```

The lack of delimiters will result in a space explosion in a later step. The zeros above represent ASCII 0's, not the null character which is the expected delimiter.

Step 6: Finalize and Store SDL Document

In this step, the server writes out the SDL file and the local dictionary file. The normal file has its tags replaced based on their index value in the local dictionary file.

Step 7: Index Document

The SDL file is indexed to allow it to be found based on its tag contents. In the example input file in step 2 above, this would be any character in the string ' Imperial Destroyer:Destroyer:1.2:Empire Inc.'

Step 8: Search for SDL Documents

This step allows users to find SDL documents using their indexed parameters. The results are to be sent back to the user uncompressed, meaning that the dictionary entries replace their corresponding indexes in the SDL file before being sent out.

Step 9: Decompress SDL Document with Local Dictionary

The vulnerability ultimately manifests here when an attempt is made to decompress a SDL file with a dictionary that has no delimiters. This potentially results in arbitrarily large dictionary contents being serialized out for each entry.

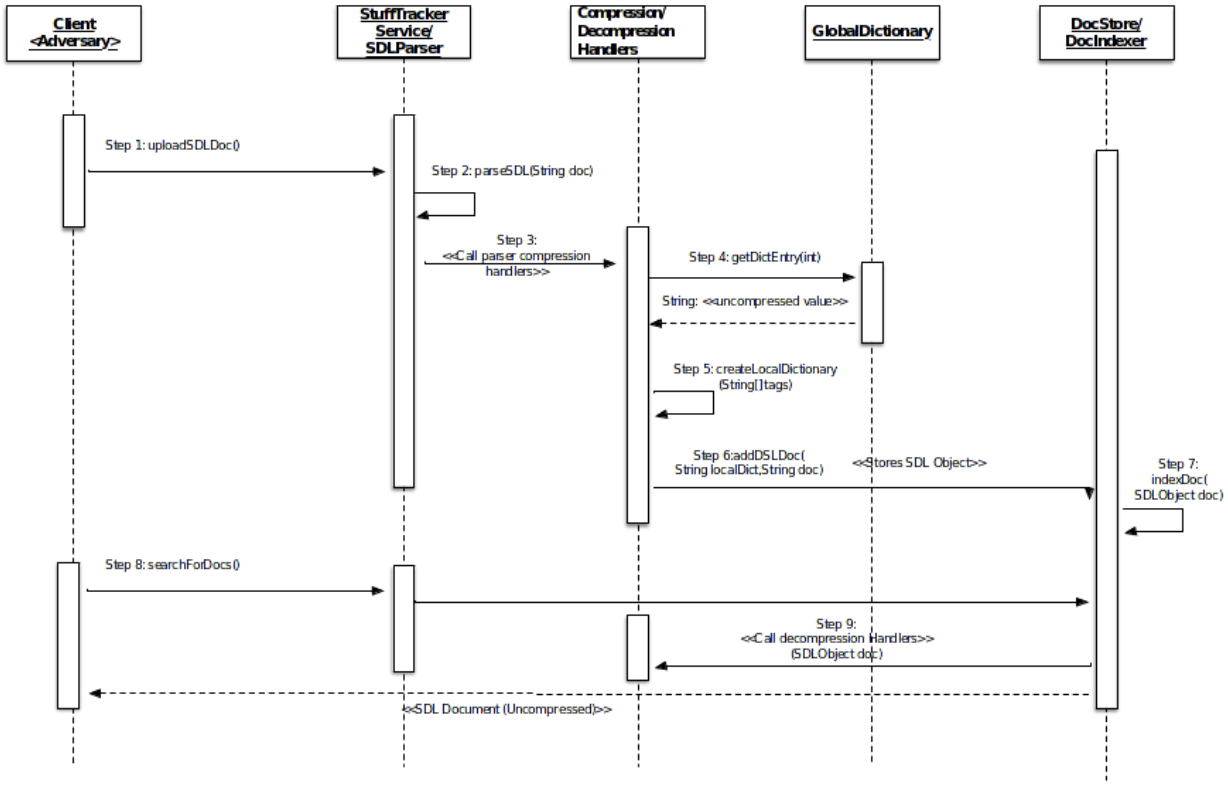


Figure 41. Vulnerability Sequence Diagram

4.3.9.2.1 Vulnerability Code

As described above, the vulnerability implementation extends over three parts: the parser, the local compressor, and the decompressor. The parser has flaws in how it recognizes data entry instances that have illegitimate/unexpected tags that cause it to reprocess legitimate tags repeatedly (the code is located in the HandleItemList class with comments). The Local Compressor has flaws in how it serializes the local compression dictionary when the parser presents it with incorrect information that causes it to overwrite a delimiting character, as described previously (vulnerable code below). Finally, the decompressor creates extremely large files in memory when the delimiter is missing and proceeded by large amounts of data (pseudocode below, actual code in IndexStore.java search() function).

The Vulnerable Local Compressor code (responsible for overwriting delimiters):

```
//Class variable that holds the final local dictionary
StringBuffer compstr = new StringBuffer(1);
//Class variable that tracks the position in the local dictionary for
appending data
static int compstrpos = 0;
//The getLocalCompressionStrForFile function will create a dictionary file
that is a list of items separated by a delimiter in normal case. When the
tags are prefixed by a 0, the delimiter will be overwritten by this logic
```

```
public String getLocalCompressionStrForFile(String loctobereplaced, String
index, String tag) {
compstr.setLength(compstr.capacity()+tag.length() + 1);
int indexOf = loctobereplaced.indexOf(index);
String substring = loctobereplaced.substring(1, indexOf + index.length());
String replace = substring.replace(index, tag);
compstr.replace(compstrpos, compstrpos+replace.length(), replace);
compstrpos += tag.length() + 1;
compstr.trimToSize();
return compstr.toString();
}
```

The Vulnerable Local Decompressor code:

```
public String search(String tag){
//Loop over results and decompress them, when decompressing results without
delimiter, space explosion ensues
StringBuffer results = new StringBuffer();
List<SDLDocRef> res = tagindex.get(tag);
Iterator<SDLDocRef> it = res.iterator();
while(it.hasNext()){
SDLDocRef ref = it.next();
results.append(ref.decompress());
}
return null;
}
```

4.3.9.2.2 Inputs and Outputs

StuffTracker services several types of user requests, including: 1) an inventory report upload request, and 2) an inventory download request and 3) an inventory search request. Requests are made to the server using HTTP.

Inventory Upload Request: Sends an XML document containing inventory items to be stored on StuffTracker server. The usage is a call to the url 'http://server_address:4567/stuff' with XML data uploaded via POST.

Example usage with curl:

```
curl -X POST -H "Content-Type: application/xml" -d
"<list><item id='1' c='t'>owner</item ></list>"
http://server_address:4567/stuff
```

Possible Result #1: "OK:{docid number}". {docid number} is the numerical, long ID value of the document just uploaded.

Possible Result #2: "Error: Java Exception internal error"

Possible Result #3: "Error: An id does not exist in the dictionary"

Inventory Get Request: Requests an XML document containing inventory items that is stored on StuffTracker server. Usage is a call to 'http://server_address:4567/stuff' with an docid parameter set on the url. The docid parameter should correspond to the numerical ID of type long that refers to an existing, previously uploaded inventory document on the server. Request made via HTTP GET.

Example usage with curl:

```
curl http://server_address:4567/getsdl?docid=123456789
```

Possible Result #1: "OK:{xml inventory response}". { xml inventory response} is an xml formatted inventory document.

Possible Result #2: "Error: Java Exception internal error"

Possible Result #3: "Error: A document with this id does not exist"

Possible Result #4: "Error: "Unknown error"

Inventory Search Request: Requests an XML document containing instances of inventory items that are stored on StuffTracker server. Usage is a call to 'http://server_address:4567/search' with a search term parameter set on the url. The search term is a text field that is used to search the ID fields of item stored on the server.

In this example inventory document: *<list><item id='Imperial Destroyer:Destroyer:1.2:Empire Inc.'>Darth Vader</item ></list>,* the search term is used to search the text:'Imperial Destroyer:Destroyer:1.2:Empire Inc'. If a compressed version of id is submitted for a particular file, it searches the corresponding uncompressed text string.

Request made via HTTP GET. Example usage with curl:

```
curl http://server_address:4567/stuff?term=Imperial
```

Possible Result #1: "OK:{xml resultset response}". { xml resultset response} is an xml formatted result document.

Possible Result #2: "Error: {message from Java exception object}"

Possible Result #3: "Error: Term size too large or too small

Dictionary Get Request: Gets a list of of types of stuff stored in the StuffTracker dictionary. Includes a numeric index and a string value for each entry in the list.

Request made via HTTP GET.

Example usage with curl:

```
curl http://server_address:4567/dict
```

Possible Result #1: "OK:{xml dictionary response file}". { xml inventory response} is an xml formatted inventory document.

Possible Result #2: "Error: Java Exception internal error"

4.4 Engagement 3 and 4 Challenge Programs

Engagement 4 was a take-home version for Blue-Teams of Engagement 3 challenges.

4.4.1 Matrixmultiply (linear_algebra_platform)

4.4.1.1 Description

The Large Scale Linear Algebra Platform (LSLAP) is a publically available service that allows for multi-core computation of common linear algebra operations. It is designed as a kind of cross between a remote MATLAB cluster, and a publicly available mathematics resource like Wolfram Alpha. The service allows users to offload large computation tasks such as matrix multiplications, decompositions, and transformations. These tasks will vary in complexity between linear and cubic in the size of the matrix. On the backend, the LSLAP uses a mixture of multi-core and single-core algorithms to compute solutions. The service is like Wolfram Alpha in the sense that it handles math queries from the public, but more like a MATLAB cluster in that it is an RPC service vice an interactive website.

The LSLAP operates as a publically available service for people interested in performing large scale computation. Computation jobs are handled by the service in a FIFO queue system. **A malicious user can submit a vulnerable computation job that will waste a significant amount of computation time, and slow down the tasks of other users.**

4.4.1.1.1 Vulnerability Description

The LSLAP will consist of several algorithms with varying time complexities. Some will be distributed, and some will be designed to run on a single core. The single core operations will be simpler linear or quadratic tasks that are non-vulnerable within our budgets. The distributed operations will be for more complex cubic time tasks like determinant calculation and matrix multiplication. The vulnerability will arise from using a parallelization scheme for matrix multiplication where the task breakdown is tied to properties of the input matrix.

The vulnerability exploits the coefficient for the cubic multiply operation, and the strength of the vulnerability depends on the number of cores available for distributed processing. Since the NUC has only two cores, the vulnerability causes the worst case performance to be double the time of the expected (average) case. With additional cores (say, if we were to network additional NUCs) we can increase the multiplier by one for each core. However, with only two cores, we still attain proof of concept.

To achieve this, we will use a parallelization scheme wherein for a square matrix with n rows each of the n^2 required dot product operations is treated as a sub task to the multiplication operation. We will batch those sub-tasks into M task groups of size K. The size K will be a complex function of the attacker controlled input matrix leading to the vulnerability. Each of the M task groups is handled by a thread worker using a very simplistic bulk synchronous parallel model of computation. We will provide a complicated heuristic, based on the input matrix, for deciding how to split up the task groups. Ideally each thread worker spends most of its time computing multiplication tasks and spends a minimal amount of time on communication. When K=(n^2/2), the tasks are evenly divided between both cores on the NUC. When K gets large, the task groups

are large meaning that not much parallelization occurs. In the worst case where K=n^2, all of the dot product tasks are performed by one worker, leading to the vulnerability.

The challenge for the Blue Teams will be to invert the math on the heuristic to determine whether it is possible to provide a matrix that induces the worst case (all tasks put on one core).

4.4.1.2 Software Design

The LSLAP application will be implemented as a web service built upon the NanoHTTPD framework. The user will make requests to the service via a HTTP POST request. The request will be parsed by the service, and handled either on a single core or in parallel by the service depending on which mathematical operation is performed. When the operation is complete, the result will be pushed back to the user through NanoHTTPD's response method.

4.4.1.2.1 User Interface

The service's main class creates a "LinearAlgebraService" which is an extension of the main NanoHTTPD class file. The LinearAlgebraService serves the parsing function for the LSLAP utilizing Google's GSON library to serialize and deserialize the JSON RPC requests and responses. Each of the linear algebra operations classes contain logic to manipulate matrices and perform the desired computation.

Pseudocode for the request handler is as follows:

```
class LinearAlgebraService extends NanoHTTPD {
      public Response serveRequest(WebRequest w)
      {
            OperationRequest    Request    =    GSON.deserialize(w.body,
      OperationRequest.class):
            assert(Request is valid)
            LinearAlgebraOperation operation = null;
            switch (Request.operation_type)
            {
                  case 1: // MultiplicationRequest
                        operation = new MultiplyOperation(request)
                  default:
                        operation   =   OtherOperation(request)   //   Non
                  vulnerable operations
            }
            OperationResult result = operation.compute();
            assert(result is valid)
            return GSON.serialize(result)
      }
}
```

4.4.1.2.2 I/O Format

Interaction with the service is through JSON RPC over HTTP. The structure of the JSON requests is as follows:

Request Object:

```
{"operation": <int>, "numberOfArguments": <int>, "args": [<arg 1>,…,<arg n>
]}
```

Response Object:

```
{"operation": <int>, “success”: <boolean>, “returnValue”: <string>}
```

Note: For both requests and responses, matrix and vector arguments are serialized to comma separated values (CSV) strings requiring at least 17 digits of decimal precision per entry (i.e, 17 ASCII characters including the decimal point). For example, a 4x4 identity matrix would be serialized as

```
1.000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000\n
0.000000000000000, 1.000000000000000, 0.000000000000000, 0.000000000000000\n
0.000000000000000, 0.000000000000000, 1.000000000000000, 0.000000000000000\n
0.000000000000000, 0.000000000000000, 0.000000000000000, 1.000000000000000\n
```

where the “\n” character just denotes a unix newline character.

Supported Requests:

1. Matrix Multiplication: Performs basic multiplication of 2 matrices
    a. Operation ID = 1
    b. Number of Arguments = 2
    c. Arg 1 = Square matrix written as CSV table
    d. Arg 2 = Square matrix of same size as arg1 written as CSV table
    e. **Response:** A CSV encoded square matrix containing the resulting product
2. Multi point Shortest Path: Given k vertices, compute the shortest path length from those k vertices to every other vertex in the matrix
    a. Operation ID = 2
    b. Number of Arguments = 2
    c. Arg1 = A weighted adjacency matrix with N nodes written as a CSV table
    d. Arg2 = A (1 x K) vector containing the K points to find shortest distances from in CSV
    e. **Response**: A CSV encoded (k x n) matrix containing the shortest paths from the K specified nodes to other vertices.
3. Graph Laplacian: Computes the laplacian of a graph given its adjacency matrix
    a. Operation ID = 3
    b. Number of Arguments = 1
    c. Arg1 = An adjacency matrix with N nodes
    d. **Response:** The laplacian of the input graph.
4. Minimum Spanning Tree: Computes the minimum spanning tree of a weighted non directed graph
    a. Operation ID = 4
    b. Number of Arguments = 1
    c. Arg1 = An adjacency matrix with N nodes in CSV
    d. **Response:** A (N x N) adjacency matrix defining the minimum spanning tree for G that preserves original indices.

4.4.1.2.3 Matrix Multiplication Component Architecture (Vulnerable)

Matrix Multiplication Requests passed to the linear algebra service are handled by the MultiplyOperation class. This class validates the arguments passed by the client, and prepares a parallel multiplication task. The number of tasks per task group is computed using the following heuristic on the first input argument.

```
heuristic = 0
A = input matrix #1

//First we calculate and store the skewness of every row
row_skewnesses = double[A.rows().size()]
foreach row in A:
      // Calculate the sample skewness of the row using method of moments:
      // (moment_3 / sample_variance^(3/2))
      u = mean(row)
      svar = sample_variance(row) // Unbiased estimator for sigma^2
      moment_3 = third_moment(row) // Expectation((x-u)^3))
      j = moment_3 / (svar^1.5) // MoM estimator for skewness
      row_skewnesses[row] = (sstd != 0) ? j : 0 // Avoids a NaN

//Then we compute the average of the skewness
heuristic = mean(row_skewnesses) – ln(2)

// That heuristic is then used to pick the task size K
N = A.rows().size()
S = N^2 * e^(heuristic)
optimal = N^2 / 2
K = MAX(S, optimal)
```

If the average row skewness is much larger than zero, then S grows to be larger than N^2, and K is chosen such that there will be only one large task (thereby eliminating all possible benefits of parallelization). If the matrix is symmetric, S will be approximately N^2/2, and an optimal choice will instead be made. After computing the task group size K, it is passed to the ParallelMatrixMultiply class which performs the computation and returns the result to the client.

4.4.1.2.4 ParallelMatrixMultiply

This is the wrapper class that splits the matrix multiplication problem into task groups for dissemination to the worker threads according to the above heuristic. ParallelMatrixMultiply calls the MultiplicationTaskGenerator to get groups of dot product operations that constitute a task group. It then starts the RPCServer and spawns the individual worker threads (RPCClients) and waits for the task queue to be empty and the worker threads to have died. It then returns the resulting matrix.

4.4.1.2.5 RPCServer

Socket server. This server spawns a new RPCClientHandler for each RPCClient that connects.

4.4.1.2.6 RPCClientHander

The RPCClientHandler consumes objects of type Request from the RPCClient. There are two basic types of requests. GET requests indicate a client wants a new taskgroup from the handler and UPDATE requests which indicate that the client has finished its task and wants to notify the client handler of the results. These requests are serialized using GSON and sent over the network to the RPCClient.

Outside of this communication, RPCClientHandler's main function is to maintain a shared queue called *tasks*, and a results matrix that contains the interim dot product results from the RPCClients. When a client requests a new task using a GET request, the RPCClientHandler pulls a SubMatrixTask off of the queue and sends it over the wire to the worker. When a RPCClient has completed its dot product operations, it sends the JSON encoded results back over the socket to the RPCClientHandler using a UPDATE request and the client handler updates the global result matrix.

4.4.1.2.7 RPCClient

This class serves as the worker thread for the parallel matrix multiplication. It communicates with the RPCClientHandler using JSON RPC. It continuously requests new dot product task groups from the handler using a GET request until the multiplication task is complete. It expects the response to a GET request to be a GSON serialized SubMatrixTask object. When it receives a SubMatrixTask, it will perform the dot product operations and then send an UPDATE request back to the handler.

4.4.1.2.8 MultiplicationTaskGenerator

Given a group size (number of dot products to group into one task), the MultiplicationTaskGenerator splits all of the dot products into groups of that size.

4.4.1.2.9 SubMatrixTask

This class is a wrapper around a set of vectors (columns from one matrix and rows from the other), and a list of <i,j> dot product operations to complete on those vectors. It is used to communicate tasks to worker threads.

Pseudocode follows:

```
class SubMatrixTask
{
      // Maps row in A to its contents
      HashMap<Integer, double[]> submatrixA;
      // Maps column in B to its contents
      HashMap<Integer, double[]> submatrixB;
      // List of dot product tasks to perform
      List<Pair<Integer, Integer>> tasks;
}
```

4.4.1.2.10 Request

This is an object used to serialize SubMatrixTask requests and updates, for communication between the RPCClient and RPCClienthandler. Pseudocode follows:

```
class Request
{
	String requestType // GET or UPDATE
	// For update requests the client notifies the handler of the
	// dot product results
	Pair[] updatePoints // coordinate of (A*B)[i][j]
	Double[] updates // value at that coordinate
}
```

4.4.1.2.11 Parallel Multi-Point Shortest Path Component Architecture (Non-Vulnerable)

Multi-point shortest path computation is implemented in the ShortestPathOperation class. This class parses an RPC request where the first input is an adjacency matrix A, and a second argument is a 1 x k vector listing the nodes (row numbers) in A that shortest path computations are to be computed on. Shortest paths are computed for each of these k nodes to each other node in the graph. Note that k is restricted to be < log(n) to avoid an unintended vulnerability. Since the user can request shortest paths for k nodes, we divide those k shortest path computations into two task groups. As a red herring, the number of tasks being assigned to each group is a random function of the same heuristic used by the matrix multiplication. This will allow for a very uneven task splitting, but because of the lesser complexity of the algorithm and input size restrictions, it will not create a STAC vulnerability within the allowed budget.

Pseudocode follows:

```
	Random R = new Random();
	R.setSeed(matrix heuristic * current time);

	int prefSize = random integer <= number of vertices; // preferred task
size
	int tasks = 0;

	Vector<Integer> curTask = new Vector<Integer>();
	numThreads = number of cores on system;

	foreach v in vertices:
	{
		add v to current task
		// If the task is the preferred size, check and see if it is the
	last task
		if (task size = prefSize and & (tasks < numThreads - 1))
		{
			tasks = tasks + 1;
			create a new task worker assigned with curTask
			reset curTask
		}
	// See if there are any tasks left over that hasn't been assigned

	}
	if (curTask.size() > 0)
	{
		//create a new task worker assigned with curTask
```

```
}
```

After the tasks are split up, each worker computes the shortest path operations for its assigned vertices using a binary heap implementation of Dijkstra's algorithm, and then the results are returned.

4.4.1.2.12 Graph Minimum Spanning Tree (Non-Vulnerable)

Given a single weighted adjacency matrix as an input, the MSTOperation class wraps a sequential implementation of Prim's algorithm for computing minimum spanning trees implemented on a binary heap. Even though this operation takes only one argument (allowing it to be twice the size), the algorithm's worst case n*log(n) time complexity will not violate the budgets set for multiplication.

4.4.1.2.13 Graph Laplacian (Non-Vulnerable)

Given a single adjacency matrix as an input, the LaplacianOperation class wraps a simple sequential implementation of a graph Laplacian calculator. This operation is simply the degree matrix minus the adjacency matrix. As such, when given the input matrix it first computes the degree matrix, and then performs the subtraction operation. This requires n^2 time, and hence won't violate the matrix multiplication budget.

4.4.1.2.14 Complexity Analysis

4.4.1.2.15 Complexity Parameter for Vulnerable Algorithm

The only legal input to the matrix multiplication algorithm is a pair of square, i.e., n x n, matrices, encoded as CSV strings and embedded within the JSON request format.

4.4.1.2.16 Mapping of Input Bytes to Complexity Parameter

Each element of an input matrix must be specified using at least 17 ASCII digits; e.g., a matrix element with value 3 would be encoded 00000000000000003. Each ASCII digit takes up 1 byte of input, hence a single matrix element takes up at least 17 bytes. Since matrices are serialized as CSVs, each element is also separated by a comma which takes up another byte. An (n x n) matrix needs n(n-1) bytes of commas in its CSV representation. Further, each row ends with a newline character requiring another n bytes. The remainder of the JSON request envelop and the HTTP POST header takes up about 300 bytes.

In totality, a matrix multiplication on two (n x n) square matrices requires roughly 300 + 2(17 * n^2) + 2n(n-1)+n total bytes of input.

The vulnerability manifests when the first of the two input matrices takes a special form.

4.4.1.2.17 Frequency of Worst Case Inputs

Inefficient performance is realized whenever the rows of the first input matrix have nonzero statistical skewness.

Let S be the average row skewness of the first input matrix. The task group size K is defined to be the maximum of (n^2)/2 and (n^2) * exp(S-ln(2)). The latter is greater than the former when exp(S-ln(2)) > 1/2, which occurs when S > 0.

A worst case input is one which causes the task group size to be greater than or equal to n^2. This condition only occurs when exp(S-ln(2)) > 1, meaning S >= ln(2).

Skewness is a convenient choice for the heuristic. It is a fairly complex function upon which to utilize abstract interpretation or symbolic analysis due to its extensive use of doubles and the operations performed thereupon (powers and roots).

Further, using exp(S-log(2)) will center the algorithm's task group size around the optimal value for symmetric matrices, leading to good expected performance on symmetric matrices. This choice was intentional since almost all fuzzing attempts are likely to be made utilizing simple random number generators such as Java's "Random.nextDouble" method or Python's "random.random" function. By default, they sample from uniform or Gaussian distributions which are symmetric. We therefore believe it to be unlikely that randomly generated matrices will trigger the vulnerability

4.4.2 SmartMail

4.4.2.1 Description

SmartMail is a service that receives email-like messages from a client and delivers them to one or more recipients. It also provides mailing list functionality, including the ability to send a message to a mailing list and to retrieve the addresses of all the public subscribers to a mailing list.

SmartMail contains **a space side-channel vulnerability**. Each mailing list has a subscriber who is its designated administrator. A client can request a mailing list's subscribers, but SmartMail does not include the administrator's address in the response. The STAC secret is the secret address of a mailing list's administrator.

A SmartMail mailing list has an address, just like any other user of the system. When a message is addressed to a mailing list, the SmartMail service "de-aliases" the mailing list address; i.e., it replaces any mailing list address in the "To:" field with the addresses of that mailing list's subscribers.

When a SmartMail message has multiple (de-aliased) recipients specified, the message delivery logic delivers the message to each of those recipients in alphabetical order of their SmartMail addresses.

For each unique recipient address to which a message is addressed, SmartMail also writes a report to a log file. Reports corresponding to secret administrator addresses are written to a Secret_Addresses log file, whereas reports corresponding to non-secret normal addresses are written to the Public_Addresses log file. Like with message delivery, the report logs are also serialized to disk based on the alphabetical order of their corresponding recipient addresses. The write buffer is flushed after each address' log report is appended to a log file

A logging report corresponding to some recipient address contains that address and its count of occurrences in the message's recipient list. All logging reports are padded out to a fixed length and then encrypted before being appended to the appropriate log file.

The adversary can send a message to the mailing list while monitoring the two log files for modification. Since the log files get appended based on the alphabetical order of the recipient

addresses, the order in which the file writes are observed leaks the secret address' alphabetical position among all of the other non-secret addresses.

To learn the secret address, the adversary can address subsequent messages to both the mailing list and an additional crafted address. By varying the crafted address and monitoring the order of writes to the two log files, the adversary can conduct a binary search to learn the secret address.

4.4.2.2 Software Design

4.4.2.2.1 Usage

The program is delivered as a .jar. It services client requests to send a message, to receive messages, and to get the list of subscribers to a specified SmartMail mailing list; note that no client will actually be supplied. For each unique recipient address to which a new message is addressed, a logging report is generated, encrypted, and appended to one of two log files. A SmartMail address takes the form *user*@SmartMail.com, where *user* is any string of case-insensitive letters of length up to 25. User message boxes are stored in memory, and thus do not persist across restarts of the program.

4.4.2.2.2 Architecture of the Application

4.4.2.2.3 Dependencies

The external dependencies are Apache's Mime4J, Apache's Hadoop (only uses some Hadoop utility functions but does not rely on Hadoop for primary map-reduce functionality), Google's Guava, and Jetty.

4.4.2.2.4 Included Data

The delivery of the challenge problem will include canned data. In particular, one or more mailing lists and their respective subscriber lists will be in included in the supplied data.

4.4.2.2.5 Data Types

EmailAddress, SecureEmailAddress, and BodyWord objects all inherit from the abstract MessageWord. As such, they all contain a string data member *value*. Each instance of a word in the message body gets wrapped by a BodyWord. Likewise, each non-secret address gets wrapped into an EmailAddress object and each secret address gets wrapped into a SecureEmailAddress object.

4.4.2.2.6 Modules and Components

The SmartMail application consists of the following major modules and their respective components.

Email Manager Module

A network facing module that implements SmartMail's message processing interface. This module includes the ability to accept, validate, and parse to-be-sent messages.

EmailParser

Responsible for validating a to-be-sent message and parsing all of its fields. The validity checks include enforcing maximum lengths on all addresses in the 'To' and 'From' fields and checking the well-formedness of all addresses. Assuming that a message passes validation, the output of the EmailParser is an EmailEvent object which is passed to the MapReduce Process Controller Module. The members of an EmailEvent include 1) a reference to an EmailAddress object which embeds the sender's address, and 2) a set containing MessageWord objects. The set includes an EmailAddress or SecureEmailAddress object arising from each of the addresses in the "To:" field. The EmailParser gets all of these EmailAddress and SecureEmailAddress objects from the AddressBook component.

Maintains a database of known SmartMail addresses and is also responsible for "de-aliasing" mailing list recipient addresses. AddressBook is used by EmailParser to map address strings found in the 'To' field of to-be-sent messages to corresponding EmailAddress and SecureEmailAddress objects.

4.4.2.2.7 MapReduce Process Controller Module

Processes message data to generate counts of recipient addresses. It receives an EmailEvent object from the Email Manager Module and uses a Map Reduce algorithm to count the occurrences of each unique recipient address.

Pipeline Controller

Directs data through the pipeline's other components.

SetPartitioner

Partitions the set of EmailAddresses and SecureEmailAddress in the EmailEvent into subsets. The partition size is fixed by the implementation. Each partition is passed to its own Mapper.

Mapper

Each Mapper acts on an individual partition. It maps each of the partition's EmailAddress and SecureEmailAddress objects to the address string embedded by the object.

Reducer

A Reducer is created for each unique address string which was mapped to by any Mapper. Each Reducer receives all of the objects that map to its address string. The Reducer outputs the address string and the count of EmailAddress and SecureEmailAddress objects which mapped to that address string. Those results are passed on to the *Logging Module.*

Logging Module: Generates log reports concerning delivered messages. These reports incorporate the statistics received from the MapReduce Process Controller.

Log Generator:

Generates a fixed-length, Advanced Encryption Standard (AES) encrypted log report for each recipient address to which a sent message was addressed, including the address itself and the count of occurrences of that address in the message's recipient list.

SecureTermMonitor:

If a message is addressed to a mailing list and the secret address of that mailing list's administrator is written within the message's body, then this component generates a security warning. To determine whether the warning should be emitted, the SecureTermMonitor iterates over all of the BodyWord objects which were parsed out of the message body. Observation of this warning serves as a STAC oracle.

Log Writer:

Writes a log report to the appropriate log file.

Message Controller Module:

Contains one component, the EmailSender/Receiver, which implements the logic for sending and receiving messages.

4.4.2.2.8 SmartMail Process Flow

This section describes the process that occurs when a message is sent. The process is illustrated in Figure 42.

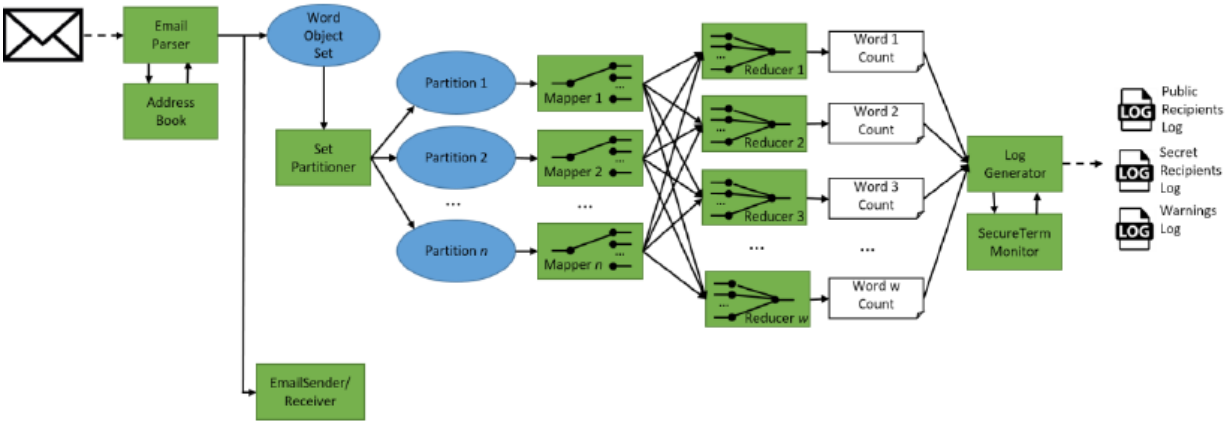


Figure 42. SmartMail Send Message Process

When the EmailParser is passed a to-be-sent message, it first performs its validity checks. The AddressBook is used to de-alias any mailing lists addresses and to map each (non-aliased) sender and recipient address to either an EmailAddress object or a SecureEmailAddress object, all of which are inserted into the WordObject Set.

Once the WordObject set has been populated, the EmailParser passes it to the MapReduce Process Control module, which is responsible for computing recipient address count statistics. To compute the statistics, the SetPartitioner partitions the WordObject Set into *n* subsets, where the maximum size of any partition is fixed by the implementation; hence *n* is just the size of the WordObject Set divided by the maximum partition size and rounded up to the nearest integer.

A Map Reduce style algorithm is used to compute the recipient address count statistics. It works by instantiating one Mapper for each partition and one Reducer for each unique address string embedded in the WordObjects of the WordObject Set. Each of the partitions is worked upon by its own Mapper. For each WordObject in its partition, a Mapper extracts the string embedded in the WordObject and uses that string to determine the Reducer to which that WordObject should be delivered. In other words, a Reducer is associated with a string, and the Mappers collectively pass all WordObjects which embed that string to that Reducer. A Reducer simply counts and outputs the number of WordObjects which were mapped to that Reducer.

The Logging Module receives the counts from the Reducers and uses that information to generate log reports, one per unique recipient address. Finally, the Logging Module writes the reports to disk in alphabetical order of the corresponding recipient addresses.

4.4.2.2.9 Inputs and Outputs

SmartMail services three types of client requests: 1) Send message, 2) Receive messages, and 3) Get list of subscribers to a mailing list. Requests are made to the server using HTTP.

*Send Message Request**:*** Sends a message to one or more recipients. Messages addressed to non-existent addresses are silently dropped, however those addresses and their respective counts are still logged in the same manner as extant addresses.

http://*server_address*:8988/*email.cgi*?from=*from*&to=*to*&subj=*subj*&content=*content*

server_address: IP address of server, probably localhost or 127.0.0.1

port: 8988

path: /email.cgi

from: The SmartMail address of the message's sender. Of form *user*@smartmail.com, where *user* is letters only, case insensitive, and has max length 25.

to: List of recipient addresses, separated using semicolons. Same rules as *from* field.

subj: A plain text description of the message. Max length 125.

content: The plain text content of the message. Max length 500.

Possible Result #1: "OK"

Possible Result #2: "Error:Java Exception internal error"

*Get Mailing List Subscribers Request***:** Retrieves the list of subscribers to a specified mailing list.

http://*server_address*:8988/list.cgi?list=*list*

list: Address of the mailing list. The SmartMail address of the mailing list. Of form *user*@smartmail.com, where *user* is letters only, case insensitive, and has max length 25.

Possible Result #1: List of subscriber addresses, separated with semicolons.

Possible Result #2: "Error: "+Variable length error message.

Possible Errors: "No such List", "Empty List", "Java Exception internal error"

*Retrieve Message Request***:** Retrieves a message which was sent to the specified user. The message received is the next item in a FIFO queue. Each retrieve request pops a message off the queue.

http://*server_address*:8988/getm.cgi?user=*user*

user: The address of the user for which a message should be retrieved. Of form *user*@smartmail.com, where *user* is letters only, case insensitive, and has max length 25.

Possible Result #1: The subject and contents of a retrieved message.

Possible Result #2: "Error: "+Variable length error message.

Possible Errors: "No message for you", "No such user", "Java Exception internal error"

In response to a send message request, the SmartMail mail service will deliver the message to its in-memory mail boxes. Further, it will generate and write out fixed-length, AES encrypted logging reports. Each logging report contains three semicolon separated fields:

1. A recipient address from the message's recipient list.
2. The count of occurrences of that recipient address in the message's recipient list.
3. Padding, filled with a variable number of arbitrary bytes to bring the length of the report up to the fixed universal report length.

4.4.3 tsp-challenge (tour_planner)

4.4.3.1 Description

This challenge program is based on GraphHopper, an open-source HTTP routing service that uses OpenStreetMap data. The challenge augments GraphHopper with the ability to calculate a Traveling Salesman tour between any user-specified subset of the 25 largest cities in Massachusetts.

Note that the challenge only supports tours between a fixed set of places, not through arbitrary points like GraphHopper's existing routing API. This limitation is imposed because the challenge's vulnerability depends on the use of a precomputed distance matrix that provides a cost metric between every pair of places (cities). This distance matrix implicitly defines a list of edges ordered by weight, which is what's used during the actual tour calculation.

The distance matrix is precomputed and stored in data/matrix.csv, to be loaded at startup time by the HTTP server.

This challenge contains a **timing side channel**. The "secret" is the set of cities for which a client requests a tour, or, equivalently, the tour that the server returns. (These are equivalent because tour calculation is deterministic and the order of cities in the original request doesn't matter.) The side channel is probabilistic: observing the side channel greatly increases the probability that an eavesdropper can guess the tour a client has requested.

Although HTTPS is not actually used in the challenge (to avoid the complexity of including a server certificate, getting clients to accept it, etc.), assume it would be used in a real-world deployment, so the tour a client requests cannot be recovered by simply reading the request content. An eavesdropper can exploit the side channel by measuring only the timings between the server's response packets.

4.4.3.2 Software Design

Precomputed distance matrix

Tours are calculated from a pre-computed distance matrix in the file `data/matrix.csv`. This matrix contains pairwise weights between every pair of a fixed set of cities.

Minimum spanning tree (MST) construction

The challenge calculates a tour as follows:

1. Construct a minimum spanning tree over the requested destinations using Prim's algorithm.
2. Output the vertices of the minimum spanning tree in depth-first order. The server returns the resulting sequence as the approximately optimal order in which to visit the requested destinations.

The timing side channel occurs because of two unusual aspects of the MST construction:

1. Instead of reducing the graph to only those vertices that are part of the requested tour, the challenge operates on the *entire* distance matrix, or rather, a list of its edges sorted by increasing weight. To find the next edge to add, it does a linear search of these sorted edges until it finds the first (i.e., least-weight) edge that can be added to the MST (i.e., the first edge that has one endpoint in the MST so far, and one endpoint not in the MST but in the set of requested tour destinations). As a result, the time to add each edge is linearly proportional to its position in the sorted list of all edges from the distance matrix.
2. Instead of returning a single HTTP response, the server uses *server-sent events* to send progress events to the client when it begins constructing the MST and after adding each edge. This exposes the time it takes to add each edge to any eavesdropper.

4.4.4 Collab

4.4.4.1 Description

Collab is a program for event scheduling. A user may create new events and de-conflict them with respect to his schedule. Additionally, a special class of users, called auditors, can schedule special auditing events for other users. The audit events scheduled by these auditors are invisible to the other users, so that the targets of said audits do not know when they are coming.

In order to create new events, a user works within a scheduling sandbox. The sandbox is a temporary calendar, pre-populated with any pre-existing events scheduled for the user, in which they may create new events. Once the user is satisfied with their new scheduling, they commit their sandbox onto the master calendar. Otherwise, they may discard their sandbox.

This challenge program exhibits a **timing-based, side-channel vulnerability** where the secret is the time of the auditing event targeting the adversarial user, of which there may only be one. The secret manifests in the form of an audit event ID, which is an integer that represents the time of the user's audit. Note that Collab itself never translates an event ID into the time domain; rather, the interpretation and mapping of an event ID to time is left to the users. As such, the secret to be retrieved is simply an integer which Collab treats as a proxy for event time. The program is structured in such a way that only the attacker's own audit event could be exposed to him; thus the attacker need not be concerned with distinguishing his own audit event from those of other users.

4.4.4.2 Software Design

When a user creates a new scheduling sandbox, a B-tree-like temporary indexing structure is created for that session. At the start of the scheduling session, the temporary indexing structure is populated with the pre-existing events from the user's calendar — including his hidden auditing event, if it exists. As the user creates new events within their sandbox session, the temporary index is additionally populated with any new events they create. Once the user is satisfied with their new event schedule, they commit the new schedule from the temporary index into the master calendar.

The vulnerability arises because an adversarial user knows how their pre-existing events will be populated into the temporary index during a new scheduling sandbox session, and hence they may anticipate how the temporary tree would be structured if it did not contain an auditing event. The adversary can use this knowledge to probe the tree for unexpected timing differences in its splitting behavior as new events are inserted. When the adversary observes that the timings do not reflect their expectations, they can infer information about their hidden auditing event.

Collab is architected to include several major components: Client, Server, BTreeIndex, and SchedulingSandbox with its dependent TempIndex. The EventResultSet object links all of these components; it is a messaging object used to communicate information about events among the other components. Figure 43 depicts the relationships between each of these components.

The following is a description of each of Collab's major architectural components:

- CollabClient: Provides a user interface for use by all users. It enables users to search for events and create new events. The client is minimally stateful, meaning that session information relating to the creation/deletion of events is maintained on the CollabServer. The client communicates to the server via a UDP interface implemented on CollabServer.
- CollabServer: The CollabServer component provides a network and session management interface which wraps other components like the BTreeIndex and SchedulingSandbox. It provides the following functionality:
    - Input Parsing: The parsing of incoming network requests.
    - Input Validation: The validation of parsed inputs.
    - Component Linking Logic: Provides application flow logic, including logic that links together base functionality. For example, it includes logic for searching the BTreeIndex and for forwarding the results to the SchedulingSandbox.

    - Session management and Authentication: Tracking of client state and authenticating that a user has access to perform an action.

- BTreeIndex: A data structure that stores the master calendar. Implemented as a traditional B-tree.
- SchedulingSandbox: Implements functionality that enables users to schedule new events. Users may create events in a sandbox without altering the data in the master BTreeIndex.
- TempIndexNode: Implements a tree index for storing events in a scheduling sandbox. It is implemented as a modified B-tree. It only contains events related to the user who created the scheduling sandbox. The challenge problem vulnerability exists exclusively in this component's data structure.
- EventResultSet: A data structure that contains search results from the master calendar. It acts as a messaging envelope for communicating search results between components. In Error: Reference source not found, the EventResultSet is depicted as being part of a data flow between the BTree and the SchedulingSandbox. This is because the data manipulated by a SchedulingSandbox instance has provenance in the BtreeIndex. When a SchedulingSandbox instance is committed, all changes made to its the EventResultSet are pushed to the master calendar.

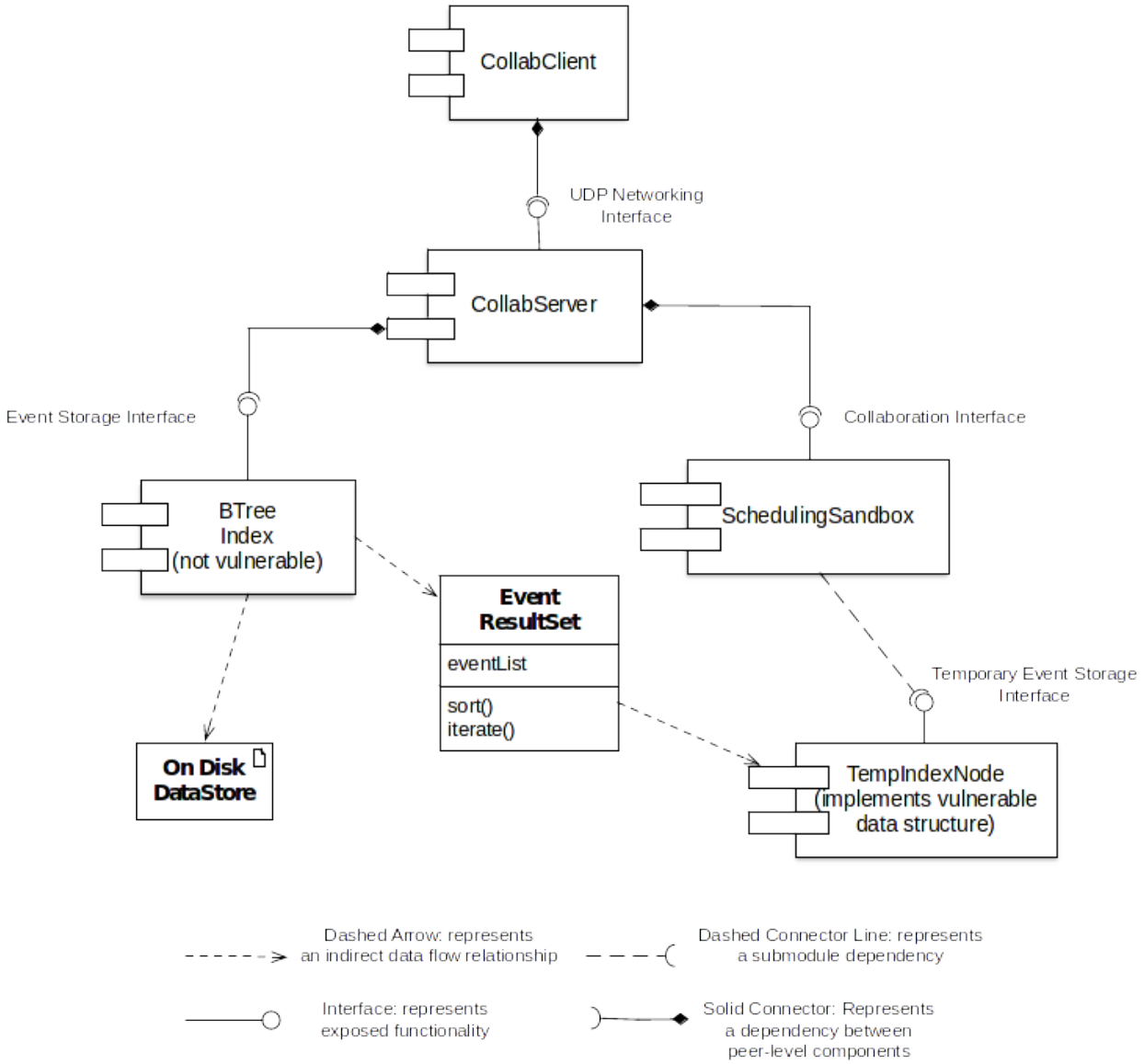


Figure 43. Collab architecture diagram

Collab Process Flow

This section describes the process by which a scheduling sandbox is initialized for a user and for how they interact with their sandbox. The UML sequence diagram shown in Figure 44 illustrates the process.

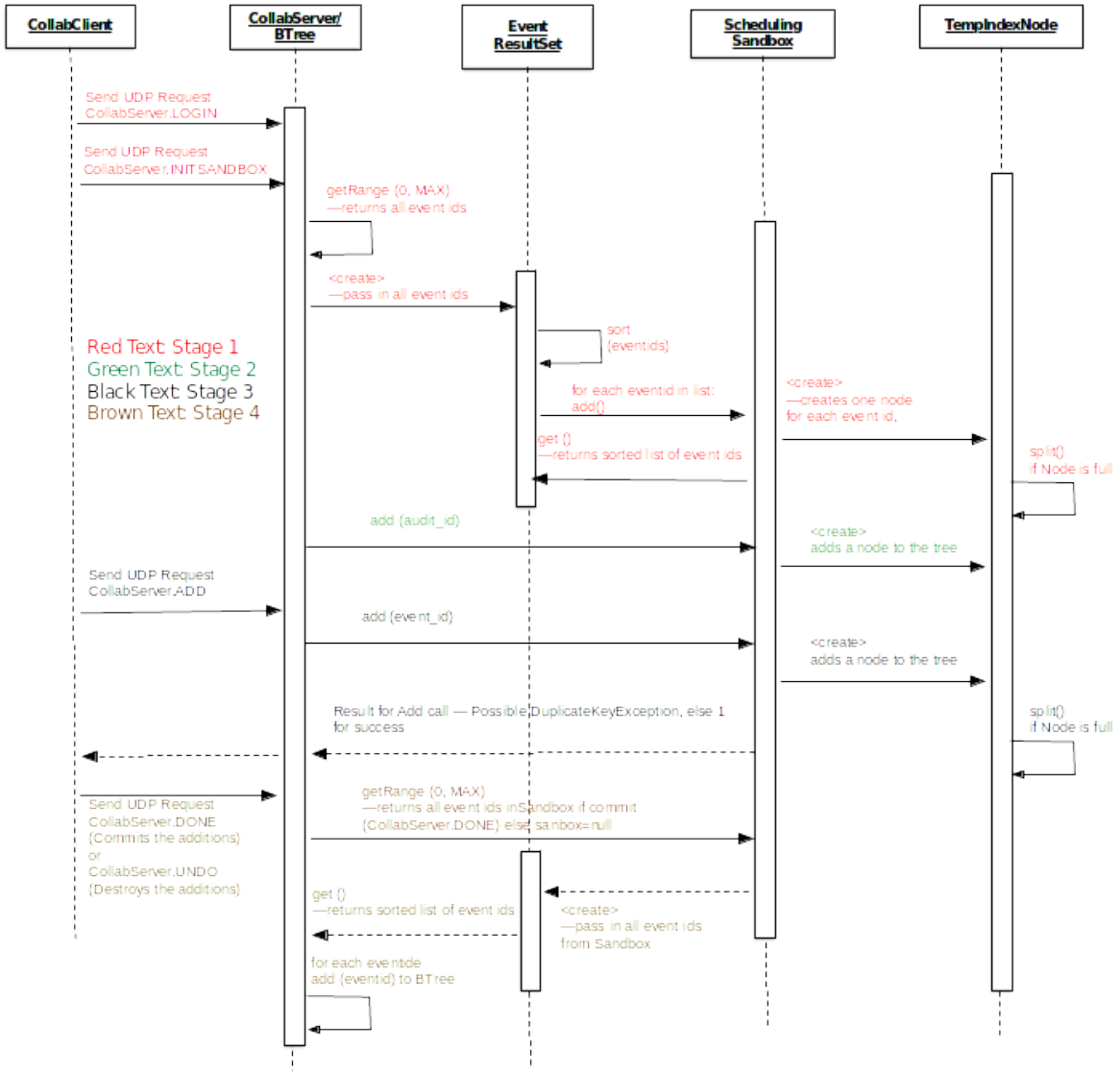


Figure 44. UML Sequence Diagram Depicting Scheduling Sandbox Process

- Stage 1, Populate User Events: All events for the requesting user are retrieved and inserted into the SchedulingSandbox's TempIndexNode. This initialization step also performs sorting, resulting in a balanced tree.
- Stage 2, Populate Auditor Event: If the user has a hidden auditing event scheduled, it too is inserted into the tree. It is placed into the appropriate sorted position based on its ID.
- Stage 3, User Works in Sandbox: The user can search and add events to the temporary sandbox tree. The diagram shows the process flow for an 'add'.
- Stage 4: Commit/Discard: The cser can commit their sandboxed schedule to the master tree, or discard their work.

4.4.4.2.1 Inputs and Outputs

Input to the Collab server is always in the form of a UDP packet, as are the server's response outputs. The following shows the packet's data payload structure for each of the operations supported by the server.

The generic form of a server request is a 1-byte operation ID, followed by a variable number of fields depending on the particular operation. All operation IDs and other constants are denoted in Base 10. Vertical bars denote separation between the fields of the packet, but are not actually to be sent in said packet. The server immediately responds to all well-formed requests with an Acknowledgement message (a UDP packet containing just the number 1 as a 4-byte integer). Subsequently, the server may also respond with another UDP packet whose payload contents depends on the operation ID of the request being serviced.

Login: Creates a session for user *username.*

- Request: | *23* | *uname_length* (4 bytes) | *username* |
- Possible Response #1: Success
    - | ACK |
    - | *session_id* (4 bytes) |

Add: Adds an event with ID *event_ID* for the user with session ID *session_id.*

- Request: | 3 | *session_id* (4 bytes) | *event_ID* (4 bytes) |
- Possible Response #1: Success
    - | ACK |
- Possible Response #2: DuplicateKeyError
    - | -5 (4 bytes) |

SearchMain: Searches the master calendar for all publicly visible events belonging to user *username* for which the event ID is in interval (*min, max*).

- Request: | 10 | *uname_length* (4 bytes) | *username* | *min* (4 bytes) | *max* (4 bytes) |
- Possible Response #1: Success
    - | ACK |
    - | <event_id> |*, where <event_id> is a 4-byte integer denoting one of the the requested event IDs. Each event ID is sent back to the user is its own UDP response, hence the Kleene star.
    - |-8 (4 bytes) |

InitiateSandbox: Initiates a new scheduling sandbox for user that owns session *session_id*.

- Request: | 13 | *session_id* (4 bytes) |
- Possible Response #1: Success

    - | ACK |
    - | ACK |
- Possible Response #2: Failure, when the user already had an initiated sandbox.
    - | ACK |
    - | -1 (4 bytes) |

SearchSandbox: Searches the calendar of a scheduling sandbox for all events for which the event ID is in interval (*min, max*). A scheduling sandbox must have already been initiated for this operation to work.

- Request: | 11 | *session_id* (4 bytes) | *min* (4 bytes) | *max* (4 bytes) |
- Possible Response #1: Success
    - | ACK |
    - | <event_id> |*, where <event_id> is a 4-byte integer denoting one of the the requested event IDs. Each event ID is sent back to the user is its own UDP response, hence the Kleene star.
    - | -8 (4 bytes) |

CommitSandbox: Commits the contents of the scheduling sandbox associated with session *session_id* to the master calendar, then destroys the scheduling sandbox.

- Request: | 14 | *session_id* (4 bytes) |
- Possible Response #1: Success
    - | ACK |
    - | ACK |
- Possible Response #2: Failure, when the user had no initiated sandbox.
    - | ACK |
    - | -1 (4 bytes) |

DestroySandbox: Destroys the scheduling sandbox associated with session *session_id* without first committing its contents to the master calendar.

- Request: | 15 | session_id (4 bytes) |
- Possible Response #1: Success
    - | ACK |
    - | ACK |

Note that there is not a one-to-one mapping between the Collab client's commands and these UDP requests which are accepted by the server. Rather, the UDP requests accepted by the server are more granular than the commands accepted by the client. As such, some of the Collab client's

commands actually result in multiple UDP packets being sent to the server. For instance, the client's Add command first sends an InitiateSandbox UDP request if the user's sandbox has not yet been initialized, and then sends an Add UDP request.

4.4.5 InfoTrader

4.4.5.1 Description

InfoTrader is a document server for stock trading professionals. Each document on the server is expected to contain research about a publicly traded company. Users can upload new documents to the server and also get documents already on the server.

Documents on the InfoTrader server are organized under a fixed hierarchical directory structure. The directory hierarchy has at its top a root folder, under which are folders for various industry segments, e.g., Pharmaceuticals or Technology. Each publicly traded company represented on the platform has a directory, labeled using the company's stock ticker symbol, located under its appropriate industry segment directory.

The format of an InfoTrader document is Genealogical Data Communication (GEDCOM) [10] and there is a global maximum size on individual documents. In addition to text, an InfoTrader document can also contain hyperlinks to other documents on the server. A hyperlink in a document is denoted by placing the name of the linked-to document in a GEDCOM level-0 SOUR (source) block beneath the message body (cf. Inputs and Outputs section for an example).

The InfoTrader server services two types of requests: *Get* and *Post. Get* requests are used to retrieve documents from the server, and *Post* requests are used to push new documents onto the server. The server responds to a *Get* request with the requested document(s) if they exist, otherwise it returns a *Document Not Found* error. The server responds to a *Post* request by generating an updated sitemap of the server's contents reflecting the newly uploaded document. An InfoTrader sitemap specifies the complete directory and document hierarchy of the server in XML. An InfoTrader sitemap contains three types of elements: Directories, Documents, and HyperLinks. A Directory element's children may be other Directories or Documents, and a Document's children may be HyperLinks.

This challenge program exhibits a **space-complexity attack**.

InfoTrader's sitemap serialization algorithm is vulnerable to a space-complexity attack due to a malfunctioning guard on how hyperlinks within documents are handled. The algorithm traverses the internal representation of InfoTrader's directory structure and serializes it out to a structured-text representation. The algorithm expects the internal representation of the directory structure to be a tree, i.e., acyclic. However, this assumption can be violated by a nefarious user.

The expected functionality of InfoTrader is that documents may only link to other documents, but not to directories. However, by bypassing a weak guard, an attacker may actually *Post* a document containing a link to the root directory. When this occurs, the server updates its internal representation of the directory structure to include the root directory as a child under the document. Later, when the server attempts to serialize out the sitemap, the serialization algorithm encounters a cycle in its internal representation of the directory structure causing it to write out the directory structure again under the document.

To ensure that the vulnerability does not result in an infinite loop, the sitemap serialization algorithm keeps track of which documents it has already written out under each company directory. For instance, assume a company directory for Microsoft. If an attacker uploads a malicious document into the Microsoft directory containing a hyper-link to the root directory, then the sitemap serialization algorithm re-serializes the complete directory hierarchy under that malicious document. However, when the malicious document is again reached during the re-serialization, a third serialization is not triggered because the algorithm recorded earlier that the malicious document was already serialized under the Microsoft directory and therefore need not be serialized again.

By repeatedly crafting malicious documents containing unexpected links back to the InfoTrader root directory, the adversarial user can cause the generated InfoTrader sitemap to grow unexpectedly quickly in size. Since documents have a fixed maximum size, a size utilization threshold on the generated sitemap can be specified which is only surpassable using the vulnerability.

4.4.5.2 Software Design

Dependencies

- Jetty

Included Data

The delivery of the challenge problem will include canned data. In particular, the server will be pre-populated with a directory structure and documents pertaining to a few publicly traded companies.

Data Types

HyperLink, Document, and Directory objects all inherit from the abstract NodeBase class (cf. Figure 45). As such, they all contain a string data member *name*. Each Directory object may have multiple Document and/or Directory objects as children. Each Document object may have some HyperLink objects as children. Each HyperLink object refers to a Document; note that the serialization algorithm can be tricked into believing that a Hyperlink refers to the root Directory, as opposed to a Document, however this is not actually reflected in the internal data-structure (cf. Vulnerable Algorithm section). No Document may have the name of a Directory, with the purposefully buggy exception that a Document can have the empty name which is also the name of the root Directory.

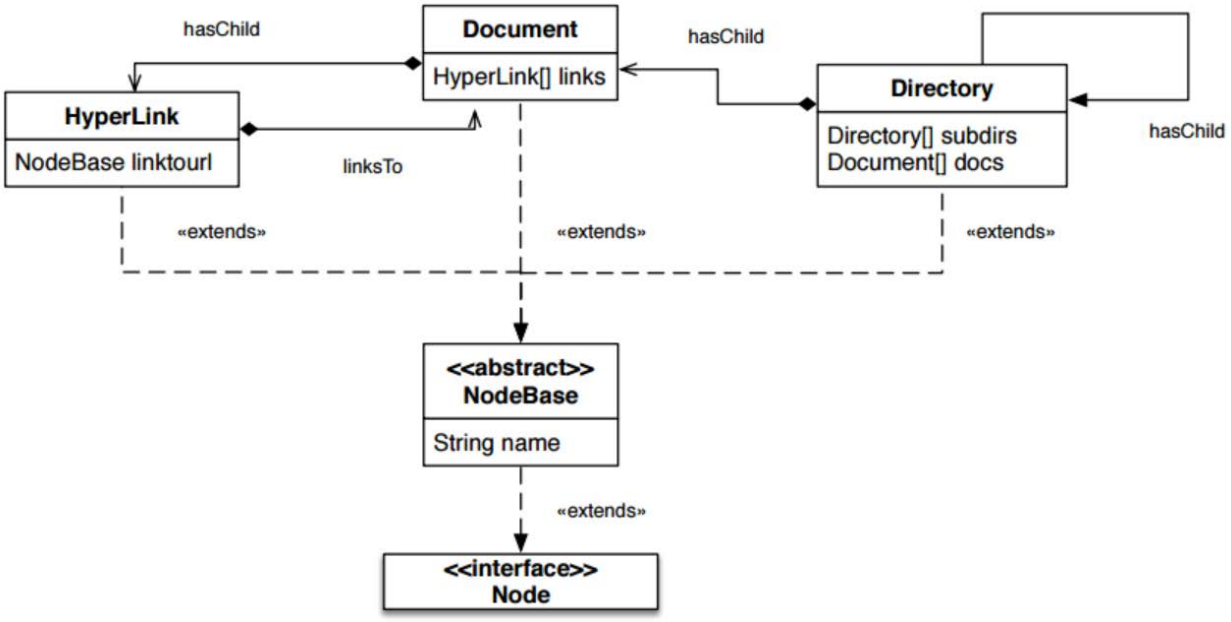


Figure 45. Data Type diagram for InfoTrader

Modules and Components

The InfoTrader application consists of the following major modules and their respective components. Their interrelationships are illustrated in Figure 46.

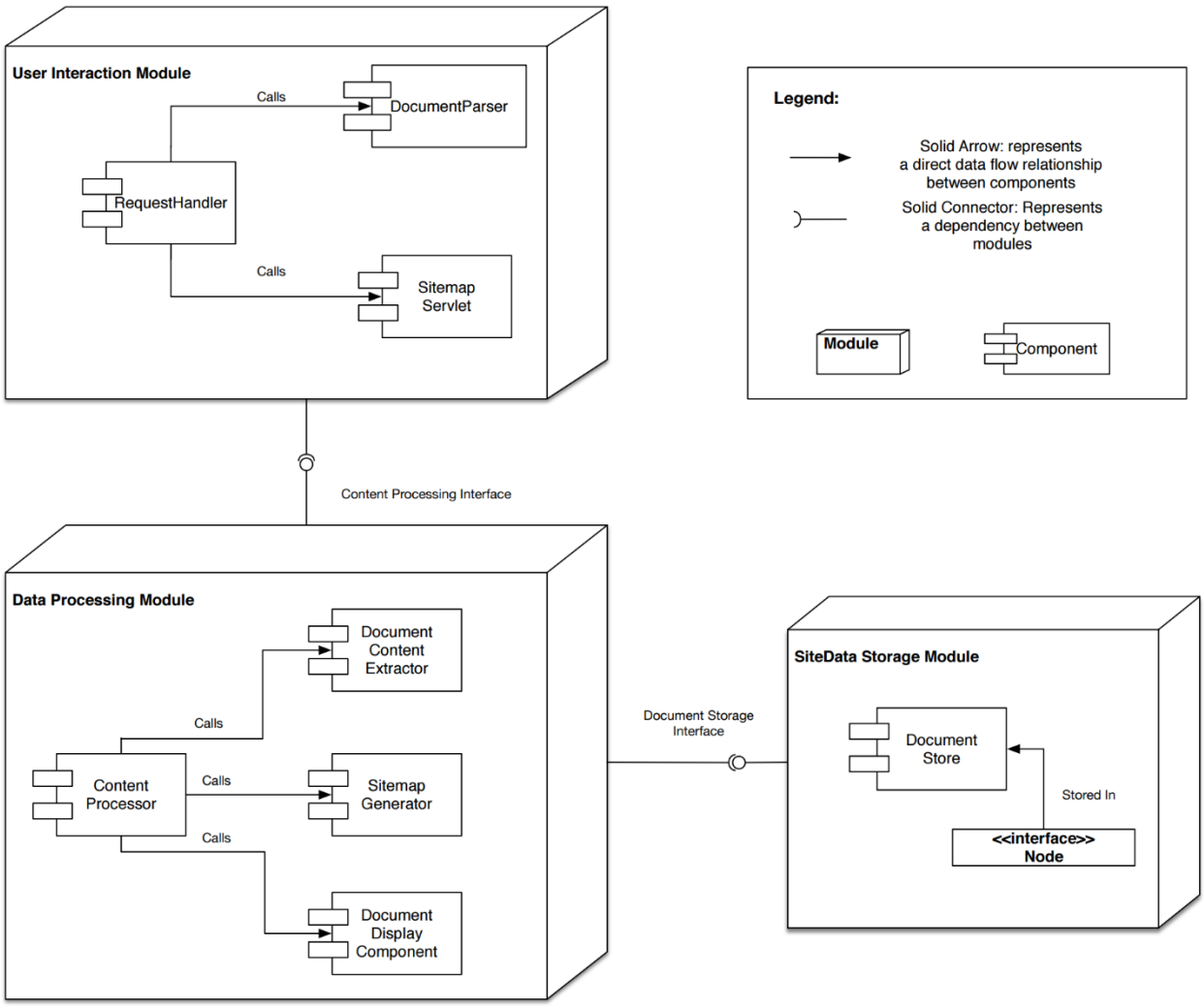


Figure 46. Architecture Diagram for InfoTrader

- **User Interaction Module:** A network facing module that implements InfoTrader's request processing interface, including the ability to accept, validate, and parse incoming requests from users.
    - RequestHandler: Handles all incoming requests and forwards them to the appropriate sub-component, DocumentParser or SitemapServlet.
    - DocumentParser: Parses POSTed documents, and validates each document by enforcing size and content restrictions.

    - SitemapServlet: Handles queries for sitemaps.

- Data Processing Module: Processes request data in order to populate InfoTrader's internal data representations or to present information to the user. It receives either 1) a POSTed and parsed Document, 2) a generated sitemap, or 3) a requested Document from the User Interaction Module. It then performs the additional processing required to prepare that data for storage or display.

    - Content Processor: A generic component that forwards information to and holds state for the other components of the module.
    - Document Content Extractor: Extracts the content of POSTed documents.
    - Sitemap Generator: Generates sitemaps, which are used by users to navigate the server and explore its Documents in a web browser. Sitemaps are serialized as XML and are treated as any other Document; i.e., the current sitemap is stored as a Document in a known location in the directory hierarchy. In particular, the sitemap is always named *sitemap.xml* in the root directory.
    - Document Display Component: Response for retrieving requested documents.
- **SiteData Storage Module:** Maintains an internal tree representation of the system's Directories, Documents, and Hyperlinks.
    - Document Store: The data-store.

4.4.5.2.1 InfoTrader Process Flow

When InfoTrader receives a POST request, the User Interaction Module parses out the arguments and performs validity checks. The resulting information is then passed on to the Data Processing Module. The Data Processing Module uses the parsed request information to instantiate a new Document object for the new document and also any required Hyperlink objects arising from links found within the document. The resulting objects and their relationships are passed to the SiteData Storage Module for storage. The SiteMap Generator is then invoked to generate an updated sitemap that reflects the modified data-store. Once generated, the sitemap is written back to the data-store as a document.

When InfoTrader receives a GET request, argument parsing and validation work in the same manner as for POST requests. The resulting information is passed to the Data Processing Module, whose Document Display component retrieves the requested document. Further, a user may also specify in a GET request whether to additionally retrieve all documents to which the specified document links, and so on transitively. In that case, the Document Display Component is also responsible for transitively traversing hyperlinks to retrieve all of the required documents.

4.4.5.2.2 Inputs and Outputs

InfoTrader services two types of client requests: 1) Get document, and 2) Post document. Both types of requests are made to the server using HTTP POST requests.

Get Document Request: Gets a pre-existing article from the user.

```
http://server_address:8988/gdoc.cgi?name=*name*&getall=[true|false]
*name*: The name of the document as it appears in the sitemap.
*getall*: Whether to recursively retrieve all documents to which this one
links.
Expected Response: The document(s).
```

Post Document Request: Pushes a new article onto the server, resulting in the generation of an updated sitemap.

```
http://server_address:8988/doc.cgi
POST data: content=*content*
```

Example POST data:

```
0 HEAD
1 SOUR MSFT
2 NAME Micosoft Zune Zooms -- Stock up 1000%
1 NOTE Zune sales look promising. Customers can't get enough.
0 @I10@ NOTE It wasn’t very easy to get my hands on a Zune. After Microsoft’s
long-pitied music player won Slate’s Reader Takeov
1 CONC er poll in which I’d promised to reassess an overlooked technology of
yore, I had to scramble to get hold of a device I hadn’t use
1 CONC d since at least 2008. My local Craigslist listings overflowed with
iPods of every variety, but there were only a couple Zunes f
1 CONC or sale, and they were the earliest, least-memorable versions of the
device. I was looking for a later-model Zune, specifically,
1 CONC the 2009-era touch-screen Zune HD. This was the best Zune Microsoft
ever made, though you might consider that damning with faint
1 CONC praise.
0 SOUR
1 TITL Top Buy is tops
0 TRLR
```

Explanation:

An InfoTrader document, such as the example above, is specified in the GEDCOM format. The numbers preceding the lines in a GEDCOM file can be thought of as data nesting levels. For instance, the “1 SOUR MSFT” line is a nested data element of the “0 HEAD” data element. And the “2 NAME …” line is in turn a nested data element of the “1 SOUR MSFT” data element. The analog in a more common document serialization format like XML would be something like <HEAD> <SOUR> <NAME></NAME></SOUR> </HEAD>.

An InfoTrader document starts with a “0 HEAD” header line, followed by a “1 SOUR <ticker>” source line where <ticker> is the ticker of the company under whose InfoTrader directory the document should be recorded. For instance, this example document would be placed under the MSFT directory. Next is a “2 NAME <name>” name line, where <name> is the name of the document.

The body of an InfoTrader document is specified using a “0 @<tag>@ NOTE <contents>” note line, where <tag> is any text (and is not used by the system) and <contents> is the body of the

document. The body of the document can be written out entirely on that line or it may instead be broken across multiple lines by using “1 CONC” concatenation lines as per the example.

Hyperlinks to another document are specified after the contents by using a “0 SOUR” source line followed by a “1 TITL <name>” title line, where <name> is the name of the document to which this document links, and a document may contain multiple hyperlinks. For instance, the example document links to another whose name is “Top Buy is tops”. As another example, if some other document were to link to this one, then that other document would include the following two lines after its contents section:

```
0 SOUR
1 TITL Micosoft Zune Zooms -- Stock up 1000%
```

An InfoTrader document is concluded with a “0 TRLR” trailer line.

`Expected Response: OK`

Vulnerable Algorithm

The Sitemap Generator contains a bug that can cause the entire directory and document hierarchy to be re-serialized under a Document once or even multiple times. A malicious hyperlink in an uploaded document is the trigger for the bug. A description of the vulnerability follows, as does pseudocode. Assume the following pseudocode for serializing out the sitemap is invoked with *serialize(“”, ∅)*; i.e., it is passed the empty string, which says to start serializing at the unnamed root directory, and the empty set, which says that no state has yet been accumulated. Explanation of “state” is forthcoming, see below.

The sitemap serialization algorithm recursively traverses InfoTrader’s internal tree representation of the directory hierarchy depth-first. Each encountered node along the way is serialized in a manner appropriate to its type; i.e., a directory is written out with its document children below it, a document is written out with its hyperlink children below it, and a hyperlink is just written out.

The three rules for serializing hyperlinks into the sitemap are as follows: 1) A hyperlink may only link to a document, not to a directory; 2) A hyperlink is only written out to the sitemap if the document to which it links actually exists; and 3) A hyperlink is only written out once beneath a document even if that document contains multiple hyperlinks linking to the same document. All three rules are important to the STAC space-complexity vulnerability.

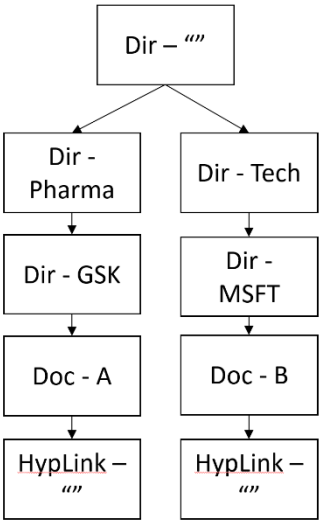


Figure 47. Example Internal Tree Representation of Directory / Document Hierarchy

To verify that a hyperlink conforms to rule #1, the target of the hyperlink is checked against the list of directory names present in the directory hierarchy. However, for STAC purposes, this list has been left purposefully incomplete. In particular, the list of directories doesn't include the empty string, which is the name of the root directory. As such, this verification check is bypassed if a user uploads a document containing an empty hyperlink.

To verify that a hyperlink conforms to rule #2, the target of the hyperlink is checked against the list of document names present in the directory hierarchy.

Hyperlink rule #3 is enforced by recursively accumulating a list of hyperlinks already written out under a document node. We refer to this list as the current accumulated *state*. When a hyperlink is written out underneath a document, a record indicating as much is added to the state. All nodes traversed beneath the document are passed the accumulated state in order to determine whether they have already been written out. Importantly, hyperlinks under different documents are treated as unique, even if they link to the same document. Confer the pseudocode for serialize_document() to see how state accumulates and is passed down the tree.

To trigger the vulnerability, the user first POSTs a document whose name is the empty string, which causes the empty string to be added to the list of document names present on the server. Next, he POSTs another document which includes a hyperlink that links to the empty string. The POST triggers a re-serialization of the sitemap. When the serializer encounters the malicious hyperlink node, both the Rule #1 and Rule #2 checks are passed, since the empty string is not in the list of directory names and since the user previously uploaded a document whose name is the empty string. As such, the hyperlink is correctly serialized as a hyperlink once, and then it is spuriously serialized again as a directory due to a missing return statement. Confer the serialize() pseudo code to observe that a hyperlink of this form does indeed result in calls to both serialize_hyperlink() and serialize_directory(). Rule #3 prevents an infinite loop by ensuring the same malicious hyperlink does not trigger yet another re-serialization under the document.

If a second malicious document containing a null hyperlink is uploaded, then they will both cause two reserializations for a total of four extraneous serializations beneath the expected one. To illustrate, let the first malicious document which was uploaded be denoted *A* and the second one be denoted *B*; the internal tree representation corresponding to this scenario is illustrated in Figure 47. When the malicious hyperlink under *A* is first encountered, it will trigger a reserialization under *A*. During that reserialization, *A*'s malicious hyperlink will be encountered again but ignored since it is in the previously-seen state set. However, *B*'s malicious hyperlink will also be encountered during this reserialization under *A*, and since hyperlinks under different documents are unique, it is not yet in the previously-seen state set. As such, a second level of reserialization will be triggered under *A*. For this level of reserialization, *B's* malicious hyperlink is also added into the previously-seen state such that neither *A* nor *B*, when they are encountered again, induces yet another reserialization. Once serialization under the top level *A* concludes, serialization will continue on the right side of tree, wherein another extraneous reserialization will occur under *B*, which in turn will induce another reserialization under *A*. This complete malicious serialization is illustrated in Figure 48.

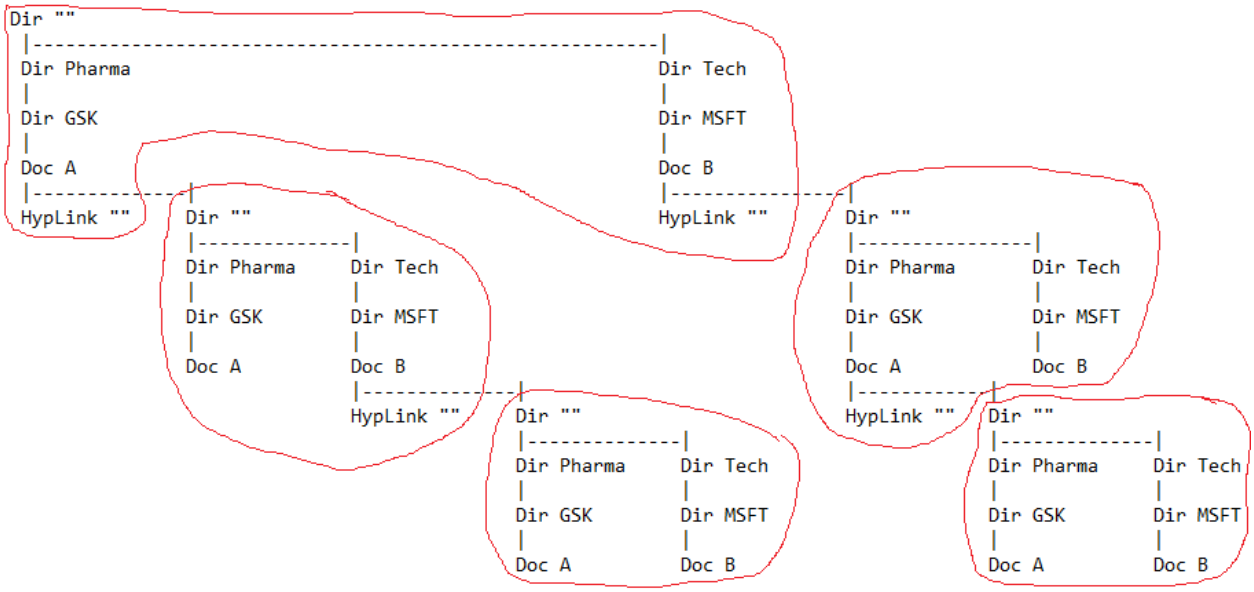


Figure 48. Malicious Serialization Arising from Scenario of Figure 47 - Red Circles Show the Five Separate Serializations of the Overall Directory Structure

Pseudocode:

```
// Calls the appropriate serialization routine depending on the type of node being processed
serialize(node, state):
if node is a hyperlink: // Determined using reflection
if node links to a Directory other than empty string (which is the name of the root dir) ||
node links to a document that is not present on the server:
```

```
return; // Don't serialize a bad hyperlink
serialize_hyperlink(node, state); // Missing return after this line allows fall
through to next if stmt
if node has the name of a directory // This check doesn't omit the empty string.
As such, it allows
serialize_directory(node, state); // the hyperlink with empty string name, but
no other HLs since
// the only other ones that could reach here are links to
// Documents and a Document cannot have the same name as
// a Directory.
if node is a document // Determined using reflection, not name
serialize_document(node, state);
return;
serialize_directory(dir, state):
write out dir;
      for each child c of dir, left to right:
      serialize(c, state) under where dir was written;

serialize_document(doc, state):
write out doc
_state = state
for each child c of doc, left to right: // c is a hyperlink
if c is not in state:
_state += c // then add it to the state
serialize(c, _state) under where doc was written // and serialize it

serialize_hyperlink(link, state):
write out link
```

Implementation Note: For conciseness, the pseudocode shows that 'state' is maintained recursively, where each node encountered by the algorithm is added to the state that is passed down to the serialization of its children. In the actual implementation, the serialization algorithm instead walks up the tree from the current node to look for other nodes that match the current node.

Red Herring

A red herring has been included in the InfoTrader application. It affects both the document retrieval functionality (gdoc.cgi request) and the document post functionality (doc.cgi). The red herring results when a document A is requested that contains a hyperlink to a document B which in turn links back to A. This results in a cycle that quickly exhausts the JVM stack, leading to a StackOverFlowException that is handled gracefully by the InfoTrader server. Since the stack is exhausted quickly and the error is handled gracefully, no excessive disk space or time utilization is incurred.

The same cycle-containing input will cause the red herring to occur in both the get and post functionality. In the post functionality (doc.cgi), the red herring manifests itself as an infinite loop in the serialization functionality where two hyperlinks that refer to each other continue to be analyzed in a loop until the stack overflows.

In the document retrieval function (gdoc.cgi), the red herring results when a red-herring-vulnerable document is requested with the 'getAll' argument set to true. The 'getAll' argument, when set to true, triggers logic that returns not only the specified InfoTrader document but also all of the other documents to which it links, and so on transitively. If 'getAll' is set to false, the red herring will not occur even when a red-herring-vulnerable document is requested.

A question targeted towards the red herring might ask if there is a sequence of requests, of cumulative size no larger than the budget's size, which results in a request taking longer than X number of seconds. On the reference platform, the supplied proof script for the actual vulnerability takes about 38 seconds, hence any time budget in excess of that should be sufficient to preclude a STAC time-vulnerability.

4.4.6 MalwareAnalyzer (malware_analyzer)

4.4.6.1 Description

Malware Analyzer is a server program that helps analysts better understand malware. In particular, it combines two malware classifiers, a packer detector, and control flow graph generator. The classifier helps analysts triage new malware, since they can use the service to determine whether new malware is similar to anything that they have previously analyzed. The packer detector tells the analyst whether or not to trust the classifier's results, since the process of packing[2] a binary executable obfuscates the features used by the classifier. The control flow graph generator enables the analyst to more easily explore their input files.

Both malware classifiers compute the pairwise similarity of a new malware instance with each of the previously submitted malware and return a rank ordered list of the top five most similar malware and their respective similarity scores. One of the classifiers uses byte 1-grams[3] for features and cosine similarity[4] for comparisons. The other classifier uses X86 assembly opcode mnemonic 1-grams for features and cosine similarity for comparisons.

The packer detector determines whether an input executable is packed by using threshold heuristics on its average byte value and byte-level entropy. Since the byte values of a packed executable generally follow a uniform distribution, their average bytes tend to be near 127 and their byte-level entropy is generally greater than that of an unpacked executable.

One possible input to Malware Analyzer is a Message-Digest 5 (MD5) digest and a byte 1-gram feature vector extracted from a new instance of malware. A byte 1-gram feature vector is an integer array of length 256, where the value at index 0 is the count of 0x00 bytes in the input file, the value at index 1 is the count of 0x01 bytes, and so on. The expectation is that the analyst would use some (non-provided) client to extract and submit feature vectors to the service. The output from Malware Analyzer includes whether the input file was packed and also the rank ordered list of the top five most similar malware with their respective similarity scores.

Another possible input to Malware Analyzer is a binary executable. In this case, Malware Analyzer will generate the byte 1-gram feature vector directly from the binary executable, and then use it in the same manner as before to determine similarity and packedness.

Yet another possible input to Malware Analyzer is the disassembly of a Windows Portable Executable 32-bit (PE32) binary executable as generated by the objdump tool. From this type of

input, Malware Analyzer can 1) determine the cosine similarity between the submitted disassemblies using opcode mnemonic 1-grams as features, or 2) construct control flow graphs for the functions represented in the disassembly.

Malware Analyzer contains a **space-complexity attack.** When Malware Analyzer receives a request, it generates some output that is written to a local file and also returned to the requestor. When a request is successfully handled, the local file is overwritten with the new output. However, if the request fails in a certain way, then the resulting error message is appended onto the log file. By inducing this type of failure, a malicious user can cause the size of the on-disk log file to grow larger than anticipated.

The failure can only occur during Malware Analyzer's computation of a binary executable's average byte value. This computation is only triggered on byte 1-gram feature vector inputs or binary executable inputs; i.e., it is only triggered on *add* requests and *Binary PUT* requests (cf. Process Flow and I/O sections for descriptions of handled request types). The computation of the average byte value is implemented as a tail recursive procedure. The average byte value procedure recurses on the input feature vector, one index at a time, while accumulating the input file's sum of bytes and count of bytes. At the bottom of the recursion, once the input feature vector has been fully consumed, the accumulated byte sum is divided by the accumulated byte count to obtain the input file's average byte value.

A space-complexity attack arises in the average byte value procedure due to Java's integer overflow semantics. By supplying a specially crafted feature vector, an attacker can cause the accumulated byte count to overflow and wrap-around. By crafting the input such that the byte count accumulator has wrapped around to exactly 0 at the bottom of the recursion, a *divide by zero* exception will be thrown. The exception causes Malware Analyzer to append the resulting stack trace to the local log file. Repeated appending of the stack trace, which will be large due to the depth of the recursion, can cause a STAC space utilization threshold to be exceeded.

To trigger the vulnerability, a malicious user must supply a specially crafted byte 1-gram feature vector whose values result in the aforementioned wrap-around and subsequent divide by zero exception. Though the same vulnerable code is executed when a user uploads a binary executable, no binary executable large enough to trigger the vulnerability can be supplied since its size would necessarily exceed the STAC input budget for this challenge problem.

All of the other computations, including both cosine similarity classifiers (byte 1-gram and opcode mnemonic 1-gram), byte-level entropy, and objdump disassembly utilities are red herrings.

4.4.6.2 Software Design

Dependencies

- NanoHTTPD (External but automatically fulfilled by Maven)

Modules and Components

Malware Analyzer consists of the following classes. Their interrelationships are illustrated in Figure 49.

- TableMalwareAnalyzer: Contains main(); responsible for starting the HTTP server.

- Server: Instantiates a NanoHTTPD HTTP server and listens for incoming client requests. Received requests result in queries to the Database, the results of which are both passed to the Logger and also returned to the requestor.
- Logger: Records to disk the results arising from the most recent request.
- Database: Non-persistent internal database used to store submitted feature vectors.
- Analysis: Implements methods for determining whether a feature vector characterizes a packed executable and also for computing the cosine similarity between two feature vectors.
- Sample: Wrapper class for maintaining information about a malware sample, including its feature vector and MD5 digest.
- ComparisonResult: Objects of this class maintain similarity score information between two feature vectors; extends *comparable* to allow for lists containing this type to be sorted.
- Dasm: Internal representation of an objdump disassembly of a PE32 binary executable.
- CFG: Internal representation of a function's control flow graph.
- BasicBlock: Internal representation of a function's disassembled basic block.
- DasmHelpers: Contains methods for parsing objdump disassemblies and generating internal disassembly, control flow graph, and basic block representations therefrom
- DasmDatabase: Non-persistent internal database used to store submitted objdump disassemblies.
- X86: Provides methods for making sense of X86 assembly instructions, such as which opcode mnemonics correspond to branch instructions, etc.

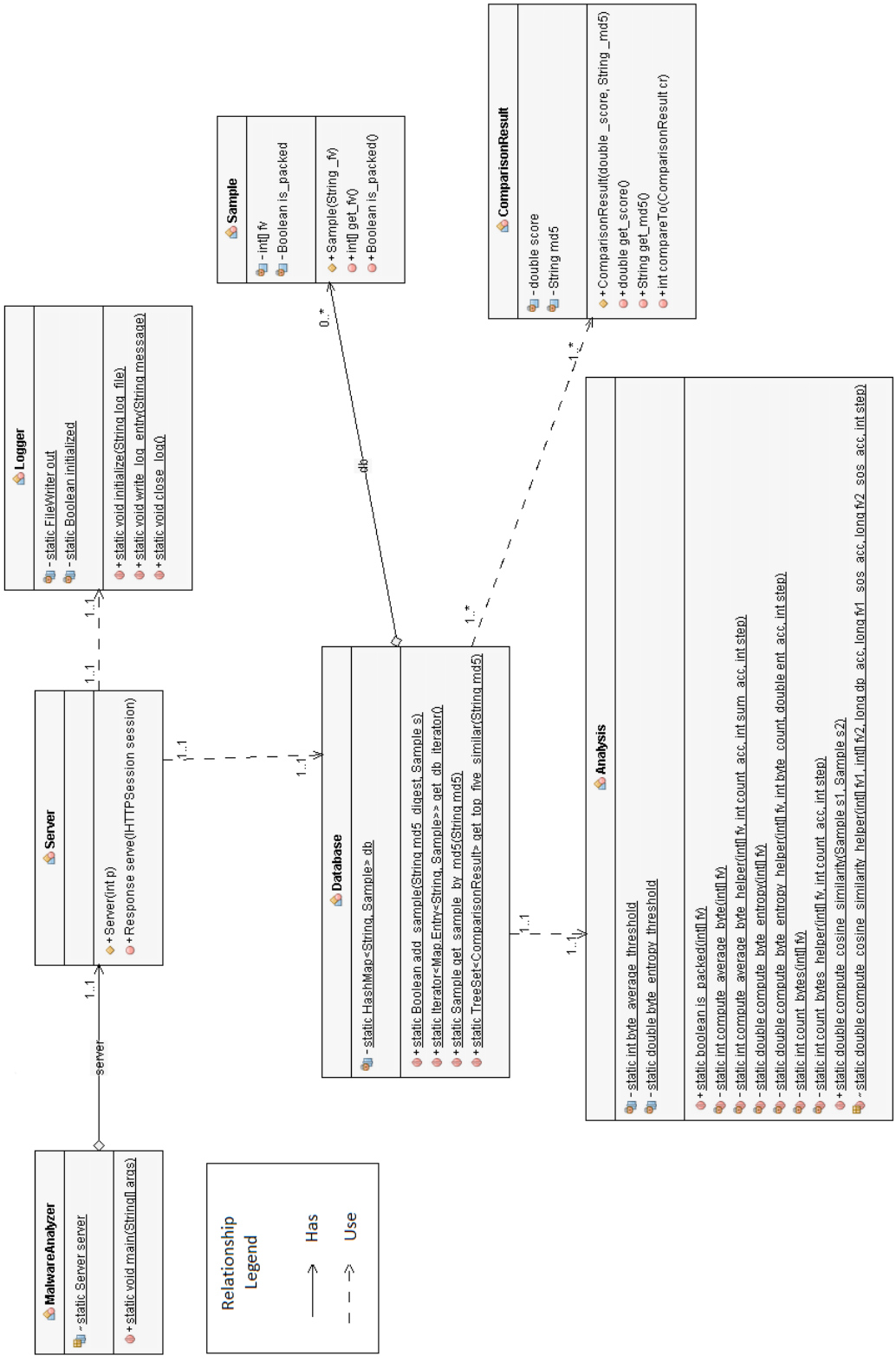


Figure 49. Malware Analyzer Class Diagram

Process Flow

Malware Analyzer services eight types of requests: add, query, binary PUT, add_dasm, list_dasms, get_function_entrypoints, get_cfg, and query_dasms.

add & query:

A user uses an add HTTP post request to add byte information pertaining to a new malware sample to Malware Analyzer's internal database. An add request must include a well-formed feature vector characterizing the user's new malware instance and also said malware instance's MD5 digest. A user submits a query request to retrieve a list of the top five most similar malware relative to a specified instance of malware. The user specifies the query malware instance by its MD5 digest, thus it is assumed that the query malware instance has already been added to the internal database via a prior add request.

When Malware Analyzer's Server receives an add request, it attempts to instantiate a Sample object from the client-supplied information. Both the Server and also Sample's constructor perform sanity checks on the input, and will together reject any malformed input data. The Database is then checked to see if the Sample already exists in the Database. If it does, then the request is aborted. However, if the Sample is not already present, then it is added to the Database. Adding a Sample to the database consists of determining whether the input feature vector characterizes a packed executable (determined by Analysis) and then internally storing the feature vector, MD5 digest, and ascertained packed status. Finally, Server uses the Logger to record the response to a local file and also returns the response to the requesting client.

When Malware Analyzer's Server receives a query request, it uses the client supplied MD5 digest to request similarity information from the Database. The Database uses the supplied MD5 digest to locate the corresponding Sample, and then uses Analysis to compute the cosine similarity between that Sample and every other Sample in the Database. Each computed similarity result is stored in a new ComparisonResult. All of the resulting ComparisonResult objects are stored in a decreasing ordered list. The top five results in the list are passed back to the Server, which uses the Logger to record the computed information to a local file and also returns the response to the requesting client.

Binary PUT:

A user pushes a binary executable (or any file actually) to the Malware Analyzer server via an HTTP PUT request. The server computes the binary executable's MD5 digest and generates the corresponding byte 1-count feature vector. It then passes that information into the same logic that handles a normal *add* request.

add_dasm:

The user uses an HTTP POST request to transmit an *add_dasm* request to the server, containing the disassembly of a PE32 binary executable and the MD5 digest of the original file. The server constructs an internal representation, a Dasm, of the disassembly and adds it to the DasmDatabase. Constructing the Dasm for the disassembly entails parsing the plaintext objdump disassembly, extracting its instructions, identifying all of the functions represented within the instructions, splitting the instructions up into basic blocks, constructing the control flow graphs of those

functions, and generating a feature vector from the counts of the X86 opcode mnemonics specified in the disassembly.

list_dasms:

The user uses an HTTP POST request to transmit a *list_dasms* request to the server. The server responds with the MD5 digests of all of the PE32 binaries whose disassembly has previously been submitted via an *add_dasm* command.

get_function_entrypoints:

The user uses an HTTP POST request to transmit a *get_function_entrypoints* request to the server. Such a request specifies the MD5 hash corresponding to a previously *add_*dasm'd disassembly. If there's a match, then server constructs and returns a list of all the virtual addresses in that disassembly that are targeted by a direct call instruction.

get_cfg:

The user uses an HTTP POST request to transmit a *get_cfg* request to the server. Such a request specifies the MD5 hash corresponding to a previously *add_dasm*'d disassembly and also the virtual address of a particular function's entry point. The server serializes out the internal CFG representation of the control flow graph corresponding to the user's specified function.

query_dasms:

The user uses an HTTP POST request to transmit a *query_dasms* request to the server. Such a request specifies the MD5 hash corresponding to a previously *add_dasm*'d disassembly. The server uses Analysis to compute the cosine similarity between the specified disassembly and all other previously *add_dasm*'d disassemblies, and returns a rank-ordered list of the top five matches.

4.4.6.2.1 Inputs and Outputs

Requests are made to the Malware Analyzer server using HTTP POST / PUT requests. Only the contents of a request's POST / PUT data are used by the server; i.e., there are no URL arguments. All requests are made to http://server_address:8001/

add Request: Adds information pertaining to a new malware sample to the internal database.

Example POST data:

```
add
ba78410702f0cc8453da1afbb2a8b67a
```

```
7904,2779,2065,2207,2637,1564,2193,1753,2453,1420,1676,1427,2009,1406,1947,17
33,2165,1158,1291,1163,1614,1076,1294,1570,1658,990,1100,1111,1430,1049,1225,
1135,2283,1250,1001,1083,1251,1091,918,1089,1214,897,933,1169,1126,1140,1124,
1142,1772,927,1015,1073,1139,932,1038,1032,1323,1184,835,1267,1234,909,899,91
5,1687,1389,1220,1337,1145,1222,1292,1156,1384,975,914,891,1119,1031,1032,991
,1757,947,959,1402,1068,1047,1340,1432,1230,1067,826,884,954,925,991,1045,144
4,1249,865,955,1112,1093,895,827,1875,1053,1493,783,1125,1002,1055,1124,1528,
686,1106,1041,2036,1821,973,1113,1249,1065,633,812,1175,980,1004,964,1770,146
2,1176,1489,1337,1146,1187,878,1254,1475,936,1708,962,1360,834,710,1064,689,7
07,633,738,627,725,725,781,564,642,582,800,556,1083,623,1050,940,651,768,873,
719,688,621,878,661,597,617,827,677,672,697,1219,747,603,636,856,723,854,858,
1119,605,695,839,824,747,900,840,1782,1258,1181,1069,858,608,955,907,951,704,
619,578,768,530,627,776,1094,712,763,696,781,562,792,741,1138,621,794,857,844
,693,860,788,1528,897,741,646,788,440,563,935,1040,641,642,1506,930,784,717,8
51,1523,751,554,620,923,563,896,895,1373,478,647,741,994,668,854,3002
```

Explanation: The POST data for an add request contains three lines: The first line contains only the text “add”, the second line is the 32 character MD5 digest of the executable, and the third line is the feature vector characterizing the executable. The feature vector is a comma separated list of 256 integers, where the 0^{th} entry in the list corresponds to the number of 0x00 bytes in the executable, the next entry corresponds to the number of 0x01 bytes in the executable, and so on up to 0xFF.

Expected Output: OK

***Query Request*:** Retrieve information about the top five most similar instances of malware relative to a specified instance of malware.

Example POST data:

```
query
ba78410702f0cc8453da1afbb2a8b67a
```

Explanation: The POST data for a query request contains two lines: The first line contains only the text “query”, and the second line is the MD5 digest of the malware instance against which similarities should be computed.

Expected Output: Similarity information, including similarity scores and feature vectors for the top five most closely related malware, formatted per the following example:

```
Querying ba78410702f0cc8453da1afbb2a8b67a (Packed)
Top Five Most Similar:
78410702f0cc8453da1afbb2a8b67aba – Score: 0.921 (Unpacked)
10702f0cc8453da1afbb2a8b67aba345 – Score: 0.901 (Packed)
02f0cc8453da1afbb2a8b67aba24af35 – Score: 0.899 (Unpacked)
da1afbb2a8b67aba78410702f0cc8453 – Score: 0.742 (Packed)
67aba78410702f0cc8453da1afbb2a8b – Score: 0.688 (Packed)
78410702f0cc8453da1afbb2a8b67aba = { CSV feature vector for
78410702f0cc8453da1afbb2a8b67aba }
10702f0cc8453da1afbb2a8b67aba345 = { CSV feature vector for
10702f0cc8453da1afbb2a8b67aba345 }
02f0cc8453da1afbb2a8b67aba24af35 = { CSV feature vector for
02f0cc8453da1afbb2a8b67aba24af35 }
```

```
da1afbb2a8b67aba78410702f0cc8453 = { CSV feature vector for
da1afbb2a8b67aba78410702f0cc8453 }
67aba78410702f0cc8453da1afbb2a8b = { CSV feature vector for
67aba78410702f0cc8453da1afbb2a8b }
```

Binary PUT Request: Extracts information from a binary executable and adds it to the internal database.

Example PUT data:

```
<binary_file_data>
```

Explanation: All HTTP PUT requests to the server are treated as Binary PUT requests; all other requests are expected to be HTTP POSTs.

Expected Output: Same as for an *add request.*

add_dasm Request: Parses and constructs an internal representation of an objdump disassembly from a user's input.

Example POST data:

```
add_dasm
01234567890123456789012345678901
windows_exes/notepad.exe: file format pei-i386
Disassembly of section .text:
01001000 <.text>:
1001000: ef out %eax,(%dx)
1001001: 6f outsl %ds:(%esi),(%dx)
1001002: dd 77 17 fnsave 0x17(%edi)
1001005: 6c insb (%dx),%es:(%edi)
1001006: dd 77 25 fnsave 0x25(%edi)
1001009: ba df 77 05 bd mov $0xbd0577df,%edx
100100e: df 77 ab fbstp -0x55(%edi)
1001011: 7a dd jp 0x1000ff0
1001013: 77 42 ja 0x1001057
…
```

Explanation: The POST data for an *add_dasm* request contains an arbitrary number of lines, however the first line must contain only the text "add_dasm", and the second line should be the MD5 digest of the malware instance whose disassembly has been uploaded. An objdump disassembly of a PE32 binary executable should follow on the ensuing lines. Such a disassembly can be generated using the "objdump -d <filename>" command. Note that the server verifies that the disassembly is that of a PE32 binary executable by checking if the first few lines contain the substring "pei-i386" anywhere, which is part of the header generated by objdump when disassembling such a file.

Expected Output: OK

list_dasms Request: Lists the MD5 sums of all previously *add_dasm*'d disassemblies.

Example POST data:

```
list_dasms
```

Explanation: The POST data for a list_dasm request must contain only a single line, containing just "list_dasm".

Expected Output: List of 32-character MD5 digests, one per line.

get_function_entrypoints Request: Lists the virtual address of each function's entry point in a specified disassembly.

Example POST data:

```
get_function_entrypoints
01234567890123456789012345678901
```

Explanation: The POST data for a *get_function_entrypoints* request contains just two lines. The first line contains only the text "get_function_entrypoints". The second line is a 32-character MD5 digest of a previously *add_dasm*'d disassembly, the list of which is available using a *list_dasms* request.

Expected Output: List of virtual addresses (integers), one per line, represented as hexadecimal numbers with preceding "Ox".

***get_cfg* Request:** Obtains the control flow graph of a specified function.

Example POST data:

```
get_cfg
01234567890123456789012345678901
0x1003a39
```

Explanation: The POST data for a *get_cfg* request contains just three lines. The first line must contain only the string "get_cfg". The second line is a 32-character MD5 digest of a previously *add_dasm*'d disassembly, the list of which is available using a *list_dasms* request. The third line is the virtual address of a function's entry point, specified as a hex string.

Expected Output: Listings of the instructions in each basic block found in the specified function. Additionally, the successors for each basic block are listed. A successor of a basic block is another basic block whose execution can immediately succeed that of the former basic block. The output is formatted per the following example:

```
0x10000000 : push %eax
0x10000004 : push %ebx
0x10000008 : push %ecx
SUCCESSORS: 0x10392929, 0x8329328,
-----
0x10000012 : push %edx
0x10000016 : push %ebp
0x10000020 : push %esp
SUCCESSORS: 0x23423423,
-----
```

query_dasms Request: Retrieve information about the top five most similar instances of malware relative to a specified instance of malware, where similarity is determined using X86 opcode mnemonic 1-gram features and cosine similarity.

Example POST data:

```
query_dasms
01234567890123456789012345678901
```

Explanation: The POST data for a *query_dasms* request contains just two lines. The first line must contain just "quest_dasms". The second line should be an MD5 digest that corresponds to a previously *add_dasm*'d disassembly

Expected Output: Rank ordered listing of the top five most similar disassemblies present in the disassembly database, including the MD5 digests corresponding to the them and their cosine similarity score with respect to the user's specified disassembly. The output is formatted per the following example:

```
Querying 01234567890123456789012345678901
Top Five Most Similar:
aa23456789012345678901234567890l – Score: 0.994
bb23456789012345678901234567890l – Score: 0.993
cc23456789012345678901234567890l – Score: 0.992
dd23456789012345678901234567890l – Score: 0.991
dd23456789012345678901234567890l – Score: 0.990
```

Vulnerable Algorithm

When a user submits a request to Malware Analyzer, a response is generated which is both written to a local response file and also returned to the user. For a successful request, any pre-existing data in the response file is overwritten with the new results. However, when a request fails, an error message is instead appended to the response file's files pre-existing data. By repeatedly submitting malicious requests that cause large error messages to be appended to the response file, a user can cause a disk-usage space utilization threshold to be exceeded.

A bug has been purposefully introduced in Malware Analyzer that can cause a large stack trace to be appended to the response file. In particular, by taking advantage of Java's integer overflow semantics, a divide by zero exception can be triggered in the method that computes an executable's average byte value. That byte averaging functionality is implemented in a recursive manner according to the following code, and would be invoked by calling compute_average_byte().

```
int compute_average_byte(int[] fv) // Arg fv is an array of 256 integers,
representing a feature vector.
{
// Call recursive averaging method, starting from first (0th) index into fv, and
with empty count
      // and sum accumulators.
return compute_average_byte_helper(fv, 0, 0, 0);
}

int compute_average_byte_helper(int[] fv, int count_accumulator, int
sum_accumulator, int index)
{
```

```
// Compute and return average = (sum / count) once the entire feature vector
      // has been consumed.
if(index > 255)
{
return sum_acc / count_acc; // divide by zero happens here!
}
// Otherwise, if the entire feature vector has yet to be consumed, update the
total count of
// of bytes and total sum of bytes, then recurse.
int new_count = count_accumulator + fv[index];
int new_sum = sum_accumulator + (index * fv[index]);
return compute_average_byte_helper(fv, new_count, new_sum, index + 1);
}
```

In Java, integers wrap around when an arithmetic overflow occurs, the maximum integer value is $(2^{31} - 1)$, and the minimum integer value is -2^{31}. As such, any sequence of integers that sum using traditional addition to $(2^{31} - 1) + 1 + 2^{31} = 4{,}294{,}967{,}296$ and which are themselves each less than $(2^{31} - 1)$ will sum to 0 according to Java.

By supplying a feature vector in an *add* request that satisfies the overflow criteria, i.e., that each entry in the feature vector is less than $(2^{31} - 1)$ and all of the entries cumulatively sum to 4,294,967,296, the denominator in compute_average_byte_helper()'s division is made to be 0. Since the division occurs on the last recursive step at a stack frame depth of 256, the resulting stack trace arising from the exception contains some 256 lines plus change. This large stack trace arising from the divide by zero exception is appended onto the response file by the Logger.

4.4.7 RSA-Commander

4.4.7.1 Description

This challenge is a peer-to-peer chatting application that utilizes RSA and US Data Encryption Standard (DES) encryption and contains a **time complexity vulnerability**.

RSA Commander uses a messaging protocol that can be abused by an attacker. A sequence of packets contains data that triggers the vulnerability (Figure 50). Blue Teams will need to be able to follow taints from differently structured sequential inputs to see how the bad control flow path is selected. An RSA Commander client cannot send the bad sequence of packets to a peer; an attacker constructs these packets by hand and communicates with a listening RSA Commander victim. Thus, Blue Teams will not be able to infer the vulnerability trigger by reverse engineering the RSA Commander Console that sends chat messages, but will instead need to correctly analyze the protocol parsing in the chat message listener.

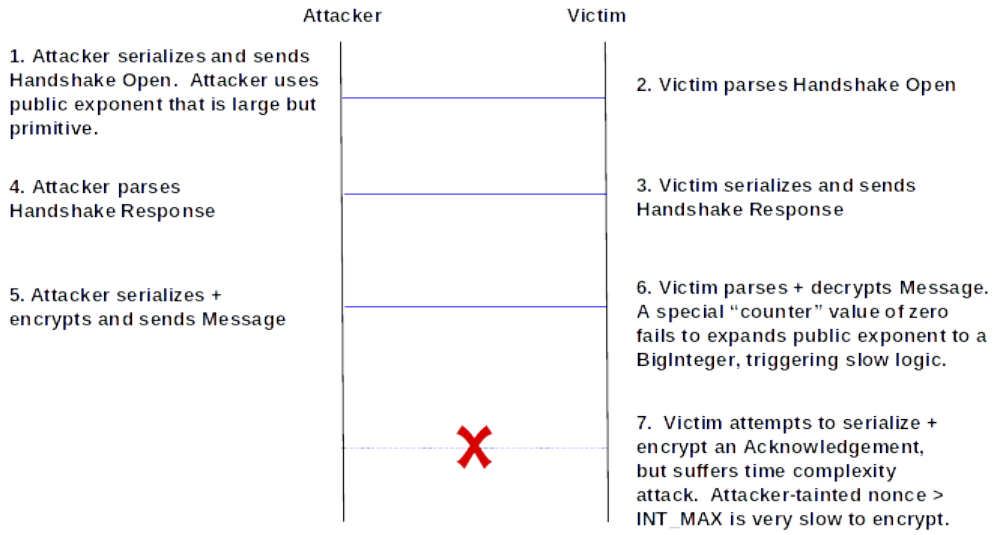


Figure 50. Attack Scenario for RSA Commander

Multiple attacker controlled inputs are needed to exercise the vulnerability. The attacker controls the nonce, counter, and their public key, all of which are needed to trigger the slow control flow path.

First, the attacker's public key selection (in the Handshake) implies a public exponent and a modulus. A custom Integer class takes a slow exponentiation path for "small" (i.e., not BigInteger) exponents; so the attacker can maximize the effect of slow exponentiation by selecting a public exponent of 65537, the typical maximum for OpenSSL. An attacker can then send a carefully crafted Message packet to a victim that triggers slow exponentiation during the Acknowledgement packet construction. (Private exponents are always BigIntegers, so the slow path is never taken when a victim decrypts a message, only when he tries to encrypt a response to the attacker.)

In the Message packet, the attacker also selects a nonce and counter. Special values are needed for both in order to violate the time budget. With respect to the counter, there is a side-effect in the custom Integer class that usually causes the public exponent to be expanded to a BigInteger during comparison with the counter; however, a "bug" causes the expansion to fail when the counter provided in the Message is zero. This means that the attacker must also supply a zero counter for the slow exponentiation path to be taken.

Finally, RSA Commander has a faulty guard (== instead of >=) that allows an attacker to craft a malicious nonce so that the slow exponentiation takes place on a very large base. Ordinarily, all nonces generated by the RSA Commander client are less than INT_MAX, since the nonces for new packets are simply incremented by RSA Commander and the faulty guard will wrap the nonce back to zero when INT_MAX is reached. However, the attacker can craft a packet with a BigInteger nonce greater than INT_MAX (but less than the modulus) and break the guard.

During the attack, the victim's Listener thread is blocked, but not the Console thread, meaning that the victim can send chat messages but not receive them

4.4.7.2 Software Design

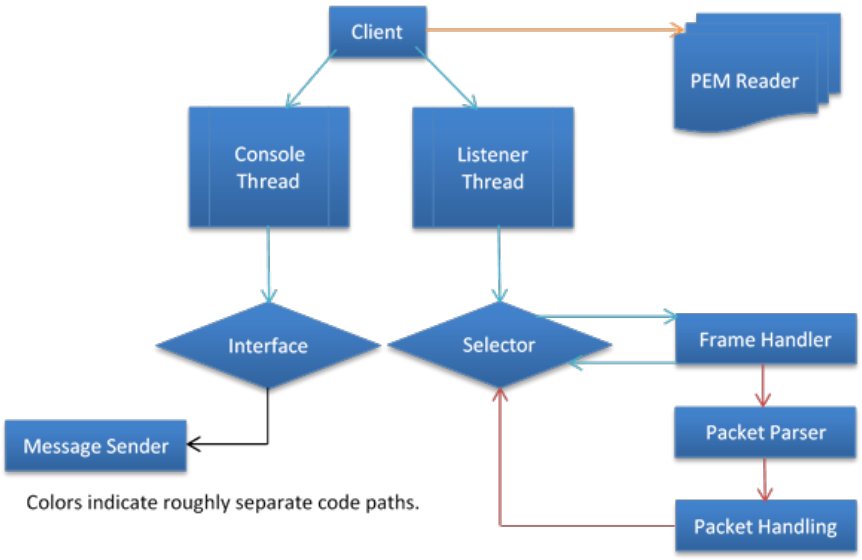


Figure 51. Overall Challenge Design and Structure

Client (Client.java)

The Client contains classes that parse command line options and starts up the threads that run the console and the listener threads. The client also reads the private key from disk and initializes storage structures that share the screen, data, and messages between the two threads.

Console (Console.java)

The console thread contains a read-execute-print-loop that reads commands and messages and transfers read messages to the message sender. See the section about Client and Interface for more.

Message Sender (Console.java::handleUserInput; line 89)

The message sender runs the client side of the handshake, message sending, and session termination. This module relies on information gained from client and the console to encrypt the message.

Listener (Communications.java::listen; line 78)

The listener thread sets up the sockets and structures required to track the sessions.

Selector (Communications.java::listen; line 93)

This part isn't so much a separate class as a major sub-component of the listener. This part is a loop that understands how to read multiple sockets from the same thread and delegate the data or control structures deeper into the frame handler / packet parser / packet handler.

Frame Handler (Session.java::handle; line 53 and Session.java::readPacket; line 100)

The frame handler is responsible for buffering data from various packets, starting sessions for new packets, and triggering packet parsing and handling when it is ready.

Packet Parser (Implementations of Handler.java::runPacketParser; line 102)

The packet parser converts buffered packets into packets for the packet handler.

Packet Handling (Implementations of Handler.java::handlePacket; line 109)

The packet handler extracts information from the packets and figures out what the proper action should be and sends messages to the screen for the user to see.

PEM Reader (OpenSSLRSAPEM.java)

The Privacy Enhanced Mail (PEM) Reader is responsible for reading OpenSSL private keys from disk. This has two guards built into it to prevent keys that are excessively large from entering the system. The PEM Reader contains inner-classes that; create self-expanding integers that the vulnerability relies on, and that create other DER (Distinguished Encoding Rules x509) decoded representations. OpenSSL PEM files are a restricted ASN.1 format called DER. DER is used as a highly compatible machine interchange format used to describe RSA keys, x509 certificates, Diffie-Hellman (DH) exchanges, and practically any other type of data which may pass to machines with any endianess. This portion of the application adds complexity to test the blue team's tools. For more information, please read the PEM Format addendum.

INTEGER class (OpenSSLRSAPEM.java::INTEGER; line 518)

INTEGER is a concrete implementation of an automatically expanding integer. It will attempt to use the primitive integer when the number will fit into it, but will automatically expand to BigInteger when the value is too large. The important method in this class is the modPow method that tries to determine which method of calculation will result in the smallest final space usage. Unfortunately, this function selects a very slow computation when the exponent is a primitive integer and a faster approach when the exponent is a big integer. The next two diagrams show the RSA equation and the INTEGER method used to calculate it. See the Encryption addendum for

more information on these values. The Dangerous self-expanding integer modPow implementation is shown below.

```
public INTEGER modPow(INTEGER pow, INTEGER modulus) {
      if (pow.isBig()) {
                  return new INTEGER(this.internalBig.modPow(pow.internalBig,
modulus.internalBig));
      } else {
            return new INTEGER(this.internalBig.pow(pow.internal).mod(modul
us.internalBig));
      }
}
```

Protocol / Packets

The budgeting of the RSA Commander challenge requires an intimate knowledge of how the individual stream components are structured. The flow of packets through the system will be detailed in the next section. The maximum sizes of the packets are listed in the heading of each packet type and each field is detailed with its purpose and the range of sizes / values it can hold. The possible values of the pType field are listed next to the “Packet Type” bullet in the details section of each packet.

This next diagram is the packet flow during both normal and attack conditions:

Protocol Flow

```
C1 | Handshake Open ------------------------> | C2    [Type: HandshakePacket]
C1 | <---------------------- Handshake Response | C2    [Type: HandshakePacket]
C1 | Message ---------------------------------> | C2    [Type: RequestPacket]
C1 | <-----------------------------Acknowledge | C2    [Type: RequestPacket]
C1 | Terminate -------------------------------> | C2    [Type: RequestPacket]
```

Figure 52. Flow during a normal send message operation

Handshake packet structure and fields:

```
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
128
|=============================================================================================|
|pType|- Request Public Key Fingerprint ----------------------SHA256--------------------------|
|---------------------------------------------------------------------------------------------|
|-----|-flg-|-- pSize --|--------- Public Exponent (variable length) ~~ -------- ~~ -------- ~~ |
|-- mSize --|--------- Public Key Modulus (variable length) -------- ~~ -------- ~~ -------- ~~ |
|=============================================================================================|
```

HandshakePacket (Max 106 Bytes)

- Packet Type (HANDSHAKE_OPEN)
    - This is the packet type and is a single byte with only a few possible values.
- Fingerprint
    - Fingerprints are SHA256 digests of the key with a fixed width of 32 bytes.
- Flags
    - The flags associated with the handshake consist of another single byte and each bit has meaning.
    - The currently used flags are: Return service (key request), Handshake request, and Handshake accepted.
- Public Exponent Size
    - This is the size of the public exponent. The size field here is a fixed width of two bytes.
- Public Exponent
    - This is the public exponent. The victim client will limit the exponent to the range [3, 65537] so the budget allocation for this field should be the number of bytes required to represent the maximum size of the public exponent. The maximum size of the Public exponent that will be accepted (not crash the app) is three bytes. This is the normal OpenSSL case.
- Modulus Size
    - This field is sized identically to the public exponent size field, two bytes.
- Modulus
    - The modulus has a range of (0, 512] bits, which produces a budget limit of 64 + 1 sign byte.

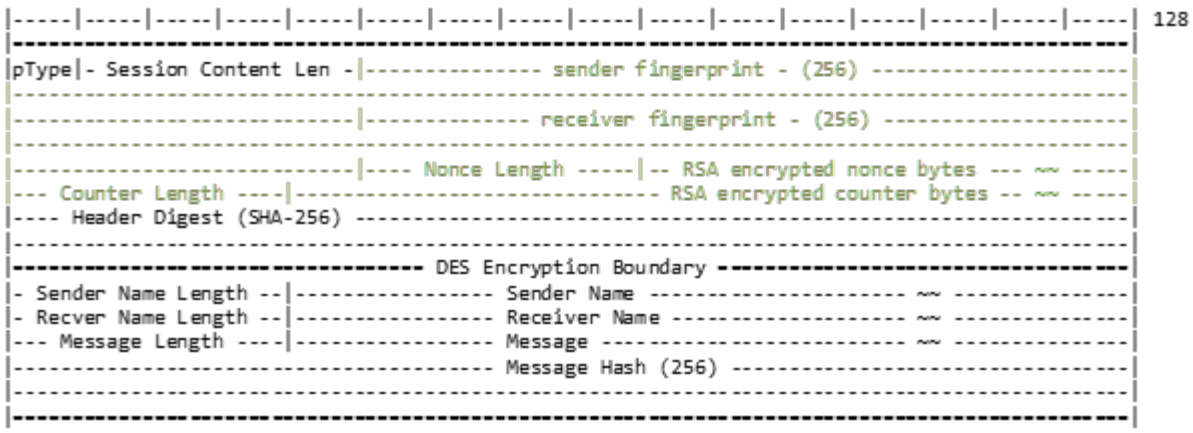


Figure 53. Request (Message/Termination) Packet Structure and Fields

RequestPacket (Max 4405 Bytes)

- TablePacket Type (REQUEST_MESSAGE or REQUEST_TERMINATE)
  - This is the packet type and is a single byte with only a few possible values.
- Session Content Length
  - This is the content length of the encrypted content and is a fixed width integer of four bytes.
- Sender Fingerprint
  - Fingerprints are SHA256 digests of the key with a fixed width of 32 bytes.
- Receiver Fingerprint
  - Fingerprints are SHA256 digests of the key with a fixed width of 32 bytes.
- **Encrypted** Nonce Length
  - The encrypted nonce length is a minimum width of four bytes and the value will never exceed 65.
- **Encrypted** Nonce
  - This an integer in the range [0,modulus), although the guard states [0, INT_MAX). Width: 65 bytes max.
- **Encrypted** Counter Length
  - The encrypted counter length is a minimum width of four bytes and the value will never exceed 65.
- **Encrypted** Counter

    - This an integer in the range [0,modulus), although the guard states [0, INT_MAX). Width: 65 bytes max.
- Header Digest
    - This is the SHA256 of the green colored headers (decrypted nonce / counter) and takes 32 bytes.
- Sender Name Length
    - The length of this field is fixed at four bytes, although under normal conditions the value will be [0, 62].
- Sender Name
    - The length of this field is of variable length, where the maximum will be 62 bytes.
- Message Length
    - The length of this field is fixed at four bytes, although under normal conditions the value will be [0, 4096].
- Message
    - The length of this field is of variable length, where the maximum will be 4096 bytes.
- Message Hash
    - This field has a fixed width of 32 bytes.

The arrow bullets indicate fields that are not DES encrypted and the diamond bullets indicate fields that are DES encrypted using the DES-CTR (CounTeR) scheme. Termination uses this same packet but the packet shouldn't have a message and therefore the nominal length is 309 bytes, but this packet should be ignored for the budget computation.

Walkthroughs

Message Flow – Attacker (Client) Perspective

This walkthrough is a demonstration of the code paths that will be taken when the attacker (client) is sending a message to a peer. We will treat the peer in the next walkthrough section.

The code will enter through the Main and Client classes are where keys are loaded and console input and reception threads are started. For the most part, the code in the reception threads will not be used at any time by the client in send mode. There are notable exceptions to this rule, the parsers and handlers, and their codes are shared between the send and receive sections of the client application for obvious reasons.

The Main/Client classes create a new thread for the Console class to run in and pass control to it. The Console class is the input manager that a chat user uses in order to send messages; it is complimented with the Screen class, which the program delivers text to the user. The Console class is purely input / execution and the Screen class is purely output.

When the client receives a command from the user telling it to send a message to a hostname + port combination, it asks for the name of the receiver and the message to send to the receiver. The responses are stored for later use.

Communications, which is a central class but has distinct methods for sending messages verses receiving them, contains a sendMessage method that begins by opening a socket to the remote peer.

The sendMessage method then proceeds to building a HANDSHAKE_OPEN packet. This is as simple as calling the default constructor on the HandshakePacket class and filling in the public key information fields (Fingerprint, Public exponent, Modulus, and their sizes if necessary). The public exponent sent by the attacker in a Handshake Open will eventually cause the victim to select a slow exponentiation. The sendMessage method sets two flags: Return Service and Handshake Request. These flags indicate to the peer that the client is beginning a handshake and soliciting the peer for their public key information.

After sendMessage constructs this packet, it will serialize it using the getParser method of the HandshakePacket class and calling serialize on the returned PacketParser. The result of serializing packets is a bytes array that can be written to the output stream on the socket that sendMessage opened.

The peer will receive and respond with the result of the handshake operation. Since we requested the peer's public key information (Fingerprint, Public exponent, and the Modulus) the peer will include it in the handshake response packet. In order to read this structure, sendMessage instantiates a HandshakeHandler and passes the stream to it until the packet is ready to be parsed and returned to the user.

Handlers in this challenge take an input stream and a packet buffer. They then read bytes from the stream and store them in the provided packet buffer for later use during the deserialization procedure.

Once a packet is fully received, then sendMessage method will instantiate a blank HandshakePacket class and obtain a new parser from the getParser method. The parser has a parse method, to which the packet buffer is passed. Packet parsing reads the values from the packet buffer and converts them into the form that the HandshakePacket requires. Finally the information is set on the packet object and returned to the sendMessage method.

The sendMessage method now has the information required to send an encrypted Message to the peer, including the peer's public key information. In order to do that, sendMessage utilizes the RequestPacket's newMessage method.

Most of the information required for a packet has already been discussed, but three fields of the message packet are still missing. These are the nonce, the counter, and the header digest fields.

The nonce is a random number in the range [0, peer modulus), this nonce will be incremented by the peer for use in the termination packet. The indication for the client is that the peer received the message by checking the nonce and header digest against what the client expects the digest to be. (In the challenge, we don't implement the typical functionality that the client resends the message if the returned nonce is incorrect.) The nonce is also used as the session key for the DES encrypted

data later in the packet. For the purposes of this challenge, the nonce will be truncated and/or expanded in order to fit in the DES key vector.

The counter is another random number in the same range [0, peer modulus), but this value will be truncated or expanded in order to fit the DES input vector.

The third non-obvious field is the header digest, which is a SHA-256 hash that contains the fields: Session content length, sender and receiver fingerprints, encrypted nonce length, plaintext nonce, encrypted counter length, and plaintext counter. The receiver will recalculate this field and check it against the stored copy to verify that the packet headers haven't been tampered with.

Once all of these fields are calculated and inserted into the RequestPacket, the sendMessage method will serialize the message and send it shortly followed by a termination packet telling the receiver that the client is done with the connection.

During the serialization stage, the nonce and counter are encrypted using the public key information provided by the peer and their decrypted forms are used as the session key and the counter for the DES- encryption of the fields within the DES encryption boundary lines from the RequestPacket layout in the previous section. The attacker's selection of nonce and counter in the Message will cause a time complexity attack when the victim tries to encrypt and serialize and Acknowledgement.

Message Flow – Victim (Peer) Perspective

This walkthrough covers the code paths taken when the victim receives a message from a client.

The Client class starts a thread running the Communication's listen method. The listen method sets up a selector that uses the underlying epoll or select system calls to multiplex socket events onto a single thread listening thread. A listening socket is bound to the listen address and port and is then attached to an event trigger in the selector. The event we are listening for on this socket is the acceptable event, which indicates to us that there is a connection waiting to be accepted. After this selector is set up we begin a loop-to-infinity waiting on socket events.

If a socket event occurs the select call returns and the handler inside the loop checks to see if it is one of two events. The first event is the acceptable event for which the handler accepts the connection, constructs a new Session object, and registers the new connection and session for readable event notifications. The second event is the readable event after which the handler passes control to the Session object.

The listen thread will be terminated when the user terminates the application via the exit console command or ^C on the console. This thread is the target of the attacker. The attacker will cause a handler to consume excessive time and during the attack the client will be able to send messages, but will be unable to receive any response messages from other users.

Let's now assume that a peer has opened a connection and we have accepted it. Now we have a new connection with a brand new session attached to it. Sessions begin in the OPEN phase, which allows them time to set up packet buffers and various details like remote addresses and storing streams before any data is received. After the OPEN phase completes the setup phase it immediately switches to the HANDSHAKE_REQUEST phase, and begins to read all of the available data that it can in the Session method readPacket. The actual reading of the data is

performed by the handler, which is defined by the session phase. For example: If the Session is in the HANDSHAKE_REQUEST phase, then the handler passed to readPacket will be the HandshakeHandler.

If there is not enough data, the session will remain in the same phase but the handler will return a HANDLER_STATE of WAITING until the selector is triggered by another readable event and the handler decides it has enough data to un-marshal the packet. If the handler returns a HANDLER_STATE of FAILED or CLOSE, then the Session will be destroyed possibly with an error posted to the screen. Alternatively, if the handler returns a HANDLER_STATE of DONE, then readPacket will invoke the handler's handlePacket method with the packet buffer as an argument.

Handlers all have the method handlePacket, which are the PacketHandlers described in the second session on the overall challenge design. These handlers internally do some checking to make sure the session is expecting the current packet handler to run and then they proceed to run the packet parser and take an action in response to the received packet.

Each packet parser functions in the same way described in the unsolicited flow in the previous walkthrough, but the packet handler is a unique piece of code that knows how to accept handshakes and acknowledge and display messages from a foreign instance. These next bullets highlight the responsibilities that each handler has.

- HandshakeHandler
    - Expects a Handshake packet with the Handshake Request flag and responds with a Handshake packet with the Handshake Accept flag or terminates the session if there was an error or guard violation
- RequestHandler
    - Expects either a Message packet or a Terminate packet and responds with an Acknowledge packet or by terminating the session, respectively. It will also terminate connections on errors or guard violations.
    - The Acknowledge response serialization uses attacker controlled values in the broken INTEGER modPow method during the encryption of the nonce and counter variables.

The attacker controlled values cause a time complexity attack as follows:

Sender Public Key

The receiver peer has no option but to accept this value, so there is a guard stating that this key, or more accurately its modulus, should never exceed 512 bits (512 bits being the default OpenSSL key size). The maximum public key value is 65537, which is not a BigInteger.

The vulnerability cannot be triggered when the victim decrypts an attacker packet because a private exponent will always trigger the fast exponentiation. The vulnerability only occurs during the encryption of the victim's response.

Counter

When counter is parsed, it gets compared to the public exponent supplied by the attacker (see pseudocode below). This comparison has a side effect which expands the public exponent to a BigInteger, except when the counter is zero. Thus an attacker-supplied counter of zero is required in order to activate the slow exponentiation.

```
if (counter.compareTo(0) >= 0 &&
counter.compareTo(owner.getReceiverKey().getPem().getPublicExponent()) < 0) {
continue parsing;
} else {
throw new PacketParserException("Failed to perform RSA decryption");
}

public int compareTo(Object o) {
INTEGER oo;
if (o instanceof BigInteger) {
oo = new INTEGER((BigInteger) o);
} else if (o instanceof INTEGER && this.compareTo(0) == 0) {
System.out.println("Causing expansion on: " + o);
oo = new INTEGER(((INTEGER) o).getInternalBig());
if (oo.compareTo(0) == 0) return 0;
} else if (o instanceof Integer) {
oo = INTEGER.valueOf((int) o);
} else if (o instanceof Number) {
oo = INTEGER.valueOf(o.toString());
} else {
oo = (INTEGER) o;
}
}
```

Nonce

The nonce, as the base of exponentiation, has a very large impact on the runtime of the modPow operation performed on the small exponent path through the INTEGER.modPow method. Values of the nonce that are larger than INT_MAX, plus a public exponent of 65537, will violate the budget. There is a guard on the nonce that should ensure that nonces larger than INT_MAX are wrapped around to 0, but instead the guard only states that nonces equal to INT_MAX should be set to 0. This works between two RSA Commander clients (because they generate packets only with sequential nonces), but not when the attacker crafts his own packets.

4.4.8 SpellCorrect (Tweeter)

4.4.8.1 Description

The spelling correction challenge is a twitter like clone that allows users to post tweets and offers to correct their spelling as they are being posted. There are two vulnerabilities in the challenge:

1. **Time-Complexity**: The spelling correction algorithm itself generates an incredible number of edits, which consumes memory and time. A single tweet message results in the time budget being exceeded.
2. **Space-Complexity**: The twitter clone allows new user registrations and this causes a new unique profile picture to be generated for that user. These user images are stored in memory for quick access, but this causes significant memory usage when a hidden parameter is used to increase the image size during registration. This vulnerability requires a sequence of user registrations to exceed the memory consumption budget.

4.4.8.1.1 Spelling Correction (Time Complexity)

The spelling corrector will take a message of no more than 140 characters and attempt to correct all of the spelling errors in the message before forwarding it to a tweet service. This is performed by three major tasks:

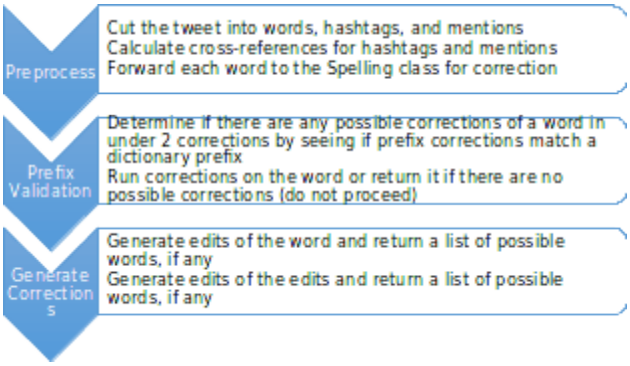


Figure 54. The Major Tasks Involved with the Time-Complexity Vulnerability

The first stage works on the tweet as a whole, while the other two stages work on individual words at a time.

4.4.8.1.2 Computing spelling corrections: "Edits"

Spelling edits are calculated in four stages: Removals, additions, transpositions, and replacements. In all of these, n represents the number of characters in the input string (think word for now). Edits are generated in **both** the Generate Corrections and the Prefix Validation stage. Each edit is added to a set in order to prevent the same edit from consuming more space via more than one of these stages. Edits-of-edits (or k-edits) are generated until we reach the kth round of edits or at least one word is found in the edits set. (In the application, k is set at 2 and not modifiable by the user.) It is important to note that the Prefix Validation and the Generate Corrections stages do not share any data and edits are not cached in any way.

These are the space (and therefore time) costs of each of these different types of edits:

- Removals
  There are n-1 different possible outcomes for removing a single character.
- Additions
  There are 26 possible characters to insert and n + 1 positions in which to add a character.
- Transpositions
  There are n-1 possible character swaps.
- Replacements
  There are 26 possible characters and n positions in which to replace a character.

4.4.8.1.3 Pseudo-code for generating edits

```
private Set<String> generate1Edits(String word) {
      Set<String> result = new HashSet<>();
      // Removals
      for (int i = 0; i < word.length(); ++i) {
            result.add(word.substring(0, i) + word.substring(i + 1));
      }
      // Transposition
      for (int i = 0; i < word.length() - 1; ++i) {
            result.add(word.substring(0, i) + word.substring(i + 1, i + 2) +
           word.substring(i, i + 1) + word.substring(i + 2));
      }
      // Replacement
      for (int i = 0; i < word.length(); ++i) {
            for (char c = 'a'; c <= 'z'; ++c) {
                 result.add(word.substring(0, i) + String.valueOf(c) + word.substring(i + 1));
            }
      }
      // Addition
      for (int i = 0; i <= word.length(); ++i) {
            for (char c = 'a'; c <= 'z'; ++c) {
                  result.add(word.substring(0, i) + String.valueOf(c) + word.substring(i));
            }
      }

      return result;
}
```

4.4.8.1.4 Prefix Validation

For each prefix of the misspelled word, edits are generated until an edit is found which corresponds to a prefix of a valid word in the dictionary. This will also loop over the various edit distances so if there is no possible word from all 1-edits, then the prefix corrector will continue to 2-edits. This

means for impossible words, there will never be a full word correction calculation. Impossible words will reach the end of the pseudo-code having never found a possible prefix and will never enter the Generate Corrections stage. The worst case of this part of the algorithm happens to be a long word missing a few initial chars.

The following is an example of Prefix Verification on the word ‘Xiffer’ using a dictionary that **only** contains the word ‘Differ’. The ellipsis mean that some steps were skipped for clarity but do not impact the example in any way.

Example (Xiffer -> Differ, Single word in the dictionary):

```
Prefix length 1: (“x”)
Is this a prefix of a word in the dictionary: No
...
Generate the next substitution and verify: (edit: “z”, “z” is NOT a prefix)
...
Generate the next substitution and verify: (edit: “d”, “d” is a prefix)
Stop
Prefix length 2: (“xi”)
Is this a prefix of a word in the dictionary: No
...
Generate the next removal and verify it is a possibility: (edit: “i”, is NOT a prefix)
...
Generate the next substitution and verify it is a possibility: (edit: “di”, is a
prefix)
Stop
...
All prefixes up to the word length
...
Prefix length n: (“xiffer”)
Generate some edits and verify them...
Generate the edit that finds “differ” and verify: (edit: “differ”, prefix/match of
differ)
Stop
```

4.4.8.1.5 Pseudo-code (not optimized) for prefix verification

```
public final boolean possiblePrefixes(String prefix) {
      if (possiblePrefixes.lookup(prefix)) return true;
      for (int i = 1; i < MAX_EDIT_LENGTH; i++) {
           lastEdits = Set({prefix});

            for (int k = 0; k < i; k++) {
                 kEdits = Set();
                  for (String lastEdit in lastEdits) {
                        kEdits.addAll(edits of lastEdit);
return true if any possiblePrefixes.lookup(edit) == true
                  }
                  lastEdits = kEdits; // replace lastEdits with kEdits so we can try k+1 edits
            }
      }
      return false;
}
```

4.4.8.1.6 Generate Corrections

Word correction edits are calculated in the same way as edits of a word prefix, but they are performed on the entire word and all edits are generated for the entire word instead of stopping at the first. This happens for a number of times and the input to the each round is the output of the previous round. If the first edits do not contain a word, then the edits of each edit of the word are calculated. This process continues until either, the third round been performed, or one or more valid suggestions are returned.

Example:

```
Edit distance 1:
Generate all possible edits of “dixxer”:
Is there a valid word with one of these edits? No
Continue
Edit Distance 2: (Edits of all edits from “edit distance 1” block above)
Generate all possible edits of all the edits from before
Is there a valid word with one of these edits? Yes, Differ.
Stop
```

4.4.8.1.7 Pseudo-code (not optimized) for generating corrections

```
for (int i = 0; i < MAX_EDIT_LENGTH; i++) {
     lastEdits = Set({word});

      for (int j = 0; j <= i; i++) {
            Map<Integer, String> corrections = new HashMap<>();
            lastEdits = generatePlus1Edits(lastEdits);
            for (String edit : lastEdits) {
                  if (dictionary.containsKey(edit)) {
                        corrections.put(nWords.get(edit), edit);
                  }
            }
            if (corrections.size() > 0) {
                  return corrections;
            }
      }
}
```

4.4.8.1.8 Time Complexity Worst Case

Longer words, as can be seen in the pseudo-code and the example fragments, will produce more edits. A primary concern here is that the set of all 4-edits can be much larger than the available memory, and 3-edits exceed a reasonable time budget often, so we are limited to MAX_EDIT_LENGTH=2.

A witness for this is the word electroencephalography, which, if miss-spelled exactly MAX_EDIT_LENGTH (from the pseudo-code) times at the beginning of the word, will force the

generation of all possible edits before repairing the word. For example, XXectroencephalography or ectroencephalography, these will both pass the prefix test because a small number of edits are generated to correct for the XX -> el or missing el from the beginning of the word. Then, they will proceed to the word correction stage where all possible edits of the large word are created, but do not contain decent repairs. Finally, the second edit round occurs meaning every edit of the already generated edits are created and the word from the dictionary is added to the candidates list.

4.4.8.1.9 User Avatars (Space Complexity)

The space complexity part of the challenge application is the way that user images are stored. Each individual image is small, but a hidden parameter of the request to register a user allows an attacker to change the size of the image produced between three image sizes, 128x128, 256x256, and 512x512. It is not used from the web interface and models dead-code, as it would occur in the real world. This leads to excessive memory usage when thousands of users are registered all at once while exercising the unused parameter.

4.4.8.1.10 Space Complexity Worst Case

The worst case here is the smallest message producing the largest output image. The image sizes are the same for every image, 512x512 in the worst case, so the worst cases are messages with the fewest number of characters and largest number of output pixels.

4.4.8.2 Software Design

Twitter is a small message sharing service on the Internet. Each message is limited to 140 characters, but are allowed to reference other users or current affairs via their '@' and '#' notations. The service will provide a selection of corrections to the users who are then going to select the appropriate correction and finalize tweets for display to other users.

All of the major components are described below, but their names may not be the same in the code.

There are a few components to this:

- Hash-tags and mentions. These are basic relational database tables.
- Twitter clone. This basic web-app models itself after Twitter.
- Twitter backend. This manages the relational databases. (Users, affairs, and messages)
- The spell correcting service that inserts itself between the tweeting frontend and backend correcting spellings before they become tweets

The databases

The database engine is H2, which is a Java relational database similar to SQLite. Hibernate and Java Persistence API (JPA) are used to build the Object-Relational Mapping (ORM).

The web site

This will be a web site run from a tomcat/jetty instance through spring-boot and will be a basic MVC page that mimics twitter.

The backend

This will be a basic service, which performs all of the mappings from user to tweets, etc. This system is for managing the tweet data, so the service will remain as simple as possible so that it doesn't pick up unintended vulnerable sections of code.

The spell corrector

This component uses only java builtins and the Trie class and runs in the manor described in the previous section. This will receive requests from the tweeting backend just before the backend starts processing the tweet. Think of the placement of this module as a frontend to backend shim or backend preprocessor. See Error: Reference source not found for a diagram of what the spell corrector looks like.

The classes in the spell corrector are enumerated here:

1. WelcomeController:
   A dummy for the aforementioned tweeting frontend. This triggers spell-correcting jobs.
2. SpellCheckingService:
   The spell corrector itself. The String arguments other than addChecker are session/job/task IDs. The String argument to addChecker is the actual message being corrected. Once the jobs are finished, the isCheckerRunning(id) call returns false and the getText and getResult calls return the original text and the corrected copy of the identified job.
3. SessionIdentifierGenerator:
   This provides individual Tasks with their session/job/task IDs.
4. Task:
   Tasks represent a currently running checker job. Each job splits the input message up into individual words and passes each through the Spelling class for corrections in order. Tasks are multithreaded, but word corrections inside tasks are sequentially corrected, this allows many tweeters to tweet, but doesn't allow them to use up all of the threads by sending messages with 70 single letter words in them.
5. Spelling:
   This class actually performs the individual word spelling corrections as described in the Challenge Idea section of this document.
6. TrieNode:
   The spelling corrector uses this Trie as the prefix tree for discovering if strings are prefixes of words in the dictionary via the IMatcher classes.

    a. TrieEdge is the internal edge representation of the TrieNode class.

7. IMatcher:
   An interface for matching words and prefixes.

   a. WordMatcher:
      A prefix matcher that returns true for active and true for success when the word is a substring of a word in the constructor provided dictionary Trie.

   b. FalseMatcher:
      A dummy matcher that always returns false for active and success is undefined.

The avatar generator

This is only made up of three basic classes, Avatar, AvatarService, and AvatarController. The Avatar class is the object that holds the image data associated with the avatar and creates the image using the visual hash implementation, “vash”. The AvatarService holds the avatars for the users in the system so they can be delivered to browsers without regenerating the image. Finally, the AvatarController allows browsers to query the AvatarService for avatars of known users. If the user is not available, the service will return a default image. Avatars are created when a new user registers or when the system comes up, but the challenge doesn’t allow the server to be restarted as a valid user input.

The culmination

The spelling checker is a component of the tweeting service provided as the challenge. The entire challenge will look something like the diagram below. The entry point is the TweetingController class, the controller behind the mock twitter webpage. It will look similar and function similarly in nature to the real twitter although with a somewhat reduced functionality for the purposes of the STAC challenge. In other words, a set of users (pre-defined or added via another controller) is added to the service and then they are able to send messages to the twitter service that people can query via the TweetingController.

The process of viewing tweets is trivial. You look up the ones you are interested in and display them using the TweetingController and TweetingModel.

The process of submitting a tweet is as follows:

1. TweetingController receives a tweet from an author.
2. TweetingController hands the tweet off to the spelling corrector.
3. TweetingController hands the spelling corrector job number off to the TweetingService.
4. The SpellCheckingService performs the corrections on the tweet.
5. When a correction job completes (polling in the TweetingService) it is saved via the TweetingModel to the database.

Libraries

The whole application is built on top of a spring-boot + hibernate + H2 application, but all of the libraries are internal to the jar so it is a single jar web application.

Spring-boot uses inversion of control (IoC), and will exercise the blue team's abilities to model included library code and form sound analysis in a real-world webserver code scenario. Spring uses introspection and reflection to achieve IoC, but the application code uses neither of the two internally so the call graph of the application code isn't disturbed.

Hibernate also uses introspection and reflection to achieve its goals, but the introspection doesn't interfere with the call graph of the application code and the reflection adds repository code at runtime, through a java interface, in the application code, that the blue teams will easily be able to model. Modeling database access code from H2 / Hibernate should be easier and faster than attempting to analyze database access code.

There is no need to analyze any of those code bases belonging to Spring or Hibernate as long as the blue team tools are capable of forming the graphs they need from the class files in the: com.tweeter, util, vash, and ec.util packages while excluding and/or modeling the library/IoC code.

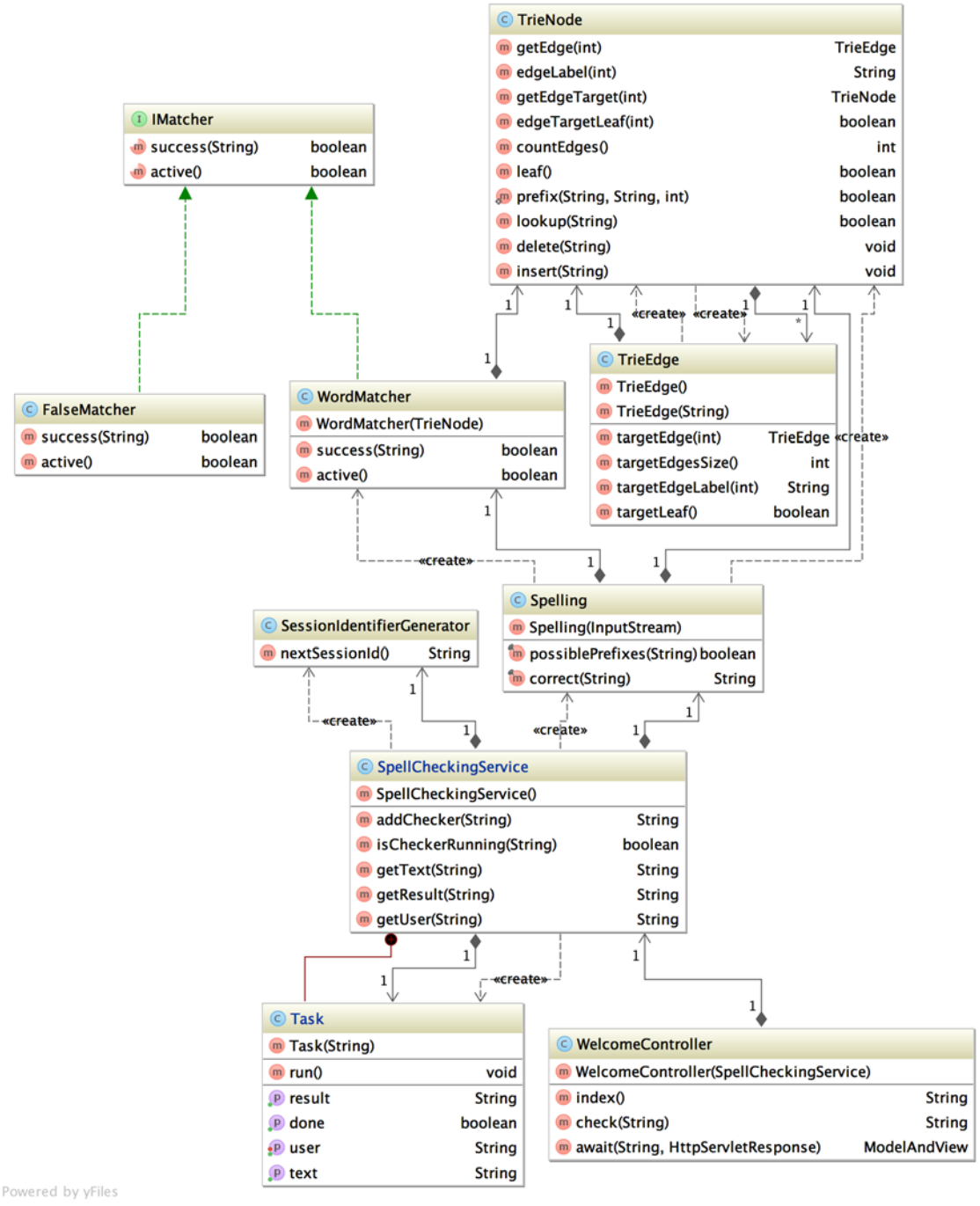


Figure 55. Class Diagram

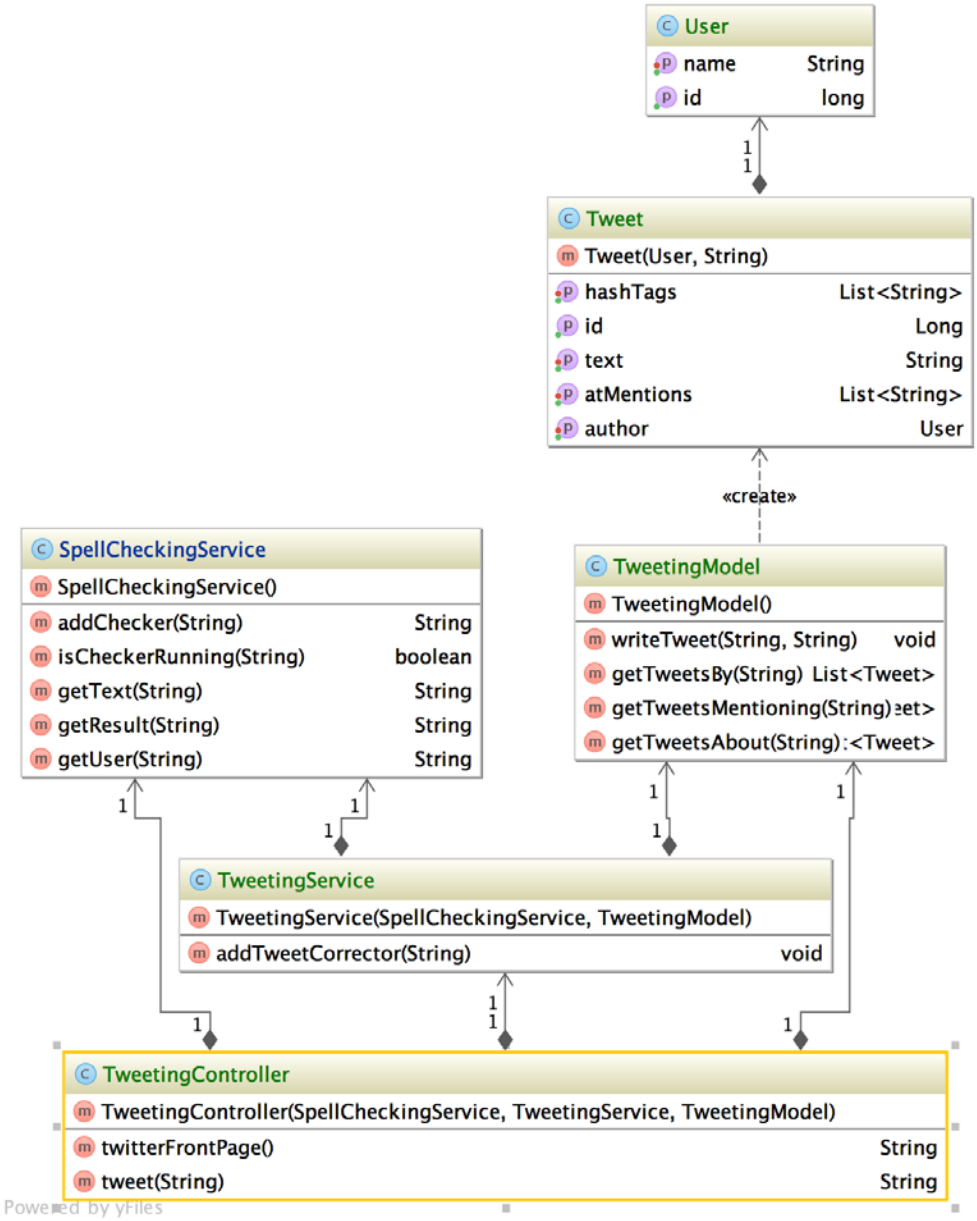


Figure 56. Class diagram (cont.)

4.5 Engagement 2 Challenges Programs

4.5.1 BTreeChallenge (Law Enforcement Database)

4.5.1.1 Description

This challenge is a timing side channel attack on an application that stores information, both public and private, in a b-tree data structure. The stored information represents the employment identification numbers of law enforcement agents. Some of the identification numbers are prohibited from being viewed publicly because the agent is operating in a clandestine manner. It is these restricted identification numbers that represent the secret data which is revealed by the side channel. The side channel reveals information about these identification numbers by emitting different timing profiles for search results that contain secrets. Ranges that contain secrets take longer to process due to expensive filtering operations that remove the sensitive identification numbers from the search results. The attacker can leverage this timing differential to perform a binary search over values returned by the range in order to recover the secret.

4.5.1.2 Software Design

The b-tree data structure used in this challenge stores all employment identification numbers, public and secret, in sequential order in a balanced tree. The b-tree algorithm itself does not process any actual data (like employee name, phone number, location, etc.…) or security information related to the employee's clandestine status, it simply stores in order the identification number for each employee and a pointer to an external structure that contains additional data. This results in a situation where searches for a set of employment ids that fall within a specified range will result in the b-tree returning all values that fall within that range – both public and secret. It is then the task of the application to filter, from those results, the identification numbers that represent employees operating in a clandestine manner. Because this filtering logic can require expensive accesses to additional employee information located outside of the b-tree, an application can leak timing information when the operations required to process the b-tree's results are noticeable to an external user. This can result in a situation where an attacker notices that certain range queries take longer to process than others, even when the size of the result list is the same.

A powerful side channel results from this situation when the attacker also has the ability to insert new public entries into the b-tree. By allowing this, attackers can now add new results into range queries that enable them to perform a binary search that determines, not only that a piece secret data exists within a range, but also its value. The attacker accomplishes this by placing new public data into the middle of a range that is taking suspiciously long to process. The attacker then searches the same range again but as two separate queries, with the new value being the upper bound of one query and the lower bound of the other. The secret lies in the result set of the query that takes longer to process. The attacker iteratively continues this search process until they narrow the search range to an individual value.

4.5.2 DotChallenge (Graph Analyzer)

4.5.2.1 Description

This challenge is **space utilization attack** on an application that parses and exports a modified version of the GraphViz DOT graph description language, graph specification. A DOT parser was

modified to create a keyword called “container”, which allows for a node in the graph to be represented by an arbitrary DOT graph. In our specification the container objects are allowed to have nested containers, creating an attack that is fundamentally equivalent to the billion laughs space utilization attack. When graphs are exported to a vector graphics format (PostScript), inputs utilizing the container keyword can have output PostScript files that are exponentially larger than their input DOT files.

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
<!ENTITY lol "lol">
<!ELEMENT lolz (#PCDATA)>
<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Space attacks for XML parsers rely on the fact that bounded nesting is allowable in the XML specification. This allows for the definition of a root element that leads to an exponential expansion to literal elements. A classic example of this technique is the, “billion laughs” attack.

In this attack, the document contains only one top level element of type “lol9”. As the document is parsed however, the parser must expand this element of “lol9” into ten elements of “lol8”, each of which expands to ten elements of “lol7”, etc. clearly leading to an exponential amount of memory consumption.

While originally demonstrated for XML, this space vulnerability can exist in any file format that allows for nesting. This challenge demonstrates this by modifying the DOT graph file format to enable nested graphs using a keyword called “container”. We notionally refer to this attack as the billon graphs attack.

4.5.2.2 Software Design

To develop this challenge, we first modified the Tigris Graph Editing Framework (GEF) library to change the specification for DOT graphs. We added the keyword “container” to allow for the hierarchical nesting described above. The below graph gives an example of a DOT file that would exhibit an exponential space explosion. The graph “main” would expand to have five nodes, each of which would be represented by a graph containing five nodes, etc.

```
graph "main" {
N1[type = "container:lol1"];
N2[type = "container:lol1"];
N3[type = "container:lol1"];
N4[type = "container:lol1"];
N5[type = "container:lol1"];
}
```

```
graph "lol1" {
k1[type = "container:lolx"];
k2[type = "container:lolx"];
k3[type = "container:lolx"];
k4[type = "container:lolx"];
k5[type = "container:lolx"];
}
```

```
graph "lolx" {
j1[type = "net"];
j2[type = "net"];
j3[type = "net"];
j4[type = "net"];
j5[type = "net"];
}
```

Figure 57. Billion Graphs Example Attack

To embed this vulnerability in a real world application we constructed a GUI based graph viewing tool. The tool is capable of visualizing arbitrary DOT graph loaded from disk, and exporting them to vector and raster based image formats. To use the tool, the user first selects an input a file with a graph described using our modified specification. They are then able to explore the graph, clicking through and seeing many sub-views of the graph. However, the graph is never fully expanded in memory during graph viewing.

When the user chooses to export the graph, the graph must be fully expanded and visually laid out. The graph layout can be done either using random layouts or force directed layouts. These laid out graphs can then be rendered to either PNG or PostScript format. Since PostScript is a vector format the expanded graph must have separate drawing instructions for each node in the graph, the output file can be extremely large relative to the input.

4.5.3 ip-challenge (Image Processor)

4.5.3.1 Description

This product is a basic mockup of some technologies going on trial to replace body scanners in them. A person would be able to walk through one of these new scanners, without removing clothing or unpacking suitcases, and image/video classification would take over the role of the human analyst. This challenge contains a **time complexity vulnerability** in the following way: if an attacker can inject a bad image into the training set, or wear something which could produce a timeout, this could cause the filtering system to fall over.

This is an image classifier that can be run in two modes, training and classify.

4.5.3.2 Software Design

The training stage performs no real processing, but the classification stage runs image algorithms on the images in the training set and the input image. The Intensify filter causes bad behavior for crafted images. An adversary would be able to send an image into the classification pipeline and an analyst would be stopped from classifying the image. A bad image is an image that allows a

lookup in the com.stac.Mathematics::intensify(IIII) function to find the largest value of the function:

$$accuracy[n] = \frac{1}{4+\tan(\frac{(30-n)}{255}+0.001)} \quad (3)$$

4.5.4 trie-challenge (SubSpace)

4.5.4.1 Description

Subspace is an anonymous, local messaging server. Registered users provide their locations ot the server. When a user wants to send a message, the message is anonymously delivered to the nearest other user. At this point, the two users can continue to communicate anonymously for as long as they wish, even if they are no longer nearby.

Subspace contains a **space side-channel exploit** that reveals a user's location. The attack is against a modified trie data structure.

The following data is stored in the servers database.

- usernames: The only person who should find out Alice's username is Alice.
- email addresses: (Same privacy concerns as usernames.)
- passwords: Only Alice should know her own password, and the password itself should never be stored unencrypted on disk.
- locations: A user's location is also private, with the exception that it can be used to determine where to route a new message. However, since the messages are anonymous, this routing does not disclose significant information about user locations.
- conversation aliases: Each conversation gets an alias. Anybody who knows a conversation's alias, knows one of the participant's email addresses, and is able to spoof that address, is able to join the conversation. Due to the hurdles, this is not a big secret, but it should not be disclosed unnecessarily either.
- conversation members: Each conversation alias has an associated set of email addresses that are party to the conversation. If Alice is one of the members of a conversation, she should not find out who else is a member of the conversation, since conversations are supposed to be anonymous. Additionally, a person who is not a member of a conversation should not be able to find out which users are talking with which other users.

There's also one type of secret which is not stored in the database:

- messages: The contents of a message should be accessible only to members of a conversation to which the message was sent. For the purposes of this app, we assume that sending an email to the configured Simple Mail Transfer Protocol (SMTP) relay server guarantees that only the specified recipients of that email will be able to read the message. This is not true in practice of course.

Subspace uses a modified Trie (https://en.wikipedia.org/wiki/Trie) data structure to find a user's nearest neighbor. In this trie, each node represents a "rectangle" in latitude-longitude space. An intermediate node has four children, one for each quadrant of the "rectangle". Thus latitudes-longitude coordinates are stored by their geographic prefixes, enabling quick searches for coordinates with a shared prefix (i.e., nearby coordinates).

This trie structure is vulnerable to a space side channel attack. Similar to words stored in a standard trie, some pairs of points have a longer "prefix" of sub quadrant divisions than others. Unlike a word trie, all "words" have equal length, because all leaf nodes have equal depth -- specifically, the depth necessary to get leaf quadrants no wider than a certain "precision", hard-coded to 1e-4 degrees. This precision works out to a depth of 22.

4.5.4.2 Software Design

The specific vulnerability arises because an attacker with HTTP access to the server and a user account on the same host -- but without read permission for the database file -- can add a point of their own to the trie and observe how much the database file grows. (The `stat()`, system call, which allows a user to see the size of a file, does not require read permission, only "execute" permission for the containing directory.) The amount by which the database grows indicates how many intermediate nodes had to be added to accommodate the point. If the quadrant to which the point was added is empty, the database file will grow by some maximum amount that indicates nodes had to be added "all the way down". If the quadrant is non-empty, the database will grow by less than this amount. This threshold amount decreases for each level of the trie, allowing an attacker to recursively check subquadrants until some user's location has been pinpointed down to the minimum quadrant size.

The trie has a precision of 1e-4 -- i.e., it stops splitting nodes by quadrant once a quadrant's width is smaller than this value. The exploit checks each node recursively, and each node has 4 children (its 4 quadrants), so each step down the trie cuts the maximum distance between corners of the node roughly in half. The number of such steps down the trie required to get from 360 (a very bad approximation of the maximum distance between two points on the whole Earth rectangle) to 1e-4 is roughly ceil(log_2(360/1e-4)), or 22.

At each of those 22 steps, the exploit checks all four sub-quadrants to see if it has any users, recursing into each sub-quadrant that does have a user(s). It takes exactly 1 HTTP GET and 1 stat() operation to determine whether or not a node contains any users.

The exploit checks quadrants in the order northeast, northwest, southwest, southeast. The worst case location for a single user is in the southeast quadrant of the southeast quadrant of the southeast quadrant, and so on. The most southeast quadrant is:

Rectangle(-90.0, -89.9999570847, 179.999914169, 180.0)

At each level, the exploit checks four quadrants, and the user is always in the last one it checks -- 4 steps per level. With 22 levels, it will take exactly 88 HTTP GETs and 88 stat()s to find the first user (not counting the initial unset and stat()).

4.5.5 stac-regex-challege (blogger)

4.5.5.1 Description

The application is a java web server which serves POJOs, and POJOs that collectively implement a simple public blogging application.

Users can browse to the homepage at http://localhost:8080. On the homepage users can sign in, create a new blog post and view posts by other users.

An example of a user interaction is provided using curl in the examples directory.

The start script (run.sh) for the server is located in the challenge_program directory. Once the server is started, the script will output "Server started, Hit Enter to stop." The web server listens on port 8080 and serves classes from the web app.

4.6 Engagement 1 Challenge Programs

4.6.1 CRIMEToy

4.6.1.1 Description

This challenge consists of a simple command-line executable that simulates compressing then encrypting an HTTP request that includes a session key intended to remain secret. It is vulnerable to the CRIME side channel attack **(space side-channel)**, described below.

4.6.1.2 Technical Details

CRIME is an attack which reveals secret data via a compressed-size side channel. The vulnerability arises when browsers compress HTTP requests before encrypting them.

CRIME is possible when the attacker can both control the request content and measure the size of the resulting compressed and encrypted data. The structure of a request document might be, e.g.:

```
POST (or GET) HTTP 1.1 /url
<More headers>
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
<Request Content>
```

If an attacker duplicates part of the request header in the request content -- e.g., the string `Cookie: phpsessionid=` -- then the request as a whole will compress to a smaller size than if the body contained unrelated data of the same length. Accordingly, if the attacker inserts a guess as to the `phpsessionid` secret. A correct guess will compress to a smaller size than an incorrect guess. This behavior allows an attacker to guess the secret one byte at a time.

This challenge uses the LZ77 (deflate sans huffman coding) compression algorithm to compress a document resembling an HTTP request, then encrypts it using AES in counter mode. The selection of LZ77 and AES-CTR ensures that the size difference between incorrect and correct guesses will always align with a byte boundary and will never be obscured by padding.

LZ77 functions by finding strings that are duplicates of each other and replacing the second occurrence with an (offset, length) pair referencing the first. This has an effect similar to these examples:

```
No Compression:
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX

Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
      Cookie:   phpsessionid=XXXX   SECRET   GOES   HERE   XXXXCookie:
      phpsessionid=XXXX SECRET GOES HERE XXXX
Length = 94
Duplicate:
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX(-47,47)
Length = 49
Random bytes:
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
      Cookie: phpsessionid=AAAAAAAAAAAAAAAAAAAAAAAAA
      Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX(-47, 21)AAA(-3,3)(-
      6,6)(- 12,12)(-15,2)
Length = 79
Bad Guess:
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
Cookie: phpsessionid=A
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX(-47, 21)A
Length = 50
Good Guess:
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX
Cookie: phpsessionid=X
Cookie: phpsessionid=XXXX SECRET GOES HERE XXXX(-47, 22)
Length = 49
```

4.6.2 Toy-challenge-hash-table

4.6.2.1 Description

This is a very simple application that provides a command line interface to a hash table, which is stored in memory. The application supports basic hash table operations such as insertion, retrieval, and deletion.

4.6.2.2 Technical Details

There is an **algorithmic complexity in time vulnerability** when many keys hash to the same bucket in the table. When this happens, the entries in the bucket form a linked list, which has performance O(n) over the number of entries, instead of a hash table's typical O(1) performance. For an input budget of 4,957,804 bytes, the resource usage limit is 30 seconds from program start to program termination.

An analysis of the vulnerability in the hash function itself is in doc/ModLinearHash-analysis.txt. A utility that exploits this vulnerability to create malicious input is in `HashTable-extra.jar`. Another utility that takes malicious input from `HashTable-extra.jar` and produces benign input of the exact same size (in bytes) is in the supplied python script: make-benign.py.

To generate malicious input, first pick the table size you want to target, which must be a power of two between 2^0 and 2^16. Then pick the number of colliding keys you want. E.g., to generate 1234 colliding keys for a table size of 2^10, and store the results in a file called `foo`, run:

```
$ java -jar HashTable-extra.jar 10 1234 > foo
Generated 1234 colliding inputs.
These work with a table size of 1024.
```

Then to create benign data of the same size in a file called `bar`, run:

```
$ ./src/extra/make-benign.py < foo > bar
```

Some sample inputs are available in [the data directory] (data/). data/16-100000-malicious contains malicious data from running `HashTable-extra.jar` with parameters of 16 and 100000, while data/16-100000-benign is a benign input of the same size. The times to process these inputs was the basis for the input budget and resource usage limit. Note that the use of a table size of 65536 here is important, because the malicious file was generated specifically for that size (2^16).

```
$ time java -jar HashTable.jar 65536 \
< data/16-100000-benign \
> /dev/null
Welcome to app.SimpleHashTableApp.
Type 'help' at any time to view a list of commands.
Goodbye!
Real 0m1.400s
user 0m2.344s
sys 0m0.218s
$ time java -jar HashTable.jar 65536 \
< data/16-100000-malicious \
> /dev/null
Welcome to app.SimpleHashTableApp.
Type 'help' at any time to view a list of commands.
Goodbye!
real 0m42.478s
user 0m43.338s
sys 0m0.264s
```

5.0 CONCLUSIONS

From the offensive perspective, side channel and algorithmic complexity attacks present some interesting advantages over more traditional classes of vulnerabilities while also posing key challenges. Whether developing an exploit for an existing vulnerability or introducing a vulnerability into some codebase, an adversary is primarily concerned with the severity of the vulnerability, the reliability of the exploit and how difficult it will be to detect the vulnerability. Memory-corruption vulnerabilities, by comparison, receive considerable attention because they may be easily and reliably exploited, often with high severity (e.g. remote code execution). However, significant research and resources went into developing an array of defenses against these vulnerabilities. Current software and hardware mitigations such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) degrade the reliability and severity of such vulnerabilities. Researchers have also developed extensive tooling ranging from static analysis to fuzzing techniques to detect memory-corruption bugs. Secure coding practices and memory-safe languages also make such vulnerabilities rarer in practice. All this has made memory-corruption vulnerabilities much rarer in practice and require significantly more skill to successfully exploit.

The landscape of cyber threats may therefore shift focus to a less traditional class of vulnerabilities. Side channels and algorithmic complexity vulnerabilities represent a real opportunity for both sophisticated and unskilled adversaries. Although there exists no simple method for measuring just how common side channel and algorithmic complexity vulnerabilities are, they appear rather prevalent. Numerous STAC performers discovered vulnerabilities in challenge apps' libraries. As a rule, programmers often optimize code for time and space which, in turn, introduces the possibility for side channel attacks. These vulnerabilities range from simple to sophisticated but, as we have seen, tend to be difficult to detect. Most enterprise security software, such as signature-based antivirus and Security Information and Event Management (SIEM) collection tools, would be woefully inadequate at detecting adversaries leveraging such vulnerabilities.

Similarly, developing an application based around this class of vulnerabilities proved challenging. Historical cases of side channel and timing complexity vulnerabilities offer some inspiration but, in general, no accepted methodology exists for constructing them. One can abuse well-known functions and data structures such as Java's HashMap implementation to incur poor performance. However, this tends to be more easily detected, much like using memory-unsafe functions in other languages. Of course, in practice, a HashMap itself may not be suspicious but, for the blue teams, it could be quite obvious. Understanding how code and data structures effect performance became key to constructing side channels and algorithmic complexity vulnerabilities.

Such interactions depend heavily on what the vulnerability might achieve from the adversarial perspective and, therefore, on the purpose of the application itself. We arrived at a methodology that might be equal parts art and science: first, create an application concept and, then, decide on a vulnerability. This coincidentally best models an adversary with source code access that introduces some "backdoor" into the application. Depending on the adversary's unique goals and motivations, much like our process of coming up with an application then vulnerability, the "backdoor" may leak information, deny service or otherwise undermine the security of some part

of the application. Such an adversary thus requires some familiarity with the source code but, in general, it should be quite easy to introduce this class of vulnerabilities into an existing codebase.

We found many generalizable ways to leak information or explode resource usage. For example, optimizing code to make common cases fast or abusing databases to slow runtime can leak potentially sensitive information. Algorithmic complexity attacks tended to be the easiest to introduce, often using research from some well-known algorithm, but also proved the easiest to detect. Multi-threaded applications proved to be another viable source of side-channels where racing conditions can be observed forming a beat frequency with two racing threads. This frequency was often correlated to the underlying secret. But multi-threading in particular is always hard to reason about, which presents a big black-box target for a potential adversary. Of course, many side channels may be already present in an application without an adversary needing to introduce them.

A recurring theme on the STAC program was the presence of unintended vulnerabilities. In general, we attempted to guard against these but only by informally reasoning about code behavior. This perhaps best illustrates how challenging it may be for traditional software developers to mitigate this class of vulnerability since even security-aware developers could introduce them unintentionally. Developers may know to use certain memory-safe functions but have little motivation to understand the security implications of time and space. In fact, they may even be incentivized to optimize for time and space which may introduce side channel vulnerabilities.

As a result of these vulnerabilities, the intentionality of them was frequently called into question by the blue teams. Since they may be so easily introduced by accident, this makes this class of vulnerability very attractive as a form of insider attack. A programmer might easily implement a side channel to leak information for their own purposes while maintaining plausible deniability that the vulnerability was simply unintentional. Real-world programs also differ greatly from the challenge applications we produced. Most have multiple authors with significantly larger and more complex codebases plus many external libraries. This means a potentially large attack surface for an adversary looking for an existing vulnerability in an application to exploit.

6.0 RECOMMENDATIONS

While TA1 teams developed many techniques for finding STAC-related vulnerabilities, recent findings in hardware side-channels proved the usefulness of STAC-related efforts and highlighted a need for even more efforts to detect, find, analyze, and potentially mitigate Space- and Time-complexity vulnerabilities and side-channels. Platforms other than Java are of a particular interest, which present deeper complexity and additional analysis problems and the DALEC's Java challenge problems are available on github for public research and DALEC team encourages further research into side-channel vulnerabilities – attacks we will likely see more of in the future.

7.0 REFERENCES

[1] IEEE Task P754, ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic, IEEE, New York, 1985.

[2] I. Dooley and L. Kale, “Quantifying the interference caused by subnormal floating-point values,” https://charm.cs.illinois.edu/newPapers/06-13/paper.pdf

[3] Wikipedia entry for HMAC. Retrieved from https://en.wikipedia.org/wiki/HMAC

[4] Wikipedia entry for Initialization Vector. Retrieved from https://en.wikipedia.org/wiki/Initialization_vector

[5] Wikipedia entry for Levenshtein distance. Retrieved from https://en.wikipedia.org/wiki/Levenshtein_distance

[6] Wikipedia entry for Bitcoin. Retrieved from https://en.wikipedia.org/wiki/Bitcoin

[7] Wikipedia entry for Subset sum problem. Retrieved from https://en.wikipedia.org/wiki/Subset_sum_problem

[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. "35: Approximation Algorithms". Introduction to Algorithms (2nd ed.). McGraw-Hill Higher Education.

[9] Wikipedia entry for PageRank. Retrieved from https://en.wikipedia.org/wiki/PageRank#Algorithm

[10] Wikipedia entry for GEDCOM. Retrieved from https://en.wikipedia.org/wiki/GEDCOM

[11] Wikipedia entry for Algebraic notation (chess). Retrieved from https://en.wikipedia.org/wiki/Algebraic_notation_(chess)

[12] Dickins, Anthony Stewart Mackay. A Guide to Fairy Chess. Dover Publications, 1971.

[13] Wikipedia entry for Power Iteration. Retrieved from https://en.wikipedia.org/wiki/Power_iteration

[14] Wikipedia entry for Arnoldi Iteration. Retrieved from https://en.wikipedia.org/wiki/Arnoldi_iteration

[15] Wikipedia entry for Representational state transfer. Retrieved from https://en.wikipedia.org/wiki/Representational_state_transfer

[16] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. Entry A000522. Retrieved from https://oeis.org/A000522

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

Ack	Acknowledgement
AEI	adjusted expenditure limit
AES	Advanced Encryption Standard
AFRL	Air Force Research Laboratory
AI	Artificial Intelligence
AIS	Assured Information Security
AJAX	Asynchronous JavaScript and XML
APAC	Automated Program Analysis for Cybersecurity
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
BBN	Raytheon BBN Technologies Corp.
CDRL	Contract Data Requirements List
CFG	control flow graph
CGA	Callgraph analysis
CSS	cascaded style sheet
CSV	comma separated values
CTR	CounTeR mode
DALEC	Developing Algorithms that Leak or Explode in Complexity
DARPA	Space/Time Analysis for Cybersecurity
DEP	Data Execution Prevention
DER	Distinguished Encoding Rules x509
DES	Data Encryption Standard
DH	Diffie-Hellman
DOT	Graph description language
EAR	Export Administration Regulations
FAT-32	File Access Table 32 bit
FIFO	first in-first out
GEDCOM	Genealogical Data Communication
GSON	Google’s open-source Java library to serialize and deserialize Java objects to JSON
GUI	graphical user interface
H2	free open source relational database management system written in Java
HMAC	Hashed Message Authentication Code
HTML	hypertext markup language
HTTP	hypertext transfer protocol
HTTPS	HTTP Secure
I/O	input/output
IBASys	Image Based Authentication System
ID	identifier
IEEE	Institute of Electrical and Electronics Engineers

IoC	inversion of control
IP	internet protocol
IR	intermediate representation
ITAR	International Traffic in Arms Regulations
IV	initialization vector
JAR, jar	java archive
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JVM	java virtual machine
KM	KiloBytes
LSLAP	Large Scale Linear Algebra Platform
MB	MegaBytes
MD5	Message-Digest 5
MST	minimum spanning tree
MVC	Model-View-Controller
NP	nondeterministic polynomial time
NUC	Intel's Next Unit of Computing
OPF	Optimized Power Flow
OpenJDK	Open Java Development Kit
PC	Program Counter
PE32	Portable Executable 32-bit
PEM	Privacy Enhanced Mail
PID	proportional–integral–derivative
PNG	Portable Network Graphics
POJO	Plain Old Java Object
PRNG	pseudo-random number generator
PSA	Python Static Analyzer
PSSE	power system state estimator
PTA	Points-to Analysis
R&D	Research & Development
RC4	Ron's Code 4 (RSA Variable-Key-Size Encryption Algorithm by Ron Rivest)
RDA	reaching-definition-based analysis
REST	Representational State Transfer
RPC	Remote Procedure Call
RSA	Rivest–Shamir–Adleman
SAC	Scientific Adversarial Challenge
SC	Space Complexity
SCM	Service Control Module
SDL	Stuff Description Language
SHA-1	Secure Hash Algorithm 1
SHA-256	Secure Hashing Algorithm, 256-Bits
SOUR	source (genealogical term)

SQL	Structured Query Language
SSA	Static Single Assignment
SSC	Space Side-Channel
SSL	Secure Socket Layer
STAC	Space/Time Analysis for Cybersecurity
TA	Technical Area
TC	Time Complexity
TCP	Transmission control protocol
TSC	Time Side-Channel
UDP	User Datagram Protocol
UI	user interface
URL	Universal Resource Locator
X86	a CPU instruction set compatible with the Intel 8086 and its successors
XML	extensible markup language
ZIM	Zeno IMproved file format