AFRL-RI-RS-TR-2019-167

COMPLEXITY AND SIDE-CHANNEL ADVERSARIAL INTEGRATED DEFECTS (CASCAID)

CYBERPOINT INTERNATIONAL, LLC.

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-2019-167 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WALTER S. KARAS
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 2019

4. TITLE AND SUBTITLE

COMPLEXITY AND SIDE-CHANNEL ADVERSARIAL INTEGRATED DEFECTS (CASCAID)

5a. CONTRACT NUMBER
FA8750-15-C-0090

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Riva Borbely
Mayur Darji
Colin Mason
P.N. Safier

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
CY

5f. WORK UNIT NUMBER
PT

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

CyberPoint International, LLC.
621 East Pratt Street, Suite 610
Baltimore, MD 21202

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/I20
675 N Randolph Street
Arlington, VA 22203

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

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-167

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

In the role of adversarial challenger on the DARPA STAC program, CyberPoint developed many challenge programs designed to test the research tools being developed under the program for their ability to help analysts detect space-and time-related vulnerabilities in Java programs. In this document, we describe our approach to our Complexity and Side-Channel Adversarial Integrated Defects (CASCAID) effort, the vulnerabilities we developed, their efficacy, and what we learned in the process.

15. SUBJECT TERMS

Cybersecurity; vulnerabilities; algorithmic complexity; side channels; polymorphic code.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WALTER S. KARAS**
U	U	U	UU	97	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Contents

Figures

Tables

1 Summary

In the role of adversarial challenger on the DARPA STAC program, CyberPoint developed many challenge programs designed to test the research tools being developed under the program for their ability to help analysts detect space- and time-related vulnerabilities in Java programs. In this document, we describe our approach to our Complexity and Side-Channel Adversarial Integrated Defects (CASCAID) effort, the vulnerabilities we developed, their efficacy, and what we learned in the process.

2 Introduction

The goal of the DARPA Space/Time Analysis for Cybersecurity (STAC) program is to develop program analysis techniques and tools for detecting vulnerabilities related to the space and time resource usage of Java programs, specifically side channel vulnerabilities in the space and time usage of the program, and vulnerabilities that could allow an attacker to cause excessive time or space (i.e., memory or disk space) resource usage by the program.

Many research teams endeavored to develop such tools, competing against a "control" team using off-the-shelf tools. These teams were collectively referred to as blue teams, and included groups primed by:

- Draper Laboratory ("Draper")
- GrammaTech
- Iowa State University ("Iowa")
- Northeastern University ("Northeastern")
- University of Colorado ("Colorado")
- University of Maryland ("Maryland")
- University of Utah ("Utah")
- Vanderbilt ("Vanderbilt")
- Invincea/Two Six Labs---the control team

Meanwhile, the efforts of the adversarial challengers (red teams, comprising CyberPoint and BBN) on the program aimed to provide an extensive array of tests for the techniques and tools developed in the program, in order to 1) determine the extent to which the techniques and tools achieve their goal, and 2) identify any weaknesses in said techniques and tools.

To that end, under the Complexity and Side-Channel Adversarial Integrated Defects (CASCAID) effort, CyberPoint has developed 16 Java applications and more than 50 distinct vulnerabilities. These can be downloaded from the DARPA STAC public release GitHub repository [1].

We have also developed a complex build system that supports rapid mixing and matching of applications with different combinations of vulnerabilities and vulnerability mitigations. This build system integrates a suite of code transformations we developed that easily provide variety between variants of an individual application, to prevent the vulnerabilities from becoming apparent simply by `diff`ing two different versions of the application.

We are pleased to provide an extensive suite of samples for testing STAC-related vulnerability detection tools, as well as testing the ability of human analysts to detect these vulnerabilities, as well as a system to enable expansion of this suite in the future.

3 Methods, Assumptions, and Procedures

3.1 Operational Definitions

The focus of the STAC program was on two categories of vulnerabilities in space and time: side channel (SC) and algorithmic complexity (AC) vulnerabilities.

Side channel vulnerabilities were defined as follows: “A challenge program contains a side channel vulnerability if and only if an adversary can extract a worst-case complete secret value from the challenge program with a defined probability of success by executing a specifically bounded number of operations,” where operations include:

- Active operations - provide an input to the program and observe its response.
- Passive operations - observe the inputs and outputs of the program.
- Oracle queries - use a notional oracle to determine whether a guess is correct.

There were three types of side channel vulnerabilities:

- side channel in space (SC-S).
- side channel in time (SC-T).
- side channel in space and time (SC-ST).

While other observables were discussed, in practice the only measure of space for side channels was the size of network packets (and occasionally sizes of their contents, in the case of an attacker who could view them). Other space observables, such as memory usage, were deemed impractical, because an attacker has no reasonable way of observing them on the victim’s machine. (The latest operational definition still includes logical file size, which we believe suffers from the same issue. We did initially propose some side channels using this observable, however none were ever approved because there was no sufficiently reasonable scenario in which an attacker would be able to observe it.)

There were also three types of algorithmic complexity vulnerabilities:

- algorithmic complexity vulnerability in time (AC-T).
- algorithmic complexity vulnerability in disk usage (AC-Disk).
- algorithmic complexity vulnerability in memory usage (AC-Memory).

These were defined as follows: “A challenge program contains a vulnerability to algorithmic complexity attack if and only if it is possible for an adversary to cause that challenge program to exceed a specific resource usage limit, on the reference platform (section 3.1), with a defined probability of success, after feeding it some number of bytes of input, where that number of bytes is less than a specific input budget.”

Challenge programs were presented to blue teams in a sequence of "engagements", during which they had a limited time to identify vulnerabilities within them. For each challenge program, the blue teams were given a set of specific questions to answer. AC questions include the resource the attacker is to use (disk space, memory, or time), the resource limit to exceed, and an input budget of the number of bytes the attacker is allowed to send. SC questions include information on the secret the attacker is trying to obtain (often a task they are to achieve with the secret), an operational budget, and the allowed scope of oracle queries. Oracle queries are specific questions that an attacker can (at least in theory) obtain answers to, to check the correctness of a guess, giving the attacker an opportunity to eliminate some uncertainty that might exist in the side channel.

Finally, both types of questions specify a probability of success. Questions that ask about a vulnerability that is not present are referred to as null questions.

3.2 The Reference Platform

In order to allow algorithmic complexity vulnerabilities to be defined in terms of specific resource usage, as well as enabling consistency in determining whether a timing side channel was strong enough to exploit, it was decided that all STAC performers would run the challenge programs on a common reference platform. The hardware selected was the Intel Next Unit of Computing (NUC), NUC5i5RYH, with SSD Samsung 850 EVO M.2 SATA 6Gb/s [1] and Memory Crucial 16GB Kit 2-8GB PC3-12800 DDR3 KIT,[2] with operating system CentOS 7. The first four engagements used Java 7 [3], and the remainder used Java 8 [4].

The NUCs are affordable, portable, and space-efficient.

In Engagements 1 through 4, the NUCs were used individually, with no networking, while in Engagement 5 and beyond, they were set up in networks of three nodes, one for the server/victim, one for the client, and a third one where network traffic to and from the victim can be observed (without impacting the timing on the server.) Figure 3-1a shows the basic setup, while fig. 3-1b shows more detail in the case that there are additional peers beyond a single victim and attacker.

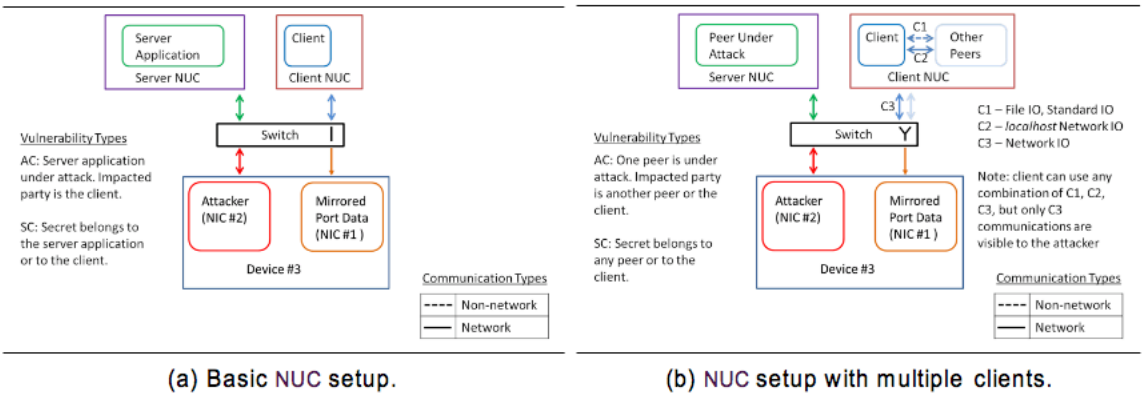


Figure 1: The NUC reference environment.

4 Vulnerability Aims

Our ultimate goal in developing vulnerabilities is to thoroughly test the vulnerability detection tools being developed in the STAC program. As such, it is important that the vulnerabilities require advanced tools to detect them. We list below the approaches that we used to make vulnerabilities hard to detect.

4.1 Rare Malicious Input

For AC vulnerabilities, fuzzing is a standard approach. (DARPA did not find any evidence to support fuzzing as a standard approach for AC vulnerabilities.) The most basic fuzzer simply runs the program with random inputs. To avoid detection by primitive fuzzers, it is imperative that input that exercises the

[1] Model mz-n5e250bw
[2] Model ct2kit102464BF160B
[3] java-1.7.0-openjdk-1.7.0.85-2.6.1.2.el7_1.x86_64.rpm
[4] java-1.8.0-openjdk-1:1.8.0.65-2.b17.el7_1.x86_64.rpm

vulnerability be rare. As such, this was one of our fundamental principles in designing AC vulnerabilities. More advanced fuzzers apply a sampling policy to cover the input space. Vulnerabilities that require input with a very small fractional part, such as slow convergence of Newton's method can evade such fuzzers. The most advanced fuzzers apply an intelligent strategy to hone in on inputs that cause increasing use of time, memory, or disk space. Both Northeastern and Colorado had such fuzzers that clearly demonstrated this ability, but only Colorado's was effective on the most hard-to-find malicious inputs in continuous input spaces.

Another approach to making vulnerable input rare is to require input that is malformed in a specific way. One vulnerability that none of the teams found, the recursive answer vulnerability in SimpleVote, revolved around an input that represented a self-referencing data structure. Zip bombs and XML bombs are similar, but standard and well-studied. Northeastern and Vanderbilt found our zip bomb vulnerability in the second engagement, but only Two Six Labs found our XML bomb in the fourth. We also identified a vulnerability in the FileUpload package from Apache Commons, where processing an HTTP request with an incorrect content-length header that is larger than the actual content seems to cause it to hang indefinitely. (CyberPoint did not provide additional clarification why this occurred.) This was exploitable through our web server code in various challenges in engagements 3 through 7, but none of the blue teams reported it.

4.2 Non-Deterministic Behavior

Another possible way to make algorithmic complexity vulnerabilities less apparent through dynamic analysis is to make their resource usage variable, so that a single test may indicate that a particular input does not trigger excessive resource usage. This is not a technique that we used heavily in earlier engagements.

In Engagement 6, there was a memory vulnerability in BattleBoats (section 6.9) which only manifested itself 80% of the time, due to the non-deterministic behavior of the garbage collector. It is unclear whether this thwarted any blue team tools; however, only one of the research teams detected the vulnerability. They did not seem to notice the inherent non-determinism, and, interestingly, in the collaborative portion of the engagement, although the teams heavily questioned what the cause of the excessive memory use was, the role of garbage collection did not come up at all in their discussion. (CyberPoint did not provide additional clarification why the teams should have identified this in their discussions.)

In Engagement 7, we employed non-determinism in several side channels, as discussed below.

4.3 Subtle Side Channels

In a search for side channel vulnerabilities, while automation is helpful, it is fairly straightforward to observe the packet sizes and timings corresponding to different secret values and look for correlations.

One way to make side channels more subtle is to make the correlation dependent on multiple timings/sizes. For example, the InAndOut confirmation number side channel in space and time (see section 6.14.1) relies on the size of packet to eliminate random noise in a timing, and the Suspicion location side channel (section 6.18.1) requires the sizes of three different messages to obtain the secret. The latter was observed by all blue teams in Engagement 7 (but they believed that packet padding rendered the side channel unexploitable). However, the side channel combining space and time simultaneously was not detected by any blue team.

Also, adding noise can make it difficult to determine whether the side channel is exploitable. In Engagement 1, one team thought that the RSA side channel [2] was mitigated by a miniscule amount of random noise introduced by one of our transforms! In Roulette, the spin time (which directly correlates with the winning slot) differs from the reminder time by a random interval which is not large enough to eliminate the side channel. While at least one blue team was initially thrown off by this, by the end of collaboration, they were convinced that the side channel was sill viable. Also, in Roulette, we used non-determinism to mitigate the CBC (cipher block chaining) side channel, by randomly (but rarely) responding with a message of a conflicting size to prevent 100% success. Since the blue teams did not detect this side channel at all, they didn't notice this mitigation, either. In the CyberWallet biased ads side channel, the advertising shown is selected based on a random number multiplied by the user's account balance. The blue teams were able to determine that the randomness here was surmountable with an oracle query in the worst case.

Other side channel vulnerabilities correlate with more subtle secret information.

For example, in our RSA vulnerability, the timing merely reveals the proximity of a multiple of one of the primes used in generating the secret key. Only two teams detected this vulnerability in Engagement 4, and one of them was the control team. Similarly, some non-obvious math is required to translate between the bytes obtained via timing/space tests and the actual bytes of the message in the CBC vulnerability. The blue teams did not detect the CBC vulnerability.

4.4 Side Channels without Conditionals

The natural way to look for side channels is to look for branches in code with dataflow from a secret, which are most often if conditionals. The expectation was that the blue teams would struggle with side channels without conditionals.

Many of our side channels fit this description.

The CBC side channel in Suspicion (section 6.18.1) and Roulette (section 6.17.1) relied on exceptions, rather than conditionals. The side channels in BraidIt were achieved by formatting a string with padding of length equaling the braid index times a length. The CyberWallet advertising side channel (section 6.13.1) and the WithMi file transfer side channel (section 6.8.1) lay entirely in the data. The roulette winning slot side channel (section 6.17.1) resided in a sleep and some modular arithmetic. The suspicion location side channel (section 6.18.1) merely allocated an array with size a function of the portion of the secret. The InAndOut side channels (section 6.14.1) relied on linear feedback shift registers (LFSRs) of different register lengths—that resulted in different computation times—to compute the secret confirmation number.

However, ultimately, we believe the blue teams primarily relied on dynamic analysis in their search for side channels, so we don't believe it mattered much what code structures they were implemented with. For example, GrammaTech primarily used their tools to identify methods that create network traffic.

4.5 Non-Locality

Throughout the STAC program, it has been an ongoing theme that blue team tools struggle with non-locality. We have generally attempted to spread our vulnerabilities over different methods and classes.

For example, the vote-percentage side channel in SimpleVote (see section 6.11.1) required setting the candidate name in a class entirely unrelated to the classes that dealt with vote tallies. Only two blue teams found this vulnerability in Engagement 5. Likewise, the trigger in one of the BraidIt side channel

vulnerabilities (Braid Selected MAX (SC-S) section 6.10.2) was far removed from the actual vulnerability. While three blue teams found the side channel vulnerability, only one of them realized that it had to be triggered.

Another example was the shuffling mitigation of the braid selection side channel (see section 6.10.2) in BraidIt, where the shuffling was well-separated from the leak, and two blue teams completely missed it. In another manifestation of non-locality, several vulnerabilities were reliant on multiple distinct "bugs" to be exploitable. For example, the Huffman decompression AC-T vulnerability in WithMi (section 6.8.1) relied on an input reader that continued to return values after reaching the end of the file, as well as a counting error that renders useless a guard that defends against excessive disk writes. No blue teams detected this vulnerability. Similarly, in Calculator, in order to exceed the disk space usage bound, it is necessary to take advantage of a bug in determining the size of a POST, as well as having just the right input to cause the message to be expanded. While the blue teams all failed to detect this vulnerability that was a result of their not even noticing that there was any potential for a disk write outside of the log file, and probably not due to non-locality.

On the other hand, we used localization to our advantage as well. In Calculator, there are several different calculators (basic, Roman numeral, etc.). Instead of having a separate handler for each of these calculators, we joined them all within a single handler. This was done to make it harder to isolate code related to a particular functionality.

4.6 Decoys

As was suggested in the canonical examples (Category 2) [3], we expected that some blue teams might have trouble discovering vulnerabilities if there were other areas of the code with higher computational complexity. This was most often tested by our nested loop transformation, which randomly selected loops in our code to add nesting to, where the added nested loops terminate non-deterministically (e.g., `if random.nextFloat() < 0.4`). There were many times that the blue teams noticed these loops, and there was at least one occasion where they believed their randomness mitigated a side channel. As far as serving as a decoy, there was at least one occasion where a blue team saw nested loops of depth d and believed that the code had $O(n^d)$ time complexity, missing the fact that the random part of the loop would prevent them from iterating *n* times. So, not only did these serve as a successful decoy from the actual vulnerability, but they demonstrated the ineffectiveness of some tools in gauging the complexity of such code constructs (which was our primary intent in including them.)

4.7 Requiring Static/Dynamic Analysis

We also used some techniques that we expected to favor a static or dynamic analysis approach.

For example, to complicate the analysis of our code, we implemented the visitor pattern, a popular design pattern in object-oriented design [4]. The visitor pattern allows for an object to be delocalized from its functionality. A layout of the visitor pattern structure is shown in fig. 4-1.

The main advantage to this design pattern is that new functionality can be added to objects without having to modify the implementation of the object or the program that the object is a part of. The visitor pattern achieves this by implementing *double dispatch*. In Java, when deciding which function to call when there are overriding functions, single dispatch is used, i.e., the JVM looks at the concrete type of a single object to decide. When using the visitor pattern, we force Java to do double dispatch—deciding the overriding function by looking at the concrete type of two objects (`ConcreteObj` and input parameter of the visit method in `ConcreteVisitor` in fig. 4-1). This dispatching is done during runtime.

We did not use the visitor pattern for its ability to allow us to delocalize functionality, however. After Engagement 4, we learned that the blue team tools were using more static analysis than dynamic analysis. The visitor design pattern required the blue teams to use dynamic analysis in order to determine the behavior of the double dispatch. We incorporated this design pattern in the object serialization functionality of SimpleVote section 6.11. Each serializable object in the application had a delocalized serialization and a deserialization method. In Engagement 5, none of the blue teams found our recursive answer vulnerability in the SimpleVote serialization (section 6.11.1), which suggests that the visitor pattern may have been an effective obstacle, even with dynamic analysis approaches.

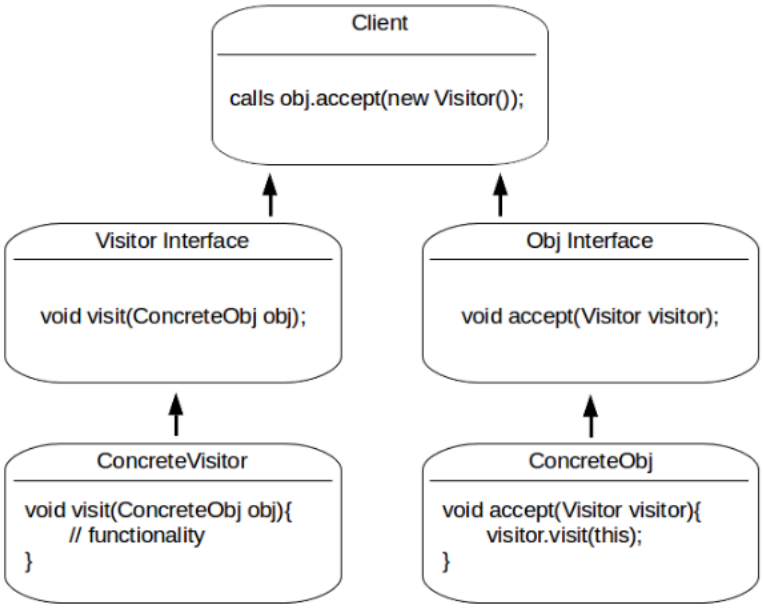


Figure 2: Diagram of the Visitor Pattern in Java

We also implemented features that we expected to favor static analysis approaches, such as requiring a trigger to enable the side channel. We expected static analysis tools to be helpful in identifying triggers, and in some cases they were, but there were a number of cases where blue teams seemed to have accidentally triggered a vulnerability dynamically without even realizing that that was necessary.

5 Build System and Transformations

One of the fundamental principles underlying our build system for CASCAID is the concept of host programs and kernels. For each application, the host program contains the main functionality of the application, while different kernels contain small portions of code containing vulnerabilities and/or mitigations thereof. As such, the code for a given challenge program may be spread across several disparate directories, with the majority of the code residing under `host_programs/program_name` (prior to Engagement 5, much of this benign code could be found instead in `kernels/common/program_name`), and the reminder residing among several different `kernels/kernel_name` directories. Sometimes we would have multiple distinct kernels implementing the same benign functionality with different vulnerabilities and mitigations, while often we would minimize code duplication with a single kernel from which multiple variants are generated via code transformations, as described in section 5.1.

The job of our build system, which we implemented with Gradle [5] and Groovy [6], is to gather up all the source code for a given challenge program and apply the relevant transformations before compiling (and testing) the code. The final piece of the puzzle is the article configuration file, which corresponds to a particular variant for a challenge program, specifying a host program, a set of kernels and their respective configuration parameters, and the specification of the transformations to apply.

From these pieces, our build system generates separate deliverables for the Engagement Lead (containing the generated/transformed source code, vulnerability descriptions and proofs) and for the blue teams (containing only bytecode and demonstrations of basic benign program functionality). These deliverables included Docker containers for each challenge program, which were built with all required dependencies.

Part of the build pipeline includes running tests, which range from tests of benign application functionality to proofs of the presence or absence of our intended vulnerabilities. Tools for running these tests include Python scripts for running tests and monitoring the results in Docker. There are three main test types: benign, malicious, and internal. Benign tests verify the basic benign functionality of the program, malicious tests verify the presence or absence of intended vulnerabilities, and internal tests are tests for our own development use that we did not deliver during the engagements, often testing edge cases, or attempting to measure normal program resource usage. A fourth kind of test, fuzz, was implemented towards the end of the effort in hopes of discovering any unintended vulnerabilities. (CyberPoint did not provide additional clarification why this was not added earlier in the effort and whether it was successful or not.)

In order to construct the deliverables, as is typical in the Gradle build lifecycle, all tests must pass before the deliverable is created.

Our build pipeline alters the typical Java build cycle with the addition of custom and basic tasks. The custom tasks include source code transformations (`TransformSourcesTask`), program and vulnerability descriptions (`ReadMeTask` and `BenignReadMeTask`), questions for the blue teams (`QuestionsTask`), counting the lines of source code (`SlocTask`), and API method listings (`MethodFinderTask`). Additional tasks are added that are not hooked into the standard build cycle but are provided to enable general targeted testing of the articles and testing by specific vulnerabilities (e.g., executing all side channel tests, all algorithmic complexity tests, or all benign tests). These additional tasks are performed after all of the source generation tasks are finished.

Note that, while the code for our build system was used to create and test the challenge programs we provided for the blue teams, it was not actually delivered at any time. At this time, this code can only be found in our final software delivery. For more information on how to use our build system, see our CASCAID User's Guide [7].

5.1 Jinja Transformations

Jinja [8] is a templating language that allows for the creation of polymorphic code. To make use of Jinja, we included Jinja directives within our Java files, changed the file extension on these files from `.java` to `.java.jptl`, and configured the Jinja parameters in an article configuration file.

Most often, this allowed us to elegantly pick and choose different versions of vulnerabilities and their mitigations, without having to duplicate code. Within the `.java` file, this was as simple as what's shown in snippet 1.

Additionally, we could configure the values of constants in the source code, e.g., the size of the largest POST to allow in the vulnerable web server, as shown in snippet 2.

Finally, there were cases where we took greater advantage of Jinja's capabilities to create truly polymorphic algorithms, using Jinja to actually generate code. For the ChangingSort and Fast Fourier Transform (FFT) algorithms, we wanted to create a large variety of recursive algorithms with different divide-and-conquer strategies. For FFT, for example, we took advantage of the ability to split an array into 32 parts, as shown in snippet 3.

As an example, with k=3, the Jinja code in snippet 3 would be transformed to that shown in snippet 4.

Aside from putting these Jinja directives in the desired Java files, we configure the article configuration file for each challenge program, in our articles directory, e.g., `articles/calculator_1.config`, as shown in snippet 5. The Jinja configuration is done under the `srcTransform:pystac.trans.jinjatransformer` directive.

```
// common code
// ...

{% if VERSION == "VULNERABLE_1" -%}
// vulnerable code
// ...
{% if VERSION == "VULNERABLE_2" -%}
// alternate vulnerable code
// ...
{% elif VERSION == "MITIGATION_1" %}
// benign code
// ...
{% elif VERSION == "MITIGATION_2" -%}
// alternate benign code
// ...
{% endif -%}

// more common code
// ...
```

Snippet 1: Using Jinja to create discrete code options. Jinja directives are delineated with "{%" and "%}".

```
private static final int MAX_POST_LENGTH = {{maxPostLength}};
```

Snippet 2: Using Jinja to define variable values. A value configured with Jinja is delineated with "{{" and "}}".

```
int q1 = (int) Math.floor((initStart + initEnd)/2);

{% for number in range(2, k + 1) -%}
   int q{{number}} = (int) Math.floor((q{{number - 1}} + 1 + initEnd)/2);
{% endfor -%}

changingSort(list, initStart, q1);

{% for number in range(1, k) -%}
   changingSort(list, q{{number}} + 1, q{{number + 1}});
{% endfor -%}

changingSort(list, q{{k}} + 1, initEnd);
{% for number in range(k , 1, -1) -%}

   if ((q{{number - 1}} + 1) <= q{{number}} && (q{{number - 1}} + 1) != initEnd) {
      merge(list, q{{number - 1}} + 1, q{{number}}, initEnd);
   }
{% endfor -%}
```

Snippet 3: Code generation with Jinja

```
int q1 = (int) Math.floor((initStart + initEnd)/2);

int q2 = (int) Math.floor((q1 + 1 + initEnd)/2);
int q3 = (int) Math.floor((q2 + 1 + initEnd)/2);

changingSort(list, initStart, q1);

changingSort(list, q1 + 1, q2);
changingSort(list, q2 + 1, q3);
changingSort(list, q3 + 1, initEnd);

if ((q2 + 1) <= q3 && (q3 + 1) != initEnd) {
   merge(list, q2 + 1, q3, initEnd);
}
if ((q1 + 1) <= q2 && (q2 + 1) != initEnd) {
   merge(list, q1 + 1, q2, initEnd);
}
```

Snippet 4: Transformation of snippet 3 with k=3

Note that the Jinja configuration was applied within a particular kernel, while the `allSrcTransform` directive was applied to all of the source code, in both the host portion, as well as all included kernels.

```
"host": "calculator",
"kernels": [
   {
      "path": "calculator",
      "srcTransform": [
         {
            "name": "pystac.trans.jinjatransformer",
            "args": [
               "numOfDivides=32", "VARIANT=VARIANT_1"
            ]
         }
      ],
   }
],
"allSrcTransform": [
   {
      "path": "buildSrc/transform/build/install/transform/bin/transform",
      "args": [
      "--libdir",
      "${libdir}",
      "--seed",
      "${articlename}",
      "--prob",
      "0.5"
   ],
   "optional": true
   }
]
```

Snippet 5: Article file to configure Jinja

5.2 JavaParser and IntelliJ Transformations

We also used another type of code transformation that was not directly related to vulnerabilities. With the approach of providing different versions of the same application with different vulnerabilities, an obvious concern is that blue teams will hone in on the differences between variants to find the vulnerabilities. Our primary weapon against this is a suite of code transformations that we developed that are randomly applied to the code automatically. (We also occasionally place extra differences, unrelated to the vulnerabilities, directly in the code with Jinja.) Initially, we implemented the transformations using JavaParser [9].

The probability of applying a given transform, along with a random seed that determines exactly when the transform will be applied, is specified in the article configuration file (see example in snippet 5).

JavaParser is a Java package for parsing, analyzing, transforming, and generating Java code. With JavaParser we implemented the following code transformations.

- Extract a suitable code block into a separate method
- Add a method call to the beginning of a method
- Extract a method into its own class
- Modify a primitive loop variable to an Object (see snippet 6)
- Randomly rename local variables
- Intelligently rename packages, classes, methods, and variables, using a developer-supplied list of synonyms for commonly used names and name fragments

- Nested non-deterministic looping (see snippet 7)

In addition to providing code diversity between variants in order to hide the differences related to vulnerabilities, some transforms also create code that tests key features of blue team approaches. Sophisticated multi-counter analysis is necessary to differentiate between nested non-deterministic looping and an O(n^k) loop [10]. This code threw several blue teams for a loop over the course of the program. In one case, they thought the randomness was sufficient to blind a side channel!

Similarly, loops with a termination condition controlled by an object also require more powerful analysis than a simple counter increment. Finally, the additional method calls may be sufficient to render the call graph too large for the blue teams to analyze.

Later, in place of JavaParser we began using IntelliJ's PSI (Program Structure Interface) [11], which supports their IDE's automatic code refactoring, thus giving us a lot of transformations that are essentially built-in. We added the following transformations:

- Replace a constructor with a builder.
- Move parts of a class into a new class.
- Move a block of code into an inner class.
- Transform for-each loops into regular indexed loops.
- Convert static methods to instance methods.

```
// original code

for (int i=0; i<n; i++) {
   System.out.println(i);
}

// gets transformed into the following

for (Obj i_orig = new Obj(0), i = i_orig; i_orig.getValue() < n; i.increment()) {
   System.out.println((int) i.getValue());
   i.foo();
}

class Obj {
   private float val;
   public Obj(float init_val) {
      val = init_val;
   }
   public void increment() {
      val++;
   }
   public void decrement() {
      val--;
   }
   public float getValue() {
      return val;
   }
   public void setValue(float value) {
      val = value;
   }
   public void setValue(int value) {
      val = value;
   }
   public void foo() {
      val /= 1.0;
   }
}
```

Snippet 6: Transforming a primitive loop variable to an Object

```
// original code
for (int i=0; i<n; i++){
   System.out.println(i);
}

// gets transformed into
for (int i = 0; i < n; ) {
   Random rand = new Random();
   for (; i < n && rand.nextDouble() < 0.9; i++) {
      System.out.println(i);
   }
}
```

Snippet 7: Transforming a loop into a non-deterministic loop

6 Challenge Programs and Vulnerabilities

In this section, we describe all of the challenge programs and vulnerabilities that CyberPoint developed for the STAC program.

These challenge programs were featured in a sequence of engagements, under the direction of Apogee Research, the Engagement Lead, where the blue teams were challenged to use their tools to detect vulnerabilities or to convince themselves of the lack thereof, in a limited amount of time.

Engagement 1 was a live engagement, where blue teams had only three days to analyze the challenge programs. Engagement 2 was a take-home engagement, where they were given three months to continue their analysis on the same challenge programs (modulo a few fixes of unintended vulnerabilities).

Likewise, Engagement 3 was a two-day live engagement, and Engagement 4 was a three-month take-home continuation with the same challenges (again, with fixes for unintended vulnerabilities in between.)

Engagement 5 comprised a two-week take-home portion, after which blue teams provided individual answers, followed by a three-day live collaborative engagement. Engagement 6 followed the same format (with a fresh set of challenges).

Finally, Engagement 7 comprised a one-week take-home analysis portion for a fresh set of challenge programs, followed by a live collaborative portion. In this engagement, blue teams did not have to provide individual answers after the take-home portion; instead, a set of five blue teams competed against the control team, who, using only off-the-shelf tools, had an additional two weeks of analysis time. Like the control team, the University of Utah team was also excluded from the collaborative engagement and performed independently.

In the following, we present the challenge programs chronologically, starting with those that appeared in the first engagement, and ending with those that first appeared in the final one. The corresponding source code, proofs, and instructions for running them can be found at Apogee Research's STAC GitHub repository [1].

6.1 GabFeed

GabFeed, which appeared in both Engagement 1 and 2, is a message board that allows people to communicate with each other. Messages get posted to public rooms, where all messages are also public. Some rooms are attributed, where messages are associated with specific users, and other rooms are anonymous, where messages are never attributed to a user. Users can also search for messages across all rooms and can view all of their own messages (including those posted anonymously).

6.1.1 Intended Vulnerabilities and Mitigations in GabFeed

6.1.1.1 Line Break (AC-T)

Being a message board application, GabFeed needed to be able to process line breaks in user messages. We use the shortest path algorithm described in [12]. This algorithm has O(n `*` `width`) behavior, where n is the number of words, and `width` is the width, in pixels, of the column where the text will be displayed. Its behavior changes to O(n^2) when the `width` is at least as long as the number of words. Normally, there is a JavaScript function that determines the width, in pixels, of the display column where the text will be, and determines the text width based on that number of pixels. The function then issues

a PUT to `https://localhost:8080/width/<found width>`, which sets the width. In benign use of the application, the width should never be large enough to trigger any vulnerabilities because it is based on how large the screen and browser are, which should always be relatively small. However, it is possible to set the width manually by issuing a PUT to the appropriate URL with any desired width. The attacker can set an exceptionally long width, and then post a long message to trigger a timeout.

In Engagement 2, four teams correctly identified the vulnerability, one team answered the question incorrectly, and four teams elected to not answer the question. Interestingly, all four of the teams that successfully detected the vulnerability found that it was not necessary to set the width in the PUT message in order to exceed the resource usage limit. Instead they found that submitting a POST containing a large amount of short words separated by spaces would allow them to exceed the resource usage limit.

We also provided a benign version that uses the divide and conquer algorithm described in [12]. This algorithm has O(n log n) behavior and thus mitigates the AC-T vulnerability described above.

6.1.1.2 Search (SC-ST)

GabFeed contains a feature that offers, each day, a set of special search terms. When a user searches for one of these terms, GabFeed provides educational information on the term, as well as the messages containing that term. Other GabFeed operations are distinct from these special searches in both request size and timing. There is also a significant difference in response times to special search terms and ordinary searches, allowing an attacker to detect when a user has searched for one of these special terms.

Given the distinct sizes of the educational information, an attacker can readily determine which term a user searched for.

In Engagement 2, blue teams were asked to determine whether there was a SC-T vulnerability allowing an attacker to determine how many special search terms a user searched for within the operational budget. Three teams correctly identified this vulnerability, while six teams elected to not answer the question.

In Engagement 2, blue teams were asked to determine if there was a SC-ST vulnerability allowing an attacker to determine which special search term was searched for within the operational budget. Three teams correctly identified this vulnerability, two answered the question incorrectly, and four teams elected to not answer this question.

6.1.1.3 Finite-Field Diffie-Hellman (SC-T)

Before logging in, the client can validate whether the server it is connected to is the real one and not an imposter using the Finite-Field Diffie-Hellman (FFDH) [13] key exchange. The server has a secret key s, and has securely distributed its public key, $p = 2^s \pmod m$, to all clients, where m is the modulus the server uses in the key exchange calculations, which the client has no control over. The client application chooses a random value a and computes $p^a \pmod m$. It also transmits $2a \pmod m$ to the server, which uses it to compute

$$(2^a)^s \pmod m = (2^s)^a \pmod m$$
$$= p^a \pmod m,$$

and sends it back to the client. The client knows that only a server in possession of s could correctly perform this computation and then it can log in with confidence.

The encryption function invokes the `ModPow(a, s, m)` procedure, where the first argument, *a*, is controlled by the user, the second argument, *s*, is the server's private key, and the third argument, *m*, is the publicly known modulus used in key exchange calculations. For each bit in the secret, the attacker can provide a specific input that makes the computation take noticeably longer if that bit is 1. The costs of this expensive computation can be influenced by crafting *a* using *m*. Knowing this, the attacker is able to deduce *s* through a timing attack, allowing the attacker to impersonate the server and perform man-in-the-middle attacks on all traffic.

In Engagement 2, blue teams were asked to determine whether there was a SC-T vulnerability allowing an attacker to determine the server's private key used in the server's authentication. Two teams correctly identified this vulnerability, four teams incorrectly answered the question, and three teams elected to not answer the question. Interestingly, one of the teams who answered incorrectly thought the SC was drowned out by our non-deterministic loop transform.

We also provided a benign version of FFDH that uses *Montgomery's Ladder* [14] in the implementation of `ModPow`. This implementation eliminates the time difference in computation times for 0 bits and 1 bits of the secret, making the attack used for the vulnerable version inapplicable.

In Engagement 2, blue teams were asked to determine whether there was a SC-T vulnerability allowing an attacker to determine the server's private key used in the server's authentication. Four teams correctly identified that the SC was too weak, one team incorrectly answered the question, and four teams chose not to answer the question.

In Engagement 2, blue teams were asked to determine whether there was a SC-S allowing an attacker to determine the server's private key used in the server's authentication. Three teams correctly identified that the SC was too weak and six teams chose not to answer the question.

6.1.1.4 Hash Table - Unbalanced Tree (AC-T)

GabFeed uses a hash table to store all chat rooms and users. This hash table uses an unbalanced tree and, as a result, collisions can result in bad performance (e.g. if the items are added in sorted order). While a plain text file with collisions isn't large enough to trigger the vulnerability within the input size budget, a compressed file enables an attacker to send enough data to cause the program to use excessive computation time.

In Engagement 2, four teams correctly identified the vulnerability, while five teams elected to not answer the question.

6.1.1.5 Hash Table - Bad Red-Black Tree (AC-T)

GabFeed uses a hash table to store all chat rooms and users. This hash table uses a red-black tree that, on PUTs, fails to balance the tree and is therefore vulnerable to denial of service. While a plain text file with collisions isn't large enough to trigger the vulnerability within the input-size budget, a compressed file enables an attacker to send enough data to cause the program to use excessive computation time.

In Engagement 2, three teams correctly identified the vulnerability, one team answered the question incorrectly, and five teams elected to not answer the question. Interestingly, the three teams that answered correctly referenced their solution to our other hash table that is vulnerable due to an unbalanced tree, to justify their answer. After Engagement 2, we stopped allowing different intended vulnerabilities to be exploited by the same input.

6.1.1.6 ChangingSort - Bad Partitioning (AC-T)

Several of our challenge programs included a polymorphic sorting algorithm, `ChangingSort`, based on `MergeSort` [15] with a variety of variants configurable via Jinja.

The Bad Partitioning vulnerability lies in the building of the queue of array partitions to be processed and causes the `ChangingSort` algorithm to have worst case $O(n^2)$ time complexity. After an index is selected at which to partition the array, it is added to the partition index queue that will be processed when merging later in the algorithm. However, if the input is of a certain length, the calculated index will be added to the queue twice, causing inefficient run time. This vulnerability does not depend on the contents of the collection being sorted, but on the size of the collection.

In Engagement 2, three teams correctly identified the vulnerability, two teams answered the question incorrectly, and four teams elected to not answer the question.

We also provided a benign version where the partitioning is done correctly, regardless of the input size.

6.1.2 Unintended Vulnerabilities in GabFeed

In Engagement 2, we provided blue teams with a benign variant of the line break vulnerability. One blue team was able to get around the input budget by using newline characters instead of spaces—each newline creates 12 characters.

In Engagement 2, a team claimed to have found an SC that reveals the username of a user involved in a private chat. This SC involved using packet contents rather than sizes and was therefore ruled out of scope as a STAC SC-S vulnerability.

6.2 Graphr

Graphr is a standalone application for the editing and analysis of graphs. It includes support for breadth-first and depth-first search, determining the edge capacity between two nodes, determining whether or not the graph is connected and/or bipartite, and finding the shortest path between two nodes. The development of Graphr was motivated by the plethora of graph algorithms with algorithmic complexity vulnerabilities.

At one point, we believed Graphr had been omitted from the engagement because it didn't have a satisfactory user story—why would anyone want to use it? So, for Engagement 3, we developed AirPlan, an application with a clearer use scenario, which contained the majority of the Graphr code. See section 6.5.

6.3 SnapBuddy

SnapBuddy is a web application for image sharing. Users can log in to SnapBuddy to upload photos to share with their friends. Photos can be marked public in which case any user of the system can view them (all profile photos are public). Photos that aren't marked public are only viewable by the owner and their friends. In addition to uploading photos, users can apply filters to their photos to change how they look. When a user logs in, they are required to set their initial location by submitting a list of Basic Service Set Identifiers (BSSIDs) associated with their current location. The server will look up the city associated with the user list of BSSIDs, and allow users to confirm that the provided city is the location they would like to choose. Once their location is set, they can see the people around them in that location (location changes can only occur a certain number of times per day).

6.3.1 Intended Vulnerabilities and Mitigations in SnapBuddy

6.3.1.1 Filter Handler - Incorrect Filter Repeat Prevention (AC Memory)

The Filter Handler limits the total number of filters that can be applied to a single image to four, and attempts it to prevent the user from using the same filter on an image more than once. However, the user is able to apply the same filter more than once because the check to see if the filter is already applied is case-sensitive, but the code that retrieves the filter and applies it to the photo is case-insensitive. Thus, the attacker can increase an image's size by applying the ScaleUp filter multiple times, changing the case of its identity string for each application of the filter (e.g. F00E, F00e, f00E, f00e).

In Engagement 2, only two teams chose to answer this question, neither of whom detected the vulnerability.

We also provided a benign version where the code that applies the filter is case-sensitive. In this case, the case-sensitive check to see if the filter is already applied works correctly, successfully mitigating the vulnerability.

6.3.1.2 Filter Handler - Unlimited Filters (AC-T)

While the web page only allows the user to apply a certain number of filters, the Filter Handler does not impose this limitation when handling a POST. Thus, an attacker can use a malicious client to construct their own post to add a large number of filters to an image, causing the sample to exceed the allotted time.

In Engagement 2, blue teams were asked to determine whether there was an AC-T that would cause the challenge program to exceed the resource usage limit. Only four teams chose to answer this question, all of whom correctly identified the vulnerability.

In Engagement 2, blue teams were asked to determine whether there was an AC memory vulnerability that would cause the challenge program to exceed the resource usage limit. Two teams correctly identified that the challenge program did not contain an AC memory vulnerability, one team found an unintended vulnerability (further explained at section 6.3.2), and six teams chose to not answer the question.

6.3.1.3 Friends (SC-S)

Recall that in SnapBuddy, the profile photo for every person is public. A SnapBuddy user is permitted to see the public profile photo of every other user by displaying either their friends or the users that can be invited to be friends. However, when retrieving thumbnail images for profile photos in the list, the Scale pre-filter is never applied. As a result, the size of each profile photo image can be mapped directly to a user. If there are collisions, the identity cannot be resolved from the profile photo image information alone, in which case we assume the attacker can make an oracle query to find the correct user. In this data set, the maximum overlap is two users, so an attack can be successful in at most two operations.

In Engagement 2, blue teams were asked to determine whether there was a SC-S allowing an attacker to identify any one (or more) of a specific user's friends. Only three teams chose to answer this question, all of whom correctly identified the vulnerability.

In Engagement 2, blue teams were asked to determine whether there was a SC-T vulnerability allowing an attacker to identify any one (or more) of a specific user's friends. Two teams correctly identified that

there was no SC-T vulnerability, two teams incorrectly answered the question, and five teams chose to not answer the question.

6.3.1.4 Location (SC-S)

In SnapBuddy, a user is able to view each location and it's associated BSSIDs at `http://localhost:8080/cities`. When initializing or updating their location, users must provide all of the BSSIDs associated with the location they would like to use. The BSSIDs are padded to 16 bytes in order to work around the padding that is present in the SSL encryption being used by SnapBuddy. As a result, the size of each POST request to change a user's location can be mapped uniquely to the location associated with the BSSIDs in the request. If there are any collisions, the response size containing the location name can be used to disambiguate between the various responses.

In Engagement 2, only three teams chose to answer this question, all of whom correctly identified the vulnerability.

6.3.1.5 Finite-Field Diffie-Hellman (SC-T)

Refer to section 6.1.1 in the GabFeed description for details.

In Engagement 2, one team correctly identified the vulnerability, three teams did not detect the vulnerability, and five teams chose to not answer the question.

6.3.2 Unintended Vulnerabilities in SnapBuddy

In Engagement 2, we provided blue teams with a variant only containing the Incorrect Filter Repeat Prevention vulnerability. One blue team claimed to have found a vulnerability where they were able to bypass the limit on the total number of filters, and apply the same filter, `filter list=F00E` six times. Apogee was not able to recreate the problem, but they believed it was possible that the exploit might have worked due to garbage collection.

6.4 TextCrunchr

TextCrunchr, which appeared in both Engagement 1 and 2, is a text analysis program. It can perform some useful analysis (word frequency, word length, etc.) and includes processing tools too (automatic Enigma encryption of input). TextCrunchr can process both plain text and compressed files.

6.4.1 Intended Vulnerabilities and Mitigations in TextCrunchr

6.4.1.1 Zip Bomb (AC-T)

When given compressed files, TextCrunchr first decompresses the file, and then it analyzes every file contained in the compressed file. The Zip Bomb vulnerability lies in this first step of decompressing the compressed file. Our zip file decompression recurses on any zip file it encounters that contains embedded zip files. We provide a guard to limit the depth at which it recurses, however, the guard is poorly implemented and checks the size of a queue whose size does not accurately indicate the total amount of files processed. This means that a zip file that decompresses to itself, for instance, could cause the decompressor to run forever.

In Engagement 2, two teams correctly detected the vulnerability, two teams did not detect the vulnerability, and five teams elected to not answer the question.

6.4.1.2 Standard Vulnerable Hash Table (AC-T)

TextCrunchr uses a hash table to store words from text files and associate those with different statistics, depending on the processor being used (word counts, etc.). This hash table uses a standard, linked-list hash table vulnerable to denial of service. The vulnerability lies in the collision handling of the hash table. When a new key-value pair hashes to a bucket that is already filled, it is added to that bucket's linked list of key-value pairs. If the new key is already present in the list then the pre-existing value is updated, if not, every other key in the list is checked before appending the new pair to the end of the list. While a plain text file with collisions isn't large enough to trigger the vulnerability within the input size budget, a compressed file enables an attacker to send enough data to achieve $O(n^2)$ performance and cause the program to exceed the time budget.

In Engagement 2, six teams correctly identified the vulnerability, while three teams elected to not answer the question. Interestingly, one team found and exploited the intended vulnerability while incorrectly claiming the vulnerability was in the sorting.

6.4.1.3 Hash Table – Bad Red Black Tree (AC-T)

Refer to section 6.1.1 in the GabFeed description for details.

In Engagement 2, five teams correctly identified the vulnerability, two teams answered the question incorrectly, and two teams elected to not answer the question.

6.4.1.4 ChangingSort – Bad QuickSort (AC-T)

Several of our challenge programs included a polymorphic sorting algorithm, `ChangingSort`, based on `MergeSort` [15] with variants configurable via Jinja.

The Bad QuickSort vulnerability causes `ChangingSort` to have $O(n^2)$ time complexity via an added `QuickSort` [16] step to the sorting algorithm. In this variant, `MergeSort` is incorrectly implemented, so that this step is used to shuffle the given list, rather than sorting it. This is done by always placing the elements of the right partition before the elements of the left partition, when merging, until the length of the partition being sorted is equal to $\max(2^{10}, 2^{(k+7)})$, where k is a constant whose value can be configured via Jinja. After the input has been shuffled, it gets passed to the `QuickSort` step, which has a worst-case complexity of $O(n^2)$. In our `QuickSort` implementation, we always select the right-most element of the partition to be the pivot point. Therefore, if the given list is already sorted, divide-and-conquer will fail to add any benefit because it will still take a long time for each partition to be recused on until the partition is of length one. However, due to the permutation that takes place in the `MergeSort` step, an attacker can't exploit the vulnerability by simply passing a sorted input to the program; the vulnerability can only be triggered by inputting a list shuffled in such a way that the list is permuted to a sorted list, prior to the `QuickSort` step.

In Engagement 2, four teams correctly identified the vulnerability in the `QuickSort` aspect of the `ChangingSort` algorithm, while five teams elected to not answer the question.

We also provided a benign version where `MergeSort` is implemented correctly and there is no added `QuickSort`, giving the algorithm worst-case $O(n \log n)$ complexity.

6.4.1.5 ChangingSort - Bad Partitioning (AC-T)

Refer to section 6.1.1 in the GabFeed description for details.

In Engagement 2, four teams correctly identified the vulnerability in the partitioning aspect of the `ChangingSort` algorithm, one team answered the question incorrectly, and four teams elected to not answer the question.

6.4.2 Unintended Vulnerabilities in TextCrunchr

In Engagement 1, there was an unintended AC-T vulnerability due to an insufficient guard on the total decompressed file size and not checking individual file sizes after decompression. One blue team was able to exceed the resource bound by crafting input, spread across four files, that was small enough to pass the guard on total decompression size.

6.5 AirPlan

AirPlan is a web-based service to aid airlines in planning flight schedules. An airline can submit and modify graphs of their proposed flight routes for analysis, to find the best flight paths and schedules. AirPlan, appearing in Engagements 3 and 4, was designed as a more practical setting for the vulnerabilities in Graphr (see section 6.2), with not only a more natural use case, but also the ability for an attacker to cause harm to someone other than themselves. In AirPlan, airports are represented as vertices, with flights represented as directed edges, mapping the problem space neatly into the domain of Graphr.

6.5.1 Vulnerabilities and Mitigations in AirPlan

6.5.1.1 Maximum Flow AC-T

A maximum flow algorithm [17] is used as part of the crew scheduling algorithm, which determines the smallest number of crews that can be used to cover all of the flights in a given flight map. In a graph where edges are assigned capacities, the maximum flow problem is the problem of finding the greatest "flow" that can be passed from a single source node to a single sink node in the graph. For example, if the edge capacity represents the number of passengers that can be accommodated on a flight, then the maximum flow between Boston and Los Angeles is the number of passengers that can be moved from Boston to Los Angeles utilizing all possible flights. For crew scheduling, the maximum flow computation is done on a graph derived from a graph of the airports and flights.

The Ford-Fulkerson method [17] for determining the maximum flow in a graph has worst-case algorithmic complexity $O(E * f)$, where E is the number of edges and f is the maximum flow. Generally, Ford-Fulkerson runs much faster: as the capacity of the augmenting path gets close to the maximum flow, the runtime approaches $O(E)$. On each iteration, the algorithm looks for a path, called the augmenting path that will add additional capacity to the current flow. The worst-case algorithmic complexity arises when the capacity of the augmenting path is very small, but the maximum flow is extremely large. Note that Ford-Fulkerson is more accurately called a method than an algorithm, as it does not fully specify the approach to choosing the next augmenting path.

While textbooks provide a standard example graph, shown in fig. 6-1 that can allegedly exhibit worst-case time performance, no natural implementation of the method would ever exhibit such performance, as it relies on making the worst-possible decision for the augmenting path at every iteration. We have engineered the order in which our edges are explored so that there exists an example on which such worst-case behavior is exhibited, but the textbook example is handled with ease. Only for a malicious graph similar to the classic textbook example, but with many additional edges prior to the source and sink, will the algorithm exhibit the worst-case performance.

In particular, the algorithm we use to find the augmenting path searches the graph in a depth-first manner. However, the order in which the edges from a given vertex are explored matters. In our algorithm, this is generally done in order of increasing edge ID. Edges that exist in the actual graph are assigned odd IDs starting from 1, and backward edges (that only exist for reverting a flow in the algorithm) are assigned even IDs, starting from 200.

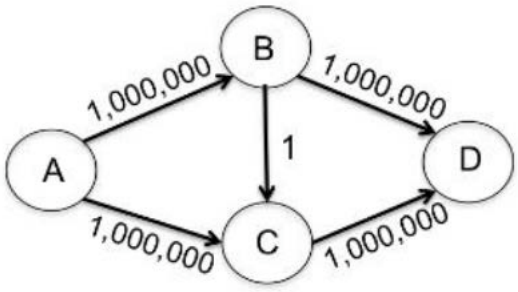


Figure 3: Standard example graph that allegedly exhibits the worst-case performance of the Ford- Fulkerson method.

Exception to this order of exploration is that, when exploring edges from the source node, it alternates, every other time exploring the edges in either ascending or descending order. This alternation is necessary to attain the worst-case complexity in the standard textbook example. To achieve the worst case with this example, it is also necessary to explore the backward edge ($C \rightarrow B$ in fig. 6-1) whenever it has capacity. Thus, by assigning the backward edges high IDs, we prevent the classic textbook example from being vulnerable in our implementation, and we hoped this would make the blue teams have to work to determine whether or not there was an exploitable vulnerability.

In Engagement 3, one of the three blue teams who answered the question correctly identified that there was a vulnerability in the vulnerable method described above. However, their response merely identified that the method was Ford-Fulkerson, and therefore it was vulnerable, without looking at the details of the implementation or attempting an exploit.

Another blue team instead found an unintended vulnerability in Engagement 3.

In Engagement 4, none of the blue teams identified the vulnerable version of Ford-Fulkerson as vulnerable. Instead, four identified unintended vulnerabilities, while one named an unintended vulnerability that had existed in Engagement 3, but had since been eliminated.

In both Engagements 3 and 4, we also included an Edmonds-Clark [17] implementation of maximum flow, which has worst-case algorithmic complexity $O(V * E^2)$, where E is the number of edges and V is the number of vertices. Since AirPlan limits the number of edges and vertices in the graphs it processes (500 each in Engagement 4), this does not contain an algorithmic complexity vulnerability. This implementation existed in variants that contained other AC-T vulnerabilities, and no blue teams believed there to be a vulnerability in this portion of the code.

6.5.1.2 Shortest Path (AC-T)

AirPlan allows a user to determine the shortest path (in terms of distance, cost, time, etc.) between two airports. We provided several different shortest path algorithms.

Dijkstra
Dijkstra's shortest path algorithm [18] contains an AC-T vulnerability when there are cycles in the graph

whose total edge weight is negative. The algorithm iteratively looks for a shorter path between the source and sink nodes, and will enter an infinite loop, repeatedly adding the cycle to the path and thus reducing the weight of the path.

We also provided a benign version that rejects graphs containing negative weight cycles, eliminating the vulnerability.

*A**
The time complexity of the A* shortest path algorithm depends on the heuristic used to determine which edge to explore next. While an optimal heuristic results in a linear complexity, a poor choice of heuristic results in exponential time. For an AC-T vulnerability, we used an inconsistent heuristic dependent only on the node ID, which leads to exponential computation time, $O(2^n)$ where n is the length of the shortest path, for very specific graphs, as described in [19].

For our benign variant, we used the consistent, admissible heuristic of the minimum outgoing edge weight.

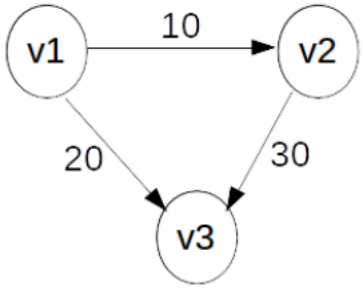
Unfortunately, the blue teams found unintended vulnerabilities in both Engagements 3 and 4 that prevented them from even trying to find either of these vulnerabilities.

6.5.1.3 XML Graph Loading (AC Memory)

The XML parser is vulnerable to the "Billion LOLs" attack [20] wherein XML entity references are expanded without bound, allowing the attacker to force exponential behavior. The same (or slightly modified) vulnerability allows for recursive XML entity declarations.

In Engagement 3, only one blue team answered this question, and they believed the challenge to be benign in terms of memory usage. In Engagement 4, only one blue team detected this vulnerability, while two other teams found an unintended vulnerability.

We also had a benign variant that limits itself to 64,000 entity expansions, but unfortunately, there was an unintended memory vulnerability. Two blue teams found it, while one didn't find any vulnerability.



(a) Example graph

	v1	v2	v3
v1		10	20
v2			30
v3			

(b) Graph matrix for example graph

Figure 4: A graph and its matrix representation.

6.5.1.4 Changing Sort (AC-T)

Engagements 3 and 4 featured a variety of recursive sorting algorithms, which were all instances of a polymorphic sorting algorithm, implemented with Jinja. These were designed to challenge the blue

teams' ability to identify and solve complex recurrence relations describing the runtime of an algorithm. To configure a sorting algorithm, one merely specifies *k*, a runtime of good or bad, and a sort of halving, uneven, or combined. *k* determines how many sub-lists a list is split into for recursive processing. In halving sort, the list is divided evenly, while in uneven sort it is not. Finally, if bad sort is selected, there is a modification that causes the runtime to reach $O(n^2)$ or higher. In halving sort, this is done by partitioning 3/4 of the array twice if the list length is a multiple of *k*. In uneven sort, this is done by calling partition on the longer sub-array twice if the length of the list is a multiple of *k*. These same algorithms appeared in Engagement 1 and 2 (see section 6.1.1) implemented with an implicit stack to prevent the excessive memory use inherent in extensive recursion.

Three blue teams detected the uneven sort vulnerability, while three found an unintended vulnerability.

6.5.1.5 Graph Size (SC-S)

AirPlan displays the graph submitted by a user in a "standard" form, which reveals the size of the graph. As shown in the example in fig. 6-2, the graph is displayed in matrix form, with a row and column for each node, with the capacity between each pair of nodes. Both vertex names and capacities are padded so that the number of nodes uniquely determines the size of the overall content—the size of this table is on the order of n^2, where *n* is the number of nodes. An observer can therefore determine the size of the graph uploaded by the size of the server's response packet.

All seven of the blue teams that answered this question correctly identified this vulnerability.

Another version adds random padding to vertex names and capacities, while a third simply omits padding to mitigate the vulnerability. Most of the blue teams who attempted the question correctly identified the benign and vulnerable versions of this vulnerability.

The blue teams were only asked about the random padding benign version. Five blue teams correctly identified this as benign, while one believed it to still be vulnerable.

6.5.1.6 Connectedness (SC-S)

AirPlan displays route map properties in such a way that the size of the packet reveals whether or not the map is connected. Again, this secret may be of interest as the possibility moving from a disconnected flight graph to a connected one represents a significant change in service that would interest a competitor or stock holder.

Each of the properties is formatted, with padding, to a certain size. This size is initially set to be 19 characters, large enough to contain all of the properties strings. However, when formatting the properties, there is a check to ensure this space is sufficient; this check is incorrectly implemented with a < where there should be a ≤. If the space is deemed insufficient (which it never would be with the ≤), the space used for the current property (and each subsequent property) is doubled. Because the disconnected graph property string takes up exactly 19 characters, in the case of a disconnected graph, the broken guard mistakenly allocates more space for this property (and all subsequent properties), making the properties packet detectably larger, and yielding distinct, identifiable size ranges for graph properties pages for connected and disconnected graphs, respectively.

Five blue teams detected this vulnerability, while two believed there was no such side channel.

Another version correctly uses the ≤, while a third adds random padding. The blue teams were only asked about the former. Five blue teams correctly identified this version as benign, while one believed there was still a side channel.

6.5.2 Unintended Vulnerabilities in AirPlan

6.5.2.1 Loading Large Graphs

In Engagement 3, there was an unintended AC-T vulnerability in the loading of a large (3800 node) graph from a text file. We corrected this for Engagement 4 by reducing the allowed number of nodes and edges in a graph.

6.5.2.2 Crew Scheduling not Guarded

In Engagement 4, the crew scheduling contained an unintended AC-T vulnerability, as it allows exceeding the graph size limitations imposed on user graphs.

*6.5.2.3 Negative Weight Cycles in A**

Unfortunately, we missed the fact that A*, like Dijkstra's algorithm, fails to handle negative weight cycles, and this left us with an unintended vulnerability in the variants of AirPlan that used the A* algorithm for shortest paths.

6.5.2.4 Large Graph Display

Finally, the code for the graph size SC-S provided an unintended AC-T vulnerability with repeated requests for the display of a graph of maximal allowed size.

6.6 BidPal

BidPal is a peer-to-peer application that allows its users to buy and sell items via a highest-bidder-wins protocol. A user that wishes to sell an item can start an auction for that item. Other users that wish to buy that item can submit a bid for that item. This bidding protocol is done in a manner such that bids (other than the winning bid) are not revealed to other users; users only learn whether their bid is at least as high as the others. After the seller closes the auction, it announces the winner (seller's choice if there is a tie).

6.6.1 Intended Vulnerabilities and Mitigations in BidPal

6.6.1.1 Auction Bid (SC-T)

Since users' bids are secret, the bidding protocol is implemented so that those bids are never revealed. There is, however, a timing side channel that allows an attacker to discover a user's bid. A user's bid is leaked through the creation time of the bid comparison message. After a user submits a bid, a bid comparison message is generated. That bid comparison message is sent to the other users that bid in the same auction to allow them to determine if their bid is greater than or equal to the sender's bid, without revealing the bid amount.

There are two different versions of this vulnerability. To create the bid comparison message, there is a loop that sets the message array. In one version of this vulnerability, the loop is indexed from the user's bid to the maximum amount, with an expensive method called each iteration, making the bid amount proportional to the time it takes to create the comparison message. In the second version of the vulnerability, the loop iterates from zero to the maximum amount, however, once the loop iteration reaches the bid amount, the expensive method is called on each iteration, again making the bid amount proportional to the creation time of the bid comparison message.

In Engagement 3 and 4, BidPal contained the second version of this vulnerability. In Engagement 3, one blue team correctly found it. In Engagement 4, two blue teams were able to correctly identify the vulnerability.

In one benign version, the bid comparison method is fast and there is no correlation between the bid amount and the time it takes for the method to complete. In another benign version of this side channel, the bid comparison method generation is slow, but it is still independent of the bid amount i.e. the expensive method is still called with every index in the bid comparison message.

Questions about the benign versions of this vulnerability in BidPal were not asked.

6.6.1.2 Auction Checksum (AC-T)

In normal use of this application, when an auction is created, a randomly-generated ID is automatically assigned to it. An unadvertised feature of this application is for a user to create an auction with an ID of their choice. During an auction, checks are performed on the bytes of serialized bid comparison messages received from each bidder, in the guise of a sort of checksum to ensure the message is valid. Under the normal use of this application, all these checks should always pass with the generated auction ID. If the check fails, then the program will continuously attempt to recreate the bid comparison message, using the same parameters, causing it to repeatedly fail. With the right username and auction ID, an attacker can trigger these checks to fail causing an infinite loop that will cause the resource time limit to be exceeded.

In Engagement 3, no team found this vulnerability and in Engagement 4 only one team found the intended vulnerability, while three other teams triggered this vulnerability with unintended input.

In the benign version, we attempted to implement the checks on the bid comparison message so that they do not fail with any auction ID or username. However, two blue teams found a way to cause the checks to still fail.

In Engagement 3, two teams attempted to answer the null question for this vulnerability and only one got it correct (the other team gave no answer). In Engagement 4, three teams found an unintended input that triggered the checksum AC-T, one team did not find any vulnerability, and the others did not attempt the question.

6.6.2 Unintended Vulnerabilities in BidPal

In Engagement 3, an attacker could have a very long user ID, and logging that user ID many times would cause the disk space usage limit to be exceeded.

In Engagement 4, two blue teams found that the auction checksum vulnerability can be triggered by submitting a bid of 0 for any auction, which was not intended to be the triggering input in the vulnerable version; further, this input also triggered the vulnerability in the version that was intended to be benign.

In Engagement 4, if a few benign users end up creating a large number of auctions, then an attacker can bid on all of them. This would cause a denial of service due to the wait time of logging and sending bid comparison messages. This was discovered by one blue team.

6.7 PowerBroker

PowerBroker is a peer-to-peer application presented to the blue teams in Engagements 3 and 4. The users are electric power suppliers. Power suppliers aim to buy power if they have demand for it or sell power if they have an excess of it. An auction is initiated by an offer to sell power, and those who wish

to buy it can submit their bids. Auctions are decided using the same auction protocol from BidPal, a highest-bidder-wins, secure multiparty computation protocol that allows the winner to be determined without anyone knowing anyone else's bid.

6.7.1 Intended Vulnerabilities and Mitigations in PowerBroker

6.7.1.1 Overlogging (AC-Disk)

An attacker can cause the victim to repeatedly attempt to connect to him, which causes the logging of each connection attempt to disk, eventually exceeding the disk-space budget. This vulnerability occurs in the connection protocol. The attacker first connects to the victim by sending him a connect message. After sending the message, the attacker disconnects from the victim. Before deciding to move to the next phase of the auction, the victim's application checks if it is still connected to all other users participating in the auction. Once it sees that the victim is not still connected to the attacker, it attempts to reconnect. The reconnect process is recursive and includes a bug where the condition to end the recursion is never fulfilled, therefore resulting in the victim to repeatedly trying to connect to the attacker. The condition to end the recursion is a limit on the number of attempts that can be made to connect, however, the counter for the number of attempts is never updated so that limit is never reached.

In Engagement 3, only one team attempted questions regarding this vulnerability, and that team found an unintended vulnerability (described below). An attempt was made to mitigate that unintended vulnerability after Engagement 3; however, in Engagement 4, three teams were still able to trigger this vulnerability (section 6.7.2). Only one team found the intended vulnerability in Engagement 4.

In the benign version, the counter for the number of connection attempts is updated so the limit is reached, thus preventing the disk-space limit to ever exceed with connection attempt logs within the allotted input budget.

In Engagement 3, only one team attempted the null question for this vulnerability and that team found unintended input that triggers the vulnerability. In Engagement 4, four teams found an unintended vulnerability and two teams said there was no vulnerability.

6.7.1.2 Auction Bid (SC-T)

PowerBroker features the first version of the auction bid side channel that appeared in BidPal. Refer to section 6.6.1 in the BidPal description for details.

In Engagement 3, only one team attempted a question regarding this vulnerability and that team successfully found it. In Engagement 4, one team incorrectly answered the question, three teams correctly identified the vulnerability and one team answered yes to there being a side channel but their explanation involved an unrelated method and was incorrect.

Regarding the null question for this vulnerability, in Engagement 3, only one team answered the question and got it wrong. In Engagement 4, three teams correctly answered the question and two teams answered incorrectly. One team that thought there was a vulnerability present had an exploit that did not work.

6.7.1.3 RSA (SC-T)

Several of our applications used a custom implementation of the RSA protocol [2]. RSA is a public-key encryption standard that leverages large prime numbers in its key generation. In most of these

applications, we believe that the implementation is not vulnerable within the provided challenge program budgets. In PowerBroker, however, we implemented a SC-T vulnerability involving RSA.

In PowerBroker, during the connection setup between users of this application, there is a step where two users (call them “Alice” and “Bob”) exchange RSA public keys. Alice will challenge Bob to make sure that he is really in possession of the secret key associated to the transmitted public key. To do this, she sends him a random value, encrypted with his public key, and asks that he decrypt it and send her back the value as proof. If he answers incorrectly, she can send another challenge, in case of something like a dropped packet or flipped bit. This allows an attacker to issue the many decryption requests necessary for the attack. Note that Bob challenges Alice in the same way, but it is only necessary to analyze one direction for the purposes of the attack.

The vulnerability here lies in the timing of the decryption. This timing leaks a multiple of one of the primes used in the RSA protocol to a small degree of error. The decryption timing is vulnerable because we are using the Chinese Remainder Theorem version of RSA [21].

Because of this, the exponentiation is done modulo the secret primes. There are inefficient operations that slow it down enough to make the timing difference detectable, such as an extra reduction step during the Montgomery multiplication used in the modular exponentiation [21]. An attacker can map the decryption timings, by using a few statistical tricks [22] and inverting some of the operations above, to a multiple of one of the primes used in the protocol. One can narrow an interval around a multiple of the secret prime to the point that it is trivial to find it by performing a small number of greatest common divisor (GCD) computations against the public modulus.

Each correct run through the attack loop effectively halves the interval around the prime. Since the prime is 512 bits, an attacker needs at most 502 of these iterations to get him within about 1000 of a multiple of the prime. Because of repeats, the attacker can perform 120 observations per iteration, meaning that it needs roughly $502 * 120$ observations. The GCD computations then effectively act as oracle queries, (but are very fast compared to getting timings), so then with 1000 queries, discovering the secret prime should take no more than $502 * 120 + 1000 = 61240$ operations.

The attack is a slight variant of the one in a smart-card scenario [21]. Once the attacker successfully recovers the secret primes, it can then compute the components of the victim’s private key, impersonate him and decrypt messages intended only for him.

In Engagement 3, only one team found the intended vulnerability. In Engagement 4, two blue teams, including the control team, correctly identified the vulnerability.

In the benign version, the RSA decryption is potentially vulnerable, but it would require extra operations since the connection protocol does not allow the attacker to ask the target to decrypt multiple messages in a single handshake, which is required in the attack on the vulnerable version.

In Engagement 3, one team correctly answered the null question and one team incorrectly answered (they were convinced that the Montgomery multiplication routine somehow leaks a secret key). In Engagement 4, a null question for the RSA vulnerability was not asked.

6.7.2 Unintended Vulnerabilities in PowerBroker

In Engagement 3, one blue team found that auction IDs get logged repeatedly, and if there is a long auction ID, it will eventually cause disk space usage to exceed the limit. We attempted to fix this by truncating the auction IDs when they are logged for Engagement 4, however, three blue teams were able to find an instance where we neglected to truncate the auction ID and that was still exploitable.

In Engagement 4, it was found by a blue team and the control team that there was no limit on the number of auctions enforced on the client side. Auctions will continue until all demand for power is met. It is possible for an attacker to prompt many auctions and cause the disk-space usage limit to be exceeded due to the logging from repeated auctions.

6.8 WithMi

WithMi, appearing in Engagements 3 and 4, is a peer-to-peer chat program that supports text chat (both group chats and one-on-one) and file transfers. The development of WithMi was motivated as a setting for various compression-related vulnerabilities.

6.8.1 Intended Vulnerabilities and Mitigations in WithMi

6.8.1.1 File Transfer (SC-S)

WithMi's file transfer protocol splits a file into fixed sized chunks and then compresses them. This yields a unique size fingerprint for each of the included files (in spite of their all having the same original size.) Thus, an attacker observing network packet sizes can determine which file has been sent. Note: to make this vulnerability less obvious, random padding is added to mask the size of the sent chunks. However, it is added only after the end of the actual compressed file bytes, leaving the size signature intact in the first chunks sent.

Of the six blue teams that answered this question in Engagement 4, all but one correctly identified the vulnerability.

The benign version first compresses the file, then splits it into chunks, with random padding added at the end to mask the unique compressed size of the file. One blue team missed the random padding, while four correctly identified this version as benign.

6.8.1.2 Huffman Compression (AC Memory)

For this vulnerability, WithMi's file transfer protocol, by default, uses a compression algorithm that is a slight modification of Huffman encoding that can cause an out-of-memory error on compression of certain rare input files. Specifically, sequences of consecutive zeros in a file's Huffman encoding are expanded exponentially.

This vulnerability can be exploited by files with sufficiently long sequences of consecutive zeroes in their Huffman encoding. To make this more easily exploitable, the implementation also incorrectly tallies the frequency of null characters in the file.

This can be exploited by an attacker sending a malicious input file compressed with `zlib` compression, rather than the default algorithm so that the attacker doesn't burden their own system memory in sending the file. Alternatively, it could use a modified client without this vulnerability to send the file. When WithMi receives a file, it automatically archives it, compressed with the default compression algorithm.

Only two blue teams answered this question. One failed to find any vulnerability, while the other reported on a vulnerability that allowed an attacker to cause a stack overflow exception, but did not exceed the memory resource bound.

The benign variant of this correctly implements Huffman encoding without the exponential expansion of sequences of consecutive zeros.

Only one blue team answered the question on the benign version, and that was the team with the stack overflow exception.

6.8.1.3 Huffman Decompression (AC-T)

For this vulnerability, WithMi's file compression is done (by default) with Huffman encoding. In brief, this compression involves computing statistics over a file to create a code, represented by a trie created specifically for that file, which encodes frequent strings with shorter codes. A compressed file includes this trie, followed by the size of the uncompressed file, followed by the encoding of the file.

This version has an algorithmic complexity vulnerability due to two "bugs" in the implementation. First, method `readBoolean()` in `BinaryIn.java` mistakenly returns false, rather than throwing an error, when reading beyond the end of the stream. This allows an attacker to submit a malicious "compressed" file, causing the decompression algorithm to read beyond the end of the stream. There is a guard preventing the decompression algorithm from getting stuck in the trie-reading phase, forever looking for the leaves that terminate the trie – the program detects this and exits with an error message.

The program further attempts to prevent spending too much time in decompression by checking the overall length of the submitted file; however the `readBoolean()` method allows an attacker to get around that by submitting a "compressed" file that lists a longer size than the included data encodes; finally, the program has an incorrectly implemented guard against this: there is a bound on the number of characters to decode, but it erroneously only counts the characters decoded in the case where the encoding ended with a '1'.

Of the four blue teams who answered this question, two found an unintended vulnerability, and two failed to detect any AC-T vulnerability in this challenge. We would have liked to revisit the class of vulnerability where reading beyond the end of a file or data structure is possible in a future engagement, but due to the other vulnerabilities already planned, as well as a temporary move toward focusing on side channels, this did not happen. We suspect that this is an area in which the blue teams' tools may be lacking.

In the benign variant, neither of these two bugs is present.

Similarly to the vulnerable version, two blue teams answered with the unintended vulnerability, and two believed it to be benign.

6.8.1.4 RLE Decompression Int Overflow (AC-Disk)

For this vulnerability, WithMi uses run-length encoding (RLE) for "compression." This simply compresses consecutive identical bits within a file. E.g., "00000011100000" would be represented as "0-6-1-3-0-5".

There is an error in the guard that is supposed to prevent a file from being decompressed to greater than the specified maximum size of 108 bytes. Instead of a long, an int is used to track the number of bytes that have been written so far. This value can overflow, causing a failure to detect that the maximum size has been exceeded. To exploit this, an attacker can use a modified client to send a file representing a sequence of n 0's followed by m 1's, where $n < 108$ and $\text{Integer.MAX\_INT} < m + n < 10^8 + \text{Integer.MAX\_INT}$. (In our attack, we used $n = 99,999,999$ and $m = 2147483647$.)

Of the five blue teams that answered this question, three found an unintended vulnerability, and two failed to find any vulnerability. As such, we re-used the integer overflow concept in Suspicion (see section 6.18.1) in Engagement 7.

The benign version of this correctly uses a long to track the number of bytes decompressed.

Three blue teams found an unintended vulnerability in this variant, while two believed it to be benign.

6.8.1.5 RLE Decompression Unexpected Negative AC-Disk

For this vulnerability, WithMi decompresses received files and writes the resulting file with a loop as shown in snippet 8, where `num` is the count of consecutive appearances of byte `b` indicated in the compressed file. An attacker can provide a "compressed" file with a negative value for `num` (e.g. -1), causing this loop to continue writing the byte until `num` reaches `Integer.MIN_VALUE`, while not increasing (and, in fact, decreasing) the calculated size of the file, and thus getting around the guard that attempts to ensure that the number of bytes written is bounded.

```
while (num != 0) {
   write b;
   num--;
}
```

Snippet 8: The vulnerable loop.

Of the five blue teams that answered this question, three found the intended vulnerability, and two found unintended vulnerabilities. It may be worth noting that the blue teams did much better with this than with the integer overflow vulnerability, which was located in the exact same method in a different WithMi challenge.

The benign version of this uses `while (num > 0)` instead of `while (num != 0)` to avoid this vulnerability.

Three blue teams found an unintended vulnerability in this variant (the same variant described in the previous paragraph), while two believed it to be benign.

6.8.1.6 User Management (SC-T)

This side channel allows an attacker to determine whether two users connecting have previously connected. When a WithMi user receives a connection from a user it hasn't previously connected with, it stores that user's identity and public key on disk before responding, which causes a detectable time difference from the case of a user that it has connected with before.

All six blue teams that answered this question correctly identified the vulnerability.

In the benign version, the user information is also stored to disk, but the response message is sent before doing that, so an attacker has no way of measuring the time difference.

All six blue teams that answered this question did so correctly.

6.8.1.7 Chat Size (SC-S)

When a user is invited to a chat, they receive a chat state message. These messages are bigger than all other WithMi messages, and their sizes differ depending on how many people are currently in the chat, allowing an attacker to determine how many people are involved in a chat.

Of the four blue teams that answered this question, two correctly identified the intended side channel, while two argued that they could just look at how many ports/IP addresses a user was interacting with. The latter is incorrect, because it does not allow one to differentiate between a single chat between three users, {*A, B, C*} versus two separate chats {*A, B*} and {*A, C*}.

In the benign version, the chat state message is padded to mask the number of users.

Oddly, on this question, not just two, but four teams argued that they could look at how many packets were being sent to how many ports/ip addresses, while only two correctly answered that there was no vulnerability.

6.8.2 Unintended Vulnerabilities in WithMi

6.8.2.1 Shallow Trie (SC-ST)

In the Huffman compression version, if the root node of the trie is a leaf, data is written without ever reading anything beyond the length, which can be `Integer.MAX_VALUE`.

6.8.2.2 Exploiting Negative Run Lengths (AC-Disk)

In the integer overflow variant with RLE compression, an attacker can send bytes with negative run lengths. Nothing will get written to disk, but it will (incorrectly) reduce the tallied total size written, so that the attacker can send more positive run lengths to exceed the disk space bound.

6.8.2.3 Modified Client to Make the Most of the Input Budget (AC-Disk)

An attacker can modify their client to send larger chunks of data, in which case each chunk can use up to 100,000,000 bytes of the victim's disk space, with only a few bytes sent.

6.9 BattleBoats

BattleBoats is a two-player peer-to-peer game that appeared in Engagements 5 and 6. Like the popular game Battleship [23], each player has a set of ships that it places on grid squares in the "ocean", and the goal is to shoot the other player's ships (without being able to see where they are located). As an additional twist, in BattleBoats, rather than selecting grid squares to fire on, players place their shots by aiming the angles, height, and initial shot velocity of a cannon that has also been placed (in an unknown grid location) by their opponent. The game was designed as a setting for the use of Newton's method [24].

6.9.1 Intended Vulnerabilities and Mitigations in BattleBoats

6.9.1.1 Cannon Location (SC-T)

BattleBoats contains a timing side channel that allows a user to determine the location of their cannon which was placed by their opponent on the opponent's ocean board. More specifically, this side channel works via a timing difference when the user's shot is on or off the (ocean) board. Off-board shots take longer to process and therefore have a longer response time. By making progressively longer shots, the user can determine how far their cannon is from the edge of the board.

The attack works by firing successive shots at grid locations (0, 0), (0, 1), (0, 2), . . . , (0, B) relative to the cannon location (where B is the board length), to determine at which point the timing difference is manifest — this reveals the y coordinate of the cannon. (If a shot of (0, j) is the first off the board, it means that the cannon has a y-coordinate of ($B - j + 1$)). Similarly, shots of (0, 0), (1, 0), (2, 0) allow the user to determine the x-coordinate of the cannon.

This is facilitated by the command `shot_help`, which provides users with cannon aim parameters that will allow them to hit the desired grid location (relative to their cannon location).

It is interesting to note that this side channel is detectable/exploitable in the 3-node networked reference environment, but not in the previous single-node environment.

Eight blue teams answered this question in Engagement 5. Two teams failed to find the vulnerability. While six teams correctly identified the vulnerability, two of them didn't describe how one would exploit it.

We mitigated this vulnerability by reducing the timing difference in processing shots that land on versus off the board, to the point that the difference is lost in noise.

While six blue teams correctly answered this question in Engagement 6, one blue team believed the timing difference to still be exploitable.

6.9.1.2 Slow Convergence of Newton's Method (AC Memory (and also AC-T))

BattleBoats uses Newton's method to calculate the time until the fired cannon shot lands which is in turn used to compute where it lands. Newton's method is slow to converge under certain circumstances. This is controlled by the parameters of the opponent's shot (the initial value for Newton's method is fixed). As in the Newton's method (Category 13) canonical example [3] provided by Apogee, the trigger requires input with extremely small fractional values.

This version checks the sequential values computed to ensure that a computational cycle hasn't been entered, but the malicious input does not enter a cycle, so this does not prevent the attack.

This uses large amounts of memory because the code tracks the trajectory of the shot, in a hash map of `TrajectoryData` values, where the base class of `TrajectoryData` is actually the source of the huge memory use. The `TrajectoryData` class only holds a few `int` values, while its base class holds an array of `BigDecimal` values.

Seven blue teams answered this question in Engagement 5. Two of them did not find any vulnerability. One team found the intended vulnerability and used the intended exploit, while another found it and had a slightly different exploit which involved sending several shots with input slightly less malicious than we intended. Another team found an unintended vulnerability, using an unintended input format (described in detail in section 6.9.3). Finally, two teams believed there was a vulnerability but failed to adequately pinpoint it. It is interesting that only two teams identified this vulnerability, given that it was identical to a canonical example that Apogee had provided them.

A benign version checks for long runs of smaller and smaller changes to the value to detect slow convergence. This mitigation was used in the negative height version. See below for engagement results.

6.9.1.3 Negative Height in Newton's Method (AC-T)

Recall that BattleBoats uses Newton's method to calculate the time until the fired shot lands (which is in turn used to compute *where* it lands). In this version, inputs are not properly validated, and the inputs to the fired shot may be such that the height never reaches zero. This can happen if the initial height is negative, and the initial y velocity is insufficient to cause it to reach 0 (according to BattleBoats' idealized laws of mechanics.) Thus Newton's method loops indefinitely trying to find a root.

This version appears to validate the inputs — including checking for negative initial height values in shots fired. If any inputs are out of bounds, it sets an error message that the caller is expected to check. And, indeed, the caller does check the error message, but the check is all wrong — looking for the empty string instead of null, and going ahead with the computation if the error isn't the empty string (which it never will be.)

Eight blue teams answered this question in Engagement 5. Three of them believed there was no vulnerability. One team found the intended vulnerability. Three found an unintended vulnerability in degree reduction, and one found an unintended vulnerability by using an unexpected format for the input (see section 6.9.3).

A benign version correctly rejected negative initial height values. However, this code appeared in challenges with other vulnerabilities, so there were no blue team questions specifically targeting this code.

6.9.1.4 Garbage-Collection-Dependent Secant Method (AC Memory)

This version of BattleBoats uses the secant method [25], rather than Newton's method, to compute where the shot lands. This method may fail to converge if a root of the equation $-4.9 * t^2 + v_{y0} * t + h_0$ lies between the two initial values of 0 and 10^{-100}, where h_0 is the initial shot height, and v_{y0} is the initial vertical velocity of the shot.

However, there is a non-local guard on the number of iterations, which only allows 400 iterations. Furthermore, the amount of long-term memory used by this method is not terribly large – it merely maintains a `Map<BigDecimal, BigDecimal>`, adding at most one per iteration. Yet, the garbage collector usually fails to keep up with all of the other `BigDecimals` used in the computation, causing the memory threshold to nonetheless be exceeded roughly 80% of the time on such inputs.

All seven blue teams answered this question in Engagement 6. Four of them missed that there was any memory vulnerability. Two found the intended vulnerability and exploit, while one exploited the intended vulnerability with several slightly malicious inputs instead of one very malicious one.

We did not implement any specific mitigation for this vulnerability, but it is not present when Newton's method is used.

6.9.2 Cannon Location Bait in BattleBoats

In addition to the intended vulnerabilities in BattleBoats, we also implemented code that was intended to look like a vulnerability, but wasn't.

The `BOATS_PLACED` message contains a message whose size may appear to reveal the location of the other player's cannon.

`Square.java`, which represents a grid square, contains an encoding method that provides a number that can be uniquely mapped to the square's coordinates. Using that method, a string, `placementMessage`, is created whose length reveals this number for the square containing the other player's cannon. However, the string that is actually included in the `BOATS_PLACED` message is the number of digits in the length of the `placementMessage`. So, if the length of `placementMessage` is 57 (which would reveal the cannon location), the string included in the message is "2", for the 2 digits it has. Since the resulting message can only be "2" or "3", this reveals very little information about the cannon location (and the size of this message reveals nothing at all).

In Engagement 5, four blue teams answered this question, with three correctly recognizing that there was no side channel, while one succumbed to our deception.

6.9.3 Unintended Vulnerabilities in BattleBoats

6.9.3.1 Degree Reduction of Large Angles (AC-T)

In Engagement 5, BattleBoats inefficiently handled shot angles of more than 360˚, reducing them by 360˚ repeatedly instead of reducing mod 360. Three blue teams found this vulnerability.

6.9.3.2 Unintended Input Format (AC Memory and AC-T)

In Engagement 5, two blue teams discovered (on two different questions) that they could bypass the input budget by shooting with an initial velocity of the form 1e−500000 (an input which would greatly exceed the input budget if written out in standard decimal format.)

There were no unintended vulnerabilities found in BattleBoats in Engagement 6, demonstrating the benefit of spreading challenge variants over multiple engagements.

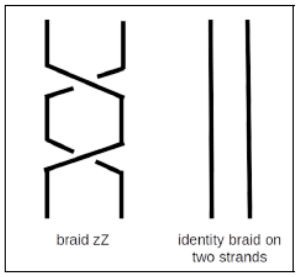
6.10 BraidIt

BraidIt is a peer-to-peer two-player game that tests the players' ability to recognize topologically equivalent braids. The development of BraidIt was motivated by the braid equivalence algorithm [26], which provided an excellent basis for some interesting and complex vulnerabilities. It was developed as a game due to the limited interest of topological braid theory to the typical user, and because it provided a context in which braids could be secret.

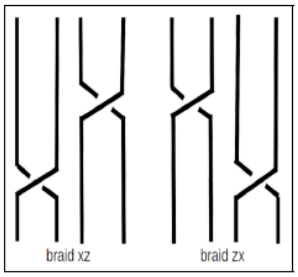
A braid is defined as a sequence of crossings on some fixed set of strands. In BraidIt, braids are represented as a word on letters [*a* − *z*] and [*A* − *Z*]. In the case of three strands, we use letters *z*, *y*, *Z*, and *Y*, where y signifies the first strand crossing over the second, *Y* signifies the strand in the first position crossing under the strand in the second position, *z* signifies the strand in the second position crossing over the strand in the third position, and *Z* signifies the strand in the second position crossing under the strand in the third.

The length of a braid is the number of crossings it contains. Two braids are topologically equivalent if one can be continuously transformed into the other.

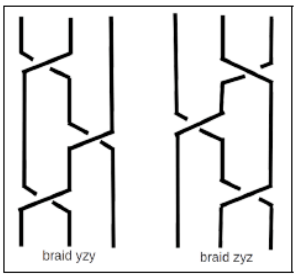
Figure 6-3 shows some examples of equivalent braids.



(a) Consecutive inverse operations are equivalent to the identity braid: $zZ = \varepsilon$.



(b) Crossings can be swapped if sufficiently separated: $xz = zx$.



(c) Example of a more complex braid equivalence: $yzy = zyz$.

Figure 5: Examples of equivalent braids.

BraidIt appeared in Engagements 5 and 6, with some significant changes between the two.

6.10.1 BraidIt in Engagement 5

In Engagement 5, game play was as follows:

- Player 1 invites Player 2 to a game on n strands
- Player 2 accepts

Then the players take turns going first in each of three rounds, comprising the following:

- Player 1 sends Player 2 five braid lengths
- Player 2's client generates five random braids with those lengths, and Player 2 secretly chooses one of them, performs modifications to its representation, and sends the resulting representation to Player 1, along with the five randomly generated braids.
- Player 1 guesses which of the five braids Player 2 selected (and modified). The application then tells him whether their guess was correct. If so, Player 1 wins the round; if not Player 2 wins the round.

6.10.2 Intended Vulnerabilities and Mitigations in BraidIt Engagement

6.10.2.1 Missing A-Handles (AC-T/AC-Disk)

Definition. Inverse: The *inverse* of a letter is the same letter in the opposite case. E.g., *x* and *X* are inverses.

Definition. Handle: A *handle* in a word is a sub-word that starts with one letter and ends with its inverse. An *x*-handle is a handle starting with the letter *x*.

Dehornoy's algorithm for reducing a braid word (which allows us to determine whether two braids are equivalent) works by recursively eliminating the leftmost permissible (as defined in [26]) handle in the word. Reduction is complete when there are no *x*-handles, where *x* is the earliest letter of the alphabet that appears in the reduced form of the braid. This version of the algorithm has been modified to ignore (rather than eliminate) *a*-handles of length more than `(MIN_LENGTH – MAX_LENGTH)/2`. Thus, a braid containing a sufficiently large permissible a-handle will never complete reduction, and the algorithm will fail to terminate.

Five blue teams answered this question, with three finding the intended vulnerability, one finding an unintended vulnerability, and one incorrectly concluding that the vulnerable code was only called on the attacker's side.

This vulnerability is mitigated by checking for `length > (MIN_LENGTH – MAX_LENGTH)`, which can never happen, instead of `(MIN_LENGTH – MAX_LENGTH)/2`. This mitigation appeared in the same variant as the False Bottom vulnerability (see section 6.10.2) in Engagement 5, as well as both mitigations of that vulnerability in Engagement 6.

6.10.2.2 False Bottom (AC-T/AC-Disk)

In this version, we have modified the method that determines whether braid reduction has completed, so that there are cases where it incorrectly returns false, and the reduction process never terminates.

The *isReduced()* method is supposed to work by simply finding the first letter in the alphabet appearing in the braid, and ensuring that its inverse does not appear in the braid—in this case, it is reduced. We have set it up so that it occasionally incorrectly determines the first letter ("bottom") in the alphabet appearing in the word. On these occasions, it determines a to be the bottom. If the given braid contains an *A*, but not an *a*, it will return false, and further reductions will not change that. (In order to ensure that the method doesn't return true when it shouldn't, we also included another variable, "lowest", which will contain the actual, correct "bottom". Additionally, we have implemented BraidIt so that randomly generated braids never contain *a* or *A*, so that there are no accidental denials-of-service in normal use of the program.)

It remains to describe the circumstances under which `bottom` is incorrectly computed. `bottom` is initially set to *z* − `cost` + 1, where `cost` is a function of the original braid (prior to any reductions), or if there has been a concatenation, then by the braid concatenated—this allows the attacker to control the cost without worrying about the other player's selections. Then, we iterate through the characters in the word, each time setting `bottom` to that character if it appears before `bottom` in the alphabet. This will behave correctly as long as the correct value for `bottom` precedes `bottom`'s initial value in the alphabet. This, of course, depends on the cost. The cost is 0 for any braid that doesn't contain all characters [*a* − *z*]. Thus for braids on less than 27 strands, as well as braids on 27 strands that don't contain crossings on all of them, the algorithm behaves correctly and always terminates. For other words, we consider the number of appearances of each character in the braid, and the cost is the difference between the maximum and the minimum. Thus the greatest value that the `cost` can take is when 25 characters appear once, and another character is repeated as many times as possible. With the length of the longest allowed braid being 52, that makes for a `cost` of 27, and bottom will be *a*, and reduction will fail to terminate if the input contains *A*, but not *a*. In any other case (if both *A* and *a* appear, or if the distribution of characters is such that `cost` isn't maximal), `bottom` will be initialized to b or beyond, and the algorithm will yield the correct value for `bottom` and will behave normally and terminate.

This can be exploited by player 2 (in a game on 27 strands) as follows:

- Player 1 offers 5 randomly selected braid lengths to choose from.
- Player 2 ignores them and responds with a malicious braid word *w* of length 52, containing *A*, but not *a*, with 25 characters each appearing once, and one character taking up the remaining spaces.
- Player 1 guesses which of the 5 original braid words, *v*, Player 2 has shown him, and attempts to determine whether they are equivalent. This requires reducing $w * v{-}1$, which will fail to terminate.

Five blue teams answered this question, with two finding unintended vulnerabilities, two finding no vulnerability, and only the control team finding the intended vulnerability.

Length Mitigation
One mitigation for this vulnerability is to set the maximum braid length to 50, whereas a length of 52 is necessary for the cost function to be non-zero to trigger the vulnerability in `isReduced()`. This mitigation appeared in Engagement 6 (see section 6.10.4 for results). Unfortunately, there was a bug that rendered this mitigation incomplete.

Cost Mitigation
Another mitigation is to modify the cost function directly to always return zero. For some reason, the blue teams were not asked about AC-T or AC-Disk vulnerabilities in this variant when it appeared in Engagement 6.

6.10.2.3 Braid Selected MAX (SC-S)

In this version, an attacker can create a trigger that will cause the size of the victim's modified braid in the `MODIFIED_BRAID` messages to be determined by which of the five braids it picked—the first, second, third, fourth, or fifth. Once this is triggered, the side channel remains effective on the victim until the victim exits the BraidIt program.

The trigger is to send a braid length that is out of range (i.e. non-positive, or greater than `Braid.MAX_LENGTH`).

Once this is triggered, the static field `ERROR_STATE` is set to be true in class `GameState`. As a result, the `SelectedState.GameState` will include the index (1, 2, 3, 4, or 5) of the selected braid. (Otherwise, this value defaults to 1.) When the `MODIFIED_BRAID` message is created, it calls `SelectedState.getBraidString()`. This method returns a padded version of the braid string, with a total size of `index * Braid.MAX_LENGTH`. Thus, there is a one-to-one relationship between the size of the sent modified braid and the index of the original braid that was chosen.

Note that, although a knowledgeable user could simply use the equivalence algorithm to determine which braid their opponent selected, this does not handle the worst-case scenario where two or more of the braids provided were equivalent. In that case, the side channel is the only way to determine which one the user picked.

Six blue teams answered this question in Engagement 5. Three of them found the intended vulnerability, although two of these failed to mention the trigger. The other teams impressed us with their creativity. Two identified an unintended vulnerability where the attacker sets the number of strands to 3 (since it can control this) and sends five distinct braid lengths. With only 3 strands, the braids are unmodifiable, so the attacker can tell by the length of the selected braid which one was selected. Finally, the control team made an interesting argument using the parity of the braid length, relying on oracle queries, but it exceeded the operational budget.

Mitigation by Absence of Trigger
One mitigation for this vulnerability is simply to not provide a way for the attacker to set `ERROR_STATE` to true, in which case the `MODIFIED_BRAID` message includes braids that are all padded to the same size, regardless of which braid was selected.

The mitigated version appeared in Engagement 6. See results in section 6.10.4.

Index Shuffling Mitigation
Another mitigation is to shuffle the ordering of the braids when the braids are created, and return them to the original ordering before sending them to the opponent. The size of modified braid in the `MODIFIED_BRAID` message is then determined by the index of the braid that the victim selected, but this is only the index in their local ordering, which is independent of the order in which the braids appear to their opponent, so this information is not useful to the attacker.

6.10.2.4 Braid Selected Subtle (SC-S)

In this version, the amount of space used for the modified braid in the `MODIFIED_BRAID` message is determined by 1) which braid was selected, and 2) the current size of the modified braid. (No trigger is required in this version.) The vulnerability lies in method `BraidSelectedState.getBraidString()`. If the modified braid has size *n*, and it was braid *i* that was selected, then this method pads the braid to size $n * i$, and it is then sent to the other player. While the size of the `MODIFIED_BRAID` message sent is, on its own, insufficient to determine which braid was selected, the attacker also has access to the actual size of the modified braid—the program displays it to him without the padding. Thus, it is simple for the attacker to determine which value of *i* is such that the `sent_length = i * displayed_length`.

Six blue teams answered this question. Two found the intended vulnerability, and one believed there was no vulnerability. The other three identified vulnerabilities that rely on the oracle and exceed the operational budget.

Index Shuffling Mitigation
This vulnerability was mitigated by shuffling the ordering of the braids when the braids are created, and returning them to the original ordering before sending them to the opponent. The size of modified braid in the `MODIFIED_BRAID` message is then determined by the index of the braid that the victim selected, but this is only the index in their local ordering, which is independent of the order in which the braids appear to their opponent, so this information is not useful to the attacker.

6.10.3 Unintended Vulnerabilities in BraidIt in Engagement 5

6.10.3.1 Unintended Vulnerability in Braid Reduction

There was a bug in our implementation of Dehornoy's algorithm which caused an unintended AC-T/AC-Disk vulnerability exploitable with certain inputs. The bug related to finding the leftmost permissible handle. The correct interpretation of that is the permissible handle with the leftmost end index, while our implementation originally found the one with the leftmost start index. One blue team found this unintended vulnerability.

6.10.3.2 Unintended Vulnerability in Connection

Another intended vulnerability was unrelated to the braid algorithm, but allowed an attacker to cause excessive computation time by attempting to connect to a user who already had a connection. One blue team found this vulnerability.

6.10.4 BraidIt in Engagement 6

For Engagement 6, we made significant changes to BraidIt to help prevent the blue teams from going down the oracle-heavy route many pursued on the side channel questions in Engagement 5. This included not allowing selected braids to have lengths near the maximum and minimum lengths, and also taking the control of the braid lengths out of the attacker's hands.

For Engagement 6, game play was as follows:

- Player 1 invites Player 2 to a game on n strands
- Player 2 accepts

Then the players take turns going first in each of three rounds, comprising the following:

- Player 2's client generates five random braids, and Player 2 secretly chooses one of them, performs modifications to its representation, and sends the resulting representation to Player 1, along with the five randomly generated braids.
- Player 1 guesses which of the five braids Player 2 selected (and modified). The application then tells him whether their guess was correct. If so, Player 1 wins the round; if not Player 2 wins the round.

6.10.5 Intended Vulnerabilities and Mitigations in BraidIt in Engagement 6

Engagement 6 included two variants of BraidIt with no intended vulnerabilities, using the mitigations described above.

6.10.5.1 Shuffling Mitigation Results

The shuffling mitigation evaded many blue teams, with many believing the vulnerability to still be exploitable. Six out of seven blue teams answered this question. Two of them failed to notice the shuffle at all. Two incorrectly thought they could modify the attacker's client to eliminate the shuffle which happens on the victim's client. Finally, two appreciated that the shuffle mitigated the STAC-vulnerability.

During the collaborative portion, however, the blue teams discovered that they had enough information to reverse engineer the random numbers generated by the non-secure `java.util.Random` in order to undo the shuffle. This vulnerability is out of scope for STAC. Nonetheless, after this, we made a point of using `SecureRandom` for random number generation.

6.10.5.2 No-Trigger Mitigation Results

All seven of the blue teams correctly identified that the no-trigger mitigation eliminated the braid selection SC.

6.10.5.3 Length Mitigation Results

Six blue teams answered the AC-Disk vulnerability question, which was impacted by an unintended vulnerability. One team incorrectly believed there to be no such vulnerability. Two thought it was vulnerable but gave answers that didn't clearly identify a vulnerability, while another provided an exploit that filled the attacker's disk. Finally, two correctly provided exploits for an unintended AC-Disk vulnerability.

6.10.6 Unintended Vulnerabilities in BraidIt in Engagement 6

6.10.6.1 Length Mitigation Failure

Due to a bug in our implementation of the cost method—failure to normalize to lower or upper case—our length mitigation is not fully effective against the false bottom vulnerability. The control team exploited this.

6.10.6.2 Unexpected Characters

Another team took advantage of a bug in our guard for ensuring braids are on allowed characters, [$a - z$] and [$A - Z$]. With these unexpected characters, they were able to defeat the length mitigation by providing a braid with non-zero cost, which exploited the false bottom vulnerability.

6.11 SimpleVote

SimpleVote is a web-based electronic voting program that appeared in Engagements 5 and 6.

Users must log in before they can access any SimpleVote functionality. Additionally, in order to participate in an election or view a previously finalized ballot, an authorized registration key for that election must be entered.

Users may update their ballot at any time up until the end of the election. They may save their progress and return at a later time to finish their ballot. Once they are satisfied with their votes, they finalize their ballots and the ballot is added to the collection of election results. However, if a user does not finalize their ballot by the end of the election, it is not included in the election results.

Users will be able to see all elections but may only act on ones in which they are eligible to participate. Each voter has a collection of traits, e.g. age, party, and district. An election author chooses which set of traits are necessary to participate in an election. Voters that match the necessary traits are each assigned a unique registration key by the election author. The registration key safeguards the voter's identity and are delivered to the voter prior to the start of the election.

SimpleVote supports the use of multiple servers for an election (e.g., for different precincts). Different voters are assigned to different servers. Servers share their tallies periodically, as well as when requested by an administrator. When an election is over, the results are available to all who were eligible to vote, but while it is in progress, only an administrator can see current results.

6.11.1 Intended Vulnerabilities in SimpleVote

6.11.1.1 Recursive Answer (AC Memory)

Ballots are represented with Question and Answer objects. An Answer instance contains a reference to a Question instance, which is usually the question associated with the given answer. However, it is

possible to have an answer store itself as its question because Answer extends Question. If an answer stores itself and is serialized, it will infinitely recurse when trying to store the answer's question.

Attack steps:

1) Create an answer to a question by posting to a ballot page, `ballots/election_id`, e.g. [("678", "337")], where "678" is a question ID and "337" is a choice ID.
2) Note the ID of the answer created by inspecting the HTML on the ballot page. Each multiple choice question lists the answer ID as the unordered list attribute—for example, "AAAAAA==@678"
3) Post the same information to the ballot page, but substitute the previous answer ID for the question ID—e.g. [("AAAAAA==@678","337")]

There is a "bug" that causes SimpleVote to search for an answer with the given question ID if it can't find a matching question. Once it finds a matching question, it looks to see if there is already an answer associated with this question. Even though the answer found already stores a question instance, it updates itself to store the new question. It is then serialized, resulting in an infinite recursion, and creating a large array on each call that is not garbage collected until the method exits.

Of the three teams that answered this question, two found an unintended vulnerability, while one did not find any memory vulnerability.

This vulnerability was mitigated by checking whether the question associated with an answer is actually an instance of answer, and, if so, storing instead that answer's question. However, given that none of the blue teams noticed the vulnerability in Engagement 5, we did not follow through with our plans to ask them about the benign version in Engagement 6.

6.11.1.2 Attacker-Controlled Compression in Aggregation (SC-S)

This is a side channel in space in the communications between two SimpleVote servers that reveals what percentage of votes a particular candidate has received in their best precinct, thanks to an attacker being able to control both the threshold for including certain data in the communication and how messages are compressed.

This side channel must be triggered by a malicious user who wishes to determine how a particular candidate is faring in their current best precinct, i.e., merely looking at SimpleVote packets (between precinct servers) in the initial SimpleVote state will not reveal the presence of the side channel. A malicious user who is interested in candidate "Joe Smith" can activate the side channel by saving a ballot with a text box answer of "Joe Smith ..." with two spaces in front of the candidate's name, and n spaces after the candidate's name, in order to learn if Mr. Smith has received at least n percent of votes in any precincts.

This ballot does not have to be finalized, just updated, so a malicious user can do this repeatedly for the duration of any election (in which it can vote). By design, every election contains at least one text box answer (non-multiple-choice) field.

When SimpleVote servers share their election results, a portion of the message only includes those candidates that received a certain percentage of votes. An unusual form of LZW compression is used on these messages. Rather than using a trie built from the text in the message to be compressed, the trie holds all single characters, plus the name of the (trimmed) last text box answer submitted (the "seed"). So, the "compressed" message will be smaller when the message contains the name of the specified candidate. Note that it is unlikely that benign users will send answers that start with multiple spaces.

Note: we must assume there is only one malicious user taking advantage of this side channel during the duration of an attack.

Communication between SimpleVote servers occurs on a port that is used exclusively for that purpose, so election results packets are easily identifiable. The election results messages are padded so that the only variation in size is due to the seed used in the compression algorithm. Thus, anytime a smaller-than-normal packet is seen, a user can conclude that the last-specified candidate has received at least the last-specified percentage of votes in some precinct.

Basic Attack Steps:

1. The malicious user sets up the side channel by specifying the candidate it wishes to learn about and the desired threshold.
2. The malicious user saves their ballot (changing something in the ballot so it is updated) ten times in order to trigger aggregation of votes between precincts. Alternatively, it could wait for hourly automatic aggregation.
3. The malicious user watches for packets coming into the server on its inter-server communication port.

By repeating these attack steps, an attacker can perform a binary search to determine the highest percentage of votes that the candidate has received in any precinct. I.e., first it determines if the candidate received at least 50% of votes in some precinct; if so, it proceeds to determine if the candidate received at least 75%; else it proceeds to determine if the candidate received at least 25%, and so on. Due to the ten operations needed to trigger aggregation, once it narrows down the vote percentage window sufficiently, it is better off using oracle queries.

Of the four blue teams that answered this question, two detected the vulnerability, and two did not. Those who did not, looked carefully at what goes into the aggregation messages, but missed that the attacker was able to affect it, demonstrating a struggle with non-locality of pieces of a vulnerability.

Postponed Seed Mitigation
One mitigation of this vulnerability is to change how the attacker's candidate "seed" is used in compression. In this version, instead of only adding the seed to the trie, first the entire summary string is processed for substrings to add to the trie. Because there is a limit of 2048 on the number of codewords to be added to the trie, by the time it gets to the attacker's seed, this may have been reached, and the seed has no impact in the worst case. Of the 6 blue teams that answered this question, all but one had the correct answer, while one believed they could still successfully seed the trie.

Percentage Magnitude Mitigation
Another mitigation compares the fraction of votes received to the seeded percentage. However, the fractions of votes received is a number between 0 and 1, while the seeded percentage is a number between 0 and 100, so this only allows the attacker to determine whether or not their seeded candidate received 100% of the votes in their best precinct. In the vulnerable version, the fraction is multiplied by 100 before making this comparison. All but one blue team correctly answered this question, with most of them specifically noting this issue, while one believed the vulnerability to still be present.

6.11.1.3 Registration Key Large N (AC-T)

Verification of registration keys is an expensive operation, due to (unnecessarily) verifying the format of the key before verifying that it is authorized in the given election. The format of the key is a combination of a representation of an integer n, and a logistic map function of *n*. The logistic map computation is

expensive, but values are cached for a subset of values, which makes it reasonably fast in practice. In this vulnerability, it is possible for an attacker to cause n to be significantly greater than any value in the cache. Although there is a guard, `MAX_N` on the largest value of *n* that will be processed, if a user submits a key with *n* = `MAX_N`, then `MAX_N` is raised by *7*. Doing this three times, followed by submission of a key with a higher value of *n* is sufficient to cause excessive computation time.

Of the six teams that answered this question in Engagement 5, three detected the vulnerability, and three did not.

We also created a benign version, where there was no way for the attacker to increase `MAX_N`, but blue teams were not asked about this variant.

6.11.1.4 Registration Key Cache Miss (AC-T)

Recall that registration keys are made of two components, an integer n value, and a logistic map function of *n*. As above, registration key verification is expensive but values are cached to prevent excessive computation time. In this version, however, an attacker can cause items to be deleted from the cache. Specifically, if an attacker enters a registration key whose *n* component equals `MAX_N` (which is originally set to 995), `MAX_N` will be increased by 1. Then, an attacker can again enter a registration key with *n* equal to the original `MAX_N`, which will now be allowed through the guard and will proceed to the logistic map computation step. Normally, additional values are not cached after the initial cache initialization. However, when a value is computed that is further than an expected gap from the closest cached value below it, caching is turned on. Since this value is beyond the expected values, this gap is exceeded, so this item is cached, and it occupies the last position in the cache, which would otherwise be left empty. The cache will then deem itself "full" because of this and will clear out several of the highest cache entries. If one then enters a registration key that would normally rely on one of these missing cache entries, the verification of this key will take longer than usual, and with a sufficiently large key, will exceed the time budget.

In Engagement 6, all seven blue teams answered this question. Four answered correctly, two thought there was no vulnerability, and one exceeded the input budget, focusing on significantly increasing the value of *n*, without realizing that they could cause the cache deletion with just a few requests.

Our mitigation for this vulnerability was to leave the ability to increase `MAX_N` by 1, but remove the cache deletion. While this left a vulnerability, we believed that it could not exceed the resource limit within the attack input budget.

6.11.1.5 Registration Key Leak (Intended SC-T, but actually benign)

Recall that each voter is given a registration key for each election in which they're eligible to participate. When a voter wishes to participate in an election, it must enter this registration key, which is checked for general validity and for being authorized for the election in question. Ballots are stored with this registration key as the only identification of the voter. The association between a voter's IP address and registration keys is secret.

Recall also that keys are created by combining two components, an integer, *n*, and a logistic map function of *n*. Our intent was that keys be uniquely identified by the value of *n*.

This variant contains a side channel vulnerability where the time of the verification step is a reversible function of *n*, enabling an attacker to determine the registration key submitted by a voter.

The verification proceeds by checking each possible value of *n* until finding one that matches the rest of the key. The check at each step takes sufficient time to separate the timings of different keys. Although logistic map values are cached, the cache is a linked list, which sufficiently slows down the lookup time.

The attacker can recognize registration key packets by their size, and then determine what key was used by the timing of the response.

In Engagement 5, unfortunately, we discovered that there was a bug that caused registration keys to not be uniquely determined by *n*, thus invalidating the side channel. Four blue teams answered this question, which asked whether blue teams could determine the contents of another user's ballot via a space side channel. One team incorrectly thought they had found an entirely different side channel, but they were mistaken both about the information that it revealed and the amount of entropy in the secret of ballot content; they failed to consider the presence of textboxes, which allowed free-form responses in ballots, making ballot contents impossible to determine with the oracle queries they thought they could use. One team found the intended vulnerability but missed (as we did) the fact that *n* didn't uniquely determine the key. Another team didn't detect any side channel at all. Finally, one team identified the intended vulnerability but correctly determined that the non-unique mapping mitigated it.

In Engagement 6, we corrected the bug that rendered this variant non-vulnerable, and instead offered two different mitigations.

Non-Unique n Mitigation
One benign version of this maintained the side channel that reveals *n*; however, in this version the registration key is not uniquely determined by *n*; instead a valid registration key can have any *n* less than or equal to *k*, where *k* is the integer that maps to the logistic mapping portion of the key.

In Engagement 6, six blue teams correctly determined that there was no exploitable side channel here, while one believed that it was still exploitable. However, the teams failed to notice that registration keys were not uniquely determined by *n*. Unfortunately, one of our engineers added a check to this version that prevented users from using registration keys not assigned to them, and the final consensus answer was that the registration key could be determined (incorrect), but that it wasn't exploitable due to this check (correct).

Eliminated n-Dependence Mitigation
Another variant eliminates the side channel by only checking the value of n specified in the registration key, rather than checking all values up to that n, thus the dependence of the timing on n is significantly reduced. Due to the logistic mapping computation for n alone, some keys have distinguishable verification times; however, there are sufficient keys with overlapping times to eliminate the vulnerability.

In Engagement 6, all five blue teams that answered this question answered it correctly, while two teams declined to answer.

6.11.1.6 CRIME Password Leak (SC-S)

This vulnerability allows the attacker to learn a user's password one character at a time. It is similar to the CRIME [27] and BREACH [28] vulnerabilities, where user input is combined with secret data and compressed, yielding a size that is indicative of how close the user input is to the secret.

Specifically, when a user-submitted password is checked, a token is created comprising

```
token = Z(s + true_password + s + submitted_password)
```

where *s* is a string of space characters, the length of which is random, but at least 9. Although random, for a given password (not the guess), the length is constant. Thus, repeated guesses for the same password have consistent behavior. Z denotes compression with the DEFLATE compression algorithm [29]. The closer the submitted password is to the true one, the more effective this compression will be, i.e., the smaller the resulting token will be.

If the password submitted is not correct, the length of this token is used as the length of the session ID included in the redirect URL. Thus, an attacker can determine the password character-by-character, by determining which character at the current position yields the shortest session ID it can obtain.

The question asked whether there was a side channel in space via which an attacker could obtain a voter's profile. Of the five blue teams that answered this question, two correctly identified the vulnerability and two believed there was none. The fifth team saw the vulnerable code, but was over-focused on the profile and allowed themselves to be misled by the additional information in the question that the target user would view their profile during the session. They decided that they only needed the password length to determine the profile contents, but determined that the length of the password was not directly correlated with the information leaked by the token, so they decided that the side channel was not strong enough.

6.11.2 Unintended Vulnerabilities in SimpleVote

6.11.2.1 Benign Memory Use

In Engagement 5, two blue teams discovered that the aggregation of votes between servers actually exceeded the memory bound.

6.11.2.2 Breaking the Budget

In Engagement 6, with the attacker able to increase `MAX_N` only by 1, it would take the submission of 8 registration keys to raise `MAX_N` high enough and successfully submit a key with large enough *n* to exceed the time bound. In addition to 462 bytes to log in, we believed each key submission to require 510 bytes, which included a Transport Layer Security (TLS) handshake, so with an input limit to 4000 bytes, we believed there was no AC-T vulnerability present. However, one blue team believed during the take-home portion that they could reduce the POST header sizes to achieve the exploit within the budget. At the live engagement, the other teams needed convincing, and they were eventually able to exploit the vulnerability within the given budget, not by reducing the header sizes, but by maintaining a TLS session to avoid repeated TLS handshakes, thus significantly reducing the number of input bytes needed.

6.12 Calculator

Calculator, which appeared in both Engagement 6 and 7, is a suite of simple web-based calculators that can be used to evaluate various integer arithmetic expressions.

6.12.1 Intended Vulnerabilities and Mitigations in Calculator

6.12.1.1 FFT Padding (AC Memory)

In Calculator, multiplication of numbers that have more than 1001 digits is done using FFTs. Our implementation of FFTs is polymorphic, where each variant is configured, using Jinja, to split the data into a different number (`numOfDivides`) of arrays in its divide-and-conquer recursion.

This vulnerability lies in a version of our FFT implementation where there is an error in the logic that determines how much padding is needed when splitting the arrays for recursion. Our class that implements FFT invokes the `helper(data,n,inverse)` method that computes how much padding to add where data is the complex array to be transformed, *n* is the length of the data to be transformed, and `inverse` is a boolean value that is `true` if we want the inverse FFT. In certain cases (namely, in the forward FFT, when the average `data` value is more than 4.5) it returns a padding of size

```
2 * numOfDivides
```

when it should return zero, causing the `data` array to be unnecessarily padded, and leading to extra iterations of the algorithm. Since all padded data is added to an accumulating array, if unnecessary padding occurs many times, it can cause excessive memory usage and exceed the resource bound.

In Engagement 7, Utah found the intended vulnerability, while both the collaborative teams and the control team found unintended vulnerabilities.

Web Server Mitigation
In this mitigation, the vulnerability is prevented by configuring the web server so that a POST large enough to exploit the vulnerability is not processed.

In Engagement 6, four teams found this version of Calculator benign and three teams found unintended vulnerabilities.

Correct FFT Mitigation
In this mitigation, padding is implemented correctly in FFT, and therefore does not use excess memory.

In Engagement 6, four teams found this version of Calculator benign and three teams found unintended vulnerabilities.

In Engagement 7, the control team and Utah found this version of Calculator benign. The collaborative teams reported two potential unintended exploits, one of which failed to achieve the required probability of success.

6.12.1.2 N^{th} Root (AC-T)

Our implementation of nth roots uses Newton's Method [24], but only takes integers, making the Category 13 canonical vulnerability [3] inapplicable. The n^{th} root vulnerability actually lies in our guards for arguments provided in an n^{th} root expression. We provide a guard on user input preventing even roots of negative numbers, but the check for even parity has an error, which allows the application to attempt to compute even n^{th} roots of negative numbers, where *n* ends with the digit 8. Though negative numbers are not accepted as input, users are able to obtain negative numbers within the root through nested calculations. Using these so-obtained negative numbers and the improper guard makes it possible for the calculation of an n^{th} root to enter an infinite loop, exceeding the resource bound.

In Engagement 7, the control team found the intended vulnerability, the collaborative teams found two unintended vulnerabilities, and Utah incorrectly found this version of Calculator benign.

We also provided a benign version where input validation correctly protects against even roots of negative numbers.

In Engagement 7, all blue teams correctly found this version of Calculator benign.

6.12.1.3 Inefficient Parsing (AC-T)

This vulnerability is the result of inefficient expression parsing. When parsing an expression, after each operator is encountered, there is a search through the rest of the expression for an integer, which can lead to $O(n^2)$ parsing time in the worst case. Additionally, the search for an integer is slow as it checks each character in the expression against an excessively long string of digits. Instead of using the necessary 10 digits, 0 through 9, it uses all digits 0 through 100. Using this, an attacker could exceed the resource bound by submitting a long string of operators.

In Engagement 6, one team did not detect the intended AC-T vulnerability, five teams found unintended vulnerabilities, and one team found both this intended vulnerability and an unintended vulnerability.

Web Server Mitigation
In this mitigation, the vulnerability is prevented by configuring the web server so that POSTs large enough to exploit the vulnerability are not processed.

In Engagement 6, one team found this version of Calculator benign and six teams found unintended vulnerabilities.

Short Digits Mitigation
In this mitigation, the vulnerability is prevented by using a short string of digits when searching through the rest of the expression for an integer. Instead of using digits 0 through 100, it uses only the necessary digits, 0 through 9.

In Engagement 7, all blue teams correctly found this version of Calculator benign.

Efficient Parsing Mitigation
In this mitigation, regular expression pattern matchers are used to determine the location of operators within the expression and extract integer chunks from the expression. This results in expression parsing that terminates quickly in the worst case, avoiding excessive time use.

In Engagement 7, all blue teams correctly found this version of Calculator benign.

6.12.1.4 Web Server POST Expansion (AC-Disk)

The web server used in Calculator handles large POSTs by writing their contents to disk. This, as well as two bugs within the web server make up the Web Server POST Expansion vulnerability. The first bug is in the computation of the length of a received POST. This length is incorrectly parsed from a string, causing it to be rounded down to the greatest multiple of 1000 less than its actual length. This makes it possible for a POST whose length is greater than `MAX_POST_LENGTH`, but only in the last three digits, to be allowed through. The second bug is in the retrieval of the POST content. It combines a genuine input stream of the POST content with a stream generated by a random number generator. This random generator uses a fixed seed, allowing an attacker to predict its output. After a byte is read from the actual POST, it is compared to the next random value, v. If it is the same, then the next `MAX_POST_LENGTH` reads will return v. The more overlap there is, the more the POST content will be expanded. By sending the precise sequence of characters produced by the stream, an attacker can successfully construct a POST that will cause Calculator to exceed the resource bound.

In Engagement 7, all blue teams incorrectly found this version of Calculator benign, due to only looking at the disk usage of log writes.

Accurate Length Parsing
Mitigation In this mitigation, the length of a received POST is parsed correctly. This allows the web

server to reject any POST whose length is greater than MAX_POST_LENGTH, preventing an attacker from submitting a POST large enough to exceed the resource bound, regardless of the presence of the POST expansion bug.

In Engagement 7, all blue teams correctly found this version of Calculator benign.

Correct POST Retrieval
In this mitigation, the genuine input stream of the POST content is not compared to the randomly generated bytes for expansion. Instead the genuine stream is accepted and copied into an input stream array 100 times. Whenever the `read` method is called, the next byte is read from the index

```
i = lastRead (mod 100)
```

of the input stream array. This makes it look like the same byte is read 100 times before moving to the next byte. However, after bytes are read from the i^{th} stream they are marked as read, so when a byte has been read from one stream, it's already marked as read from the others. As a result, the input stream is not actually expanded at all, and an attacker's disk usage is limited to the size of the actual POST.

In Engagement 6, all seven teams identified this version of Calculator as benign.

6.12.2 Unintended Vulnerabilities in Calculator in Engagement 6

6.12.2.1 Unintended Vulnerability in Multiplication (AC-T)

An AC-T was discovered in Calculator which takes advantage of the processing of multiplication. When handling multiplication, if one of the numbers has less than 1001 digits then the method `simpleLogSpaceMultiply` is used to calculate the product.

Recall that our build system makes certain pre-specified modifications to our source code to prevent vulnerabilities from being obvious by diffing variants of the same challenge program. Unfortunately, several nested non-deterministic loops (as shown in snippet 7) were added to the `simpleLogSpaceMultiply` method. Blue teams observed that, although each of these add a very small amount of computation time on its own, its effect can be amplified by entering an expression containing a large quantity of multiplications between numbers with less than 1001 digits.

6.12.2.2 Unintended Vulnerabilities in FFT

When handling multiplication, if both numbers have more than 1001 digits then FFT is used to transform both numbers before calculating the product, and the product itself. This process takes place in the `primaryTransform` method where the numbers are recursively split into `numOfDivides + 1` sub-arrays and passed as an argument to a recursive call of `primaryTransform` until each array has a length less than or equal to `numOfDivides`. Once this condition has been met, each array is passed to the method `quickTransform` to finish the FFT process.

Blue teams found that an AC memory vulnerability existed in the primaryTransform step of our FFT implementation. Since each number is recursively split numOfDivides + 1 times using primaryTransform, there will be numOfDivides + 1 more invocations of the method primaryTransform upon every invocation of itself. This allows the attacker to exceed the resource bound by exploiting the array duplications and recursive invocations by entering multiple small expressions of the form

$$A^n \times A^n \times A^n$$

in parallel, each expression resulting in a number with more than 1001 digits.

Blue teams also found that an AC-T vulnerability existed in the `quickTransform` step of our FFT implementation. Unfortunately, due to our random code transformations, several nested non-deterministic loops (as shown in snippet 7) were added to the `quickTransform` method. Blue teams observed that, although each of these adds a very small amount of computation time on its own, its effect can be amplified by entering an expression of the form

$$A^{n1} \times \ldots \times A^{nm}$$

that results in a number with more than 1001 digits.

6.12.2.3 Unintended Vulnerability in Processing of Parentheses (AC-T)

When processing expressions Calculator uses the method `processOutcome`, which uses an operand stack to store temporary computation results. However, if the expression contains a sequence of parentheses-enclosed expressions without operators between them, `processOutcome` will evaluate each parentheses-enclosed expression, but only return the result of the last expression evaluated in the sequence. This allows an attacker to input a lot of computation without causing the final result to become large enough to be rejected by the application. An attacker could then exceed the resource bound by crafting an input of the form *A...A* where

$$A = (B^n \times B^n \times B^n \times B^n) \times (B^n \times B^n \times B^n \times B^n \times B^n)$$

6.12.2.4 Unintended Vulnerability in Processing of Expressions (AC Memory)

After an expression is evaluated, the `processExpression` method returns a string containing the final result. This string is generated using the `LargeInteger` class's `toString` method which blue teams determined to have high memory usage. Therefore, an expression with a large result can result in large memory usage. The input budget of 10KB prevents an attacker from causing more than approximately 1.2GB of memory usage with a single query. However, the attacker is still able to exploit this vulnerability by sending numerous smaller requests in parallel.

6.12.2.5 Unintended Vulnerability in Division (AC Memory)

An AC memory vulnerability was discovered in Calculator that takes advantage of the processing of division. When handling the division of two numbers, the divide method allocates many new arrays in each iteration of the main loop for storing coefficients derived from the dividend. Since the dividend is part of the user input, the attacker is able to control the size of the allocated arrays. This allows the attacker to exceed the memory resource bound.

6.12.3 Unintended Vulnerabilities in Calculator in Engagement 7

6.12.3.1 Unintended Vulnerabilities in Subtraction

An AC memory vulnerability was discovered in Calculator that takes advantage of division and subtraction in a similar way to the unintended vulnerability in division found in Engagement 6. The `divide` and `subtract` methods (the `divide` method calls the `subtract` method) both allocate many new arrays in each iteration of their main loops for storing coefficients derived from operands, leading to inefficient memory use. Since the attacker has complete control over the mathematical calculation that the calculator performs, it is able to control the size of the allocated arrays. This allows the attacker to exceed the memory resource bound.

An AC-T was also discovered in Calculator. Unfortunately, due to our random code transformations, several nested non-deterministic loops (as shown in snippet 7) were added to the `subtract` method.

Blue teams observed that, although each of these adds a very small amount of computation time on its own, their effect can be amplified by entering expressions that contain many subtractions and/or divisions (the `divide` method calls the `subtract` method) between large numbers.

6.12.3.2 Unintended Vulnerabilities in Expression Parsing

Calculator parses user input using the method `processExpression`. Before parsing begins, the expression is checked for valid parentheses use using the method `checkParentheses`. One of the collaborative teams discovered that an AC-T vulnerability existed in this method through dynamic analysis. They found that the time resource bound could be exceeded by entering an expression with a large number of highly nested parentheses.

After checking for valid parentheses use, `processExpression` begins to parse the expression. During parsing, when an operator is encountered, the method looks ahead in the input for an integer using the `getNextIntChunk` method. The control team discovered that an AC memory vulnerability existed in this method due to a large amount of memory being allocated every time it is called. They found that the memory resource bound could be exceeded by entering an expression that begins with many operators and contains a large numerical value.

6.13 CyberWallet

CyberWallet is a web-based banking application. Clients can log in to access their bank accounts. A specific bank owns the bank accounts that the application is accessing, defined by its unique routing number. Clients can view their accounts' transaction histories, their balances, and the co-owners (if any) of the accounts they own. The client can also transfer money between accounts they own or from their account to another account with the bank. There are three types of accounts they can own: checking, savings, and a certificate of deposit. Each account is assigned a tier (Bronze, Silver, Gold, or Platinum) that is determined by the amount of money in it. Different tiers correspond to different ranges in account balances, and they can affect properties such as interest on the account.

6.13.1 Intended Vulnerabilities and Mitigations in CyberWallet

6.13.1.1 Biased Ads (SC-S)

When clients log in, they can browse through the accounts they own, view information specific to an account they own, or make transfers from any of their accounts. Every page in this web-based application displays an ad. The vulnerability lies in the ads. On pages that contain account-specific information (e.g., balance, transactions, etc.), the size of the ad that is displayed is within a specific size range that correlates to the tier of the account. Recall that an account tier maps to a fixed balance range. Below in table 6-1 is the tier mapping.

Table 1: Mapping of account tiers to fixed balance ranges.

Tier	Balance Range
Bronze	[$0, $10,000)
Silver	[$10,000, $100,000)
Gold	[$100,000, $1,000,000)
Platinum	[$1,000,000, ∞)

An attacker can therefore monitor a target client's network traffic and deduce the account tier of their account based on the size of packets observed. The pages that contain no account-specific information display ads with sizes that do not correlate to any account tier. The ad names all start with the word "ad" and are followed by a two-digit number. The ads are chosen on the server using an algorithm that uses the account balance and maps it to the second digit of the ad name. Since there are four account tiers, ads that share the same second digit, that is within 0 − 3, are vulnerable and map to a respective account tier (e.g., *ad01*, *ad11*, *ad21*, etc. would map to Tier 1 [which is labeled as tier Silver]). In the algorithm to choose ad names, a random amount (drawn from a `SecureRandom` Gaussian distribution) is added to the account balance. This random amount can only change the account tier by bumping it up if it is greater than or equal to 1 and the account balance is near the boundary of the higher tier range. This is the worst-case scenario, and the probability that an ad name selected corresponds to the wrong tier is 40%. The random amount was added to test the blue teams' ability to determine how much randomness it takes to mitigate a side channel.

In Engagement 7, all of the teams answered this question and correctly identified the vulnerability.

There was also a benign version in which there is a significantly reduced correlation between the ad size and the account tier. The account balance is used in the algorithm to generate the ad name, however, it is multiplied by a random amount, thus reducing the correlation.

In Engagement 7, all of the teams correctly identified that there was no vulnerability in the benign version.

6.13.1.2 Account Number (SC-T)

When a client makes a transfer, the details of the transfer (routing number, account number, transfer amount, and description) are checked on the server. The validation of each transfer detail, except the destination account number, is fast. The server stores all account numbers within a trie data structure where each node corresponds to a digit. When the account number trie is searched for an account number, the trie is traversed for each digit starting from the leftmost digit. For example, if account numbers 12345 and 13234 are searched for in a trie consisting of 12346, 12434, 45245, and 80843, etc., as shown in Figure 6-4, then the search will traverse all the way to the last digit for 12345 and only to the second digit for 13234. Therefore, the search time of an account number has positive linear relationship with the number of digits traversed through the trie. An attacker can use this relationship to discover another account number that exists within the trie by trying to make a transfer. With each transfer, the attacker can time the validation of the transfer and then modify the account number digits to obtain a greater validation time. Eventually, the attacker can discover an account number that is found within the trie and passes the validation. After the attacker discovers another client's account number, it can use that account number and a random two-digit authorization code found by trial and error to reverse transfer money from that target account into their own account.

In Engagement 7, all of the blue teams correctly answered this question except for the control team. The control team seemed to have misinterpreted the question and assumed that they had to target a particular account (where the account number is known); therefore, they answered that money can be stolen from an account, but not via a timing side channel.

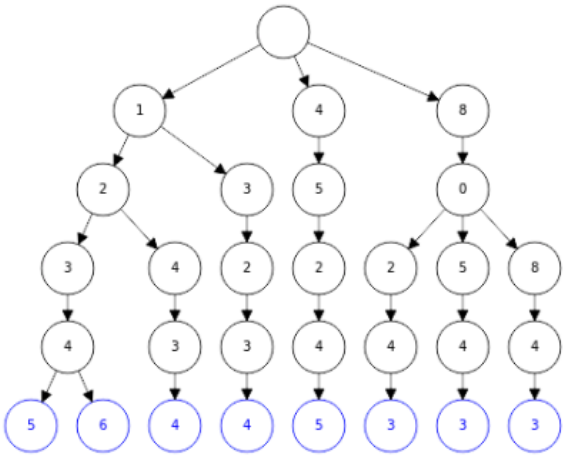


Figure 6: Example of a trie containing numbers: 12346, 12345, 12434, 13234, 45245, 80243, 80543, and 80843.

In the benign version, during the transfer details validation, the account number is searched for in a hashmap of account numbers on the server's database before searching the trie. The hashmap search time is constant and will reject all non-existent account numbers, therefore eliminating the attacker´s ability to research and discover account numbers from validation timings.

In Engagement 7, all of the blue teams correctly answered that there was no timing side channel in the benign version.

6.13.2 Unintended Vulnerabilities in CyberWallet

There were no unintended vulnerabilities in this challenge program.

6.14 InAndOut

InAndOut is an online order-placing and tracking application for a pizza parlor that appeared in Engagement 7. The user can build their pizza by selecting the toppings, and the order is queued for processing. After the order is processed, the user can pick up the pizza online.

Upon connecting with the website, the user lands at the login page, where it must provide a username and password to log in to the application.

After the user logs in, it is redirected to a page where it can build their own pizza by choosing the toppings and the number of pizzas with the chosen toppings.

Upon successful placement of the order, the user is issued a confirmation code that it can use to pick up the pizza online on a different page. The store's inventory is updated to reflect the ingredients used in the order (the inventory manager runs as a separate process).

After the pizza is processed, the user has five minutes to pick up the pizza online; if the user does not pick up the pizza within this period, it is donated to a homeless shelter.

6.14.1 Intended Vulnerabilities in InAndOut

We used LFSRs as the basis of a variety of computations used in InAndOut because they are very simple to implement, but, on the other hand, they exhibit rich and complex behavior. Applications of LFSRs

include error correcting codes, pseudorandom sequence generation, test pattern generation and signature analysis in VLSI circuits, and program counters in simple computers. Their inherent simplicity and complex behavior make them ideal candidates for developing vulnerabilities that are hard to find and analyze.

An LFSR [30] is a device whose current state is a linear function of its previous state. It consists of a register of *L* bits, and its state is the register's content. The state of the LFSR is advanced from state *n* to state $n + 1$ by first shifting the register's content to the right (hence the name shift register [31]), then using the rightmost bit in state *n* as the leftmost one in state $n + 1$ (hence the name feedback), and, finally, applying the linear function to the register's contents.

The most commonly used linear function is exclusive-or (XOR) applied to *k* pairs of bits; the set of these pairs is known as the toggle mask. Figure 6-5 is a sketch of a 16-bit LFSR in the Galois configuration [30].

The initial state of the LFSR is called the seed, and because the operation of the register is deterministic, the stream of values produced by the register is completely determined by the seed and the toggle mask. Likewise, because the register has a finite number of possible states, they must eventually repeat; therefore, this sequence is cyclic (a so-called *m*-sequence). The total number *m* of states in the sequence for an *L*-bit LFSR is $m \leq 2^L - 1$; the upper bound obtains when the taps' locations correspond to the powers > 0 of a primitive polynomial [32, 33] over GF(2). If the linear function is XOR, and if the initial state consists of all zeroes, then the state will never change (therefore $m \leq 2^L - 1$ as opposed to $m \leq 2^L$).

In InAndOut we always use a primitive polynomial as the toggle mask, and therefore $m \leq 2^L - 1$ always.

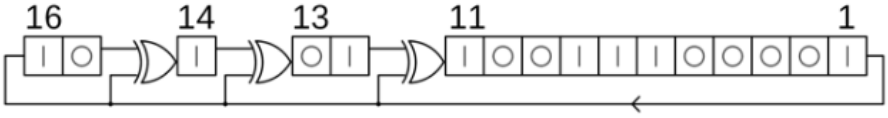


Figure 7: A 16-bit Galois LFSR. The numbers shown at the top correspond to the non-zero powers of the primitive polynomial $x^{16} + x^{14} + x^{13} + x^{11} + x^1 + 1$ and indicate the pair of bits to be XORed, so the register cycles through the maximum number of $2^{16} - 1 = 65535$ states excluding the all-zeros state. The shown state (numbers inside the squares), 1010 1100 1110 0001, will be followed by the state 1110 0010 0111 0000.

We used LFSRs to create two SCs (time and space/time) and two AC vulnerabilities in InAndOut.

6.14.1.1 Confirmation Code (SC-T)

When an order is accepted, a secret confirmation code C is generated and sent back to the user. This confirmation code will enable the user to pick up the pizza that it ordered.

The SC leaks the secret confirmation code, and the vulnerability is in the calculation of the confirmation number; timing the calculation reveals the confirmation code.

To signal the beginning of the calculation of the confirmation code, a first message is sent to the inventory manager. The sequence of messages is:

1. The server receives a message from the user with an order.
2. The server sends a first message to the inventory manager.
3. The server sends a message to the user with the calculated confirmation code.

4. The server sends a second message to the inventory manager.

By measuring the time elapsed between an inventory message and the message sent to the user with the confirmation code, an attacker can calculate the confirmation code as explained below.

The confirmation code is a string of the form $F_1–F_2–F_3–F_4–F_5–F_6$ where F_i is a numerical field. We allow for 63 different confirmation codes (see below for the reason to limit the number of confirmation codes).

To calculate a numerical field, we first get a random number, $b \in [1, 63]$. The binary expansion of b indicates whether the corresponding F is zero or a different value.

For example, if b is 7, then its binary expansion is 000111, indicating that the first three Fs are zero, and the last three Fs are non-zero. The nonzero values are calculated using an LFSR of a given length; for each field a different LFSR is used. Because the length of the LFSRs are different, the computation times are different, thus creating a timing side channel.

In more detail: the confirmation code C is a string of the form

$$C = \text{“}F1–F2–F3–F4–F5–F6\text{”},$$

where F_i is either zero or N_i, and N_i is computed with the i^{th} LFSR and has the value

$$N_i = 2^{(i-1)};$$

i.e., the possible values of N_i are

$$N_1 = 1,\ N_2 = 2,\ N_3 = 4,\ N_4 = 8,\ N_5 = 16,\ N_6 = 32.$$

Whether F_i is zero or N_i is decided randomly; this random choice is derived from the binary representation of the random number $b \in [1, 63]$:

$$b = b_1b_2b_3b_4b_5b_6.$$

When b_i is 1 then $F_i = N_i$, otherwise $F_i = 0$. Note that there are $26 - 1 = 63$ different confirmation codes (a code with all zeros is not allowed).

The calculation of each number N_i takes a time t_i ($t_i \neq t_j$ for $i \neq j$), and therefore the total time T it takes to calculate a specific confirmation code C is

$$T = \sum_{i=1}^{6} t_i \, 1(F_i),$$

where $1(F_i)$ is the indicator function for the field F_i ($1(F_i) = 1$ if F_i is not zero, and $1(F_i) = 0$ otherwise).

Thus, there exists a linear relationship between the number and the time T. The confirmation code can be reconstructed from the binary representation of N.

$$N = \sum_{i=1}^{6} F_i \,,$$

Remark:
As the server approaches and surpasses 1000 received orders, we found that response times slow down

causing a large amount of overlap in the confirmation codes that are mapped to any single response time. As a result, we decided to enact an order limit of 1000 that will cause the server to shutdown once this limit is reached.

When less than 1000 orders are received by the server, we found that up to five confirmation codes can map to a single response time. Therefore, we adjusted the number of available operations to the blue teams to allow up to four guesses for a given response time.

The collaborative teams and the control team correctly identified this vulnerability. Utah, on the other hand, believed that there was significant random noise in the confirmation number generation process.

In the benign variant, the lengths of the LFSRs used to calculate the fields of the confirmation code are such that the execution time is on the order of tens of microseconds, and, therefore, the SC is buried in noise.

6.14.1.2 Confirmation Code (SC-ST)

The secret leaked through this SC is the confirmation number for a pizza order.

The time part of the SC is similar to the time SC described above: the time difference between the first inventory message from the server to the inventory manager and the message from the server to the client with the confirmation number is a function of the confirmation number. However, in contrast to the time SC, the message from the server to the client is delayed by a random time, and this random time is encoded in the packet length of the second inventory message.

To summarize, the chain of events is the following:

1. A POST message is received by the server with the pizza order.
2. The server sends a message of size s_1 to the inventory manager to signal the beginning of the calculation of the confirmation number.
3. The confirmation code C is calculated. The calculation takes a time $T \propto C$.
4. A random time t_2 is generated.
5. A second message to the inventory manager is created, and the size of this message is $s_3 = s_1 + s_2$, where s_2 is proportional to t_2.
6. The sending of this second message is delayed by t_2, and then it is sent to the inventory manager.
7. Immediately thereafter the confirmation number is sent to the client.

Thus $N = F_i$ is proportional to $(T + t_2) - (s_3 - s_1)$, and an attacker can recover N by observing the time difference $T + t_2$ between the first and second messages to the inventory manager and the difference $(s_3 - s_1)$ in packet size. Finally, the confirmation code C can be calculated from the binary expansion of N.

None of the blue teams detected this vulnerability. The collaborative teams attempted looking at space and time sequentially, i.e., narrow down the secret with space and then narrow it down further with time. However, this side channel required *combining* space and time to observe any correlation. We are surprised that none of their tools were capable of doing this.

In the benign variant, s_2 is random, and the lengths of the LFSRs used to calculate the fields of the confirmation code are such that the execution time is on the order of tens of microseconds, and, therefore, the SC is buried in noise.

6.14.1.3 Order Priority Denial of Service (AC-T)

After an order is created by the customer, it is placed in a priority-based queue to await processing. The priority of an order is a function of the toppings requested. To a benign user, there are 15 toppings available. However, there is an extra, secret topping that results in a higher priority. An attacker that knows the existence of this topping, can include it in their orders, thus ensuring that their orders will be processed first and delaying infinitely the processing of orders by benign users.

Topping i has popularity ranking r_i ($r_i \geq 1$). An order consists of the number of pies ordered and a list of the toppings desired; in this list, the toppings are specified by popularity ranking, not by name. For example, if a customer orders a pizza with toppings rated 1, 6, and 15 in popularity, then the client sends to the server the string "1,6,15" (the quantity of pies is sent in a separate string).

The priority $\mathcal{P}$ of an order is a function of the toppings ordered as encoded in a bit array. If the topping with rank r_i is ordered, then bit r_i is set. After the bits corresponding to the ordered toppings are set, we permute the set bits using an LFSR. If B_T is the bit array with bits set according to the ordered toppings, we use B_T as the input state to an LFSR, and the resulting LFSR state, B_P is a permutation π of the set bits in B_T, i.e., $B_P = \pi(B_T)$. Finally, we use B_P as the big-endian binary representation of the priority $\mathcal{P}$.

To permute the set bits of B_T, we feed each of the positions of the set bits to an LFSR, which we run for one step, thus mapping the set bits to different set bits, except for the zeroth bit, which is always mapped back to zero (this is a characteristic of any LFSR that uses the XOR function). The output of the LFSR is used to set the corresponding bit position of B_P, and we use B_P as the binary representation of the priority.

For example, if bit A is set in B_T, feeding the LFSR with A as the initial state and running the LFSR for one step yields back a number B that is used to set bit B of B_P. The end result is a deterministic permutation of the set bits of B_T.

Finally, the priority value is the value encoded by B_P in big-endian representation.

For example, if, after the permutation of the set bits in B_T the set bits of B_P are 1, 4, and 15, then the priority $\mathcal{P}$ will be

$$\mathcal{P} = 2^{14} + 2^{11} + 2^{0},$$

There is an additional topping that an attacker can order using its own ad-hoc client: the topping with zero popularity-ranking, that isn't available to benign users (who order their pizza through a web browser). By including this topping in their order, an attacker's pizza order will have a higher priority than any order placed through the browser. The vulnerability stems from the fact that there are 15 toppings, but we use a bit array of length 16 and big-endian representation to calculate the priority, i.e., the most significant bit (MSB) is bit 0. When the MSB is not set, $\mathcal{P} \leq 215 - 1 = 32767$, whereas setting the MSB yields $\mathcal{P} > 32767$ always. Thus, any malicious order (an order that requests the topping with a zero ranking in popularity) will always have a larger priority than any benign order.

If an attacker continuously places orders with topping zero, their orders will always be processed first, causing what is known as process starvation (see, e.g., [34]), and the orders by benign users will never be processed.

All of the blue teams identified an unintended vulnerability in lieu of the priority vulnerability. This impacted the intended benign variant as well. See below for details.

In the benign version, topping zero is ignored, and therefore all orders have a priority ≤ 215 − 1 = 32767. Because ties in priority are settled by a coin toss, an attacker that places orders with the maximum priority $\mathcal{P}$ = 32767 (which corresponds to orders wherein all 15 toppings are ordered) has no definite advantage over benign users.

6.14.1.4 Large-Memory Order (AC-S Memory)

The cooking of an order is mimicked by using an LFSR with the requested toppings as input, and calculating and storing the full state-sequence of the LFSR. Because we use a primitive polynomial to generate the toggle mask for the LFSR used in the calculation, the length of the computed full state-sequence is $2^L - 1$, where L is the size of the LFSR.

The sequence calculated thus is stored in a Vector of type Integer. Each variable of type Integer requires 16 bytes; therefore, the storage of the full state-sequence of an LFSR of size L will be on the order of $16 * (2^L - 1)$ bytes (the Vector Collection will incur additional overhead). For L = 27 the worst-case memory consumption by the state-sequence storage will be on the order of 2.15 GB.

Recall that the size of the LFSR depends on the particular toggle mask used. To mimic the cooking of an order, the toggle mask is derived from the toppings and quantity ordered. For a certain combination of toppings and quantity ordered, a toggle mask for an LFSR of size 27 will obtain, which will trigger the AC memory vulnerability. Otherwise, the derived toggle mask will be for an LFSR of size between 19 and 22 (depending on toppings and quantity), and these LFSR sizes will not trigger the vulnerability.

The collaborative teams and the control team correctly identified this vulnerability. Utah incorrectly argued that, because there were only a small number of possible toppings and a maximum pizza quantity of ten, it was impossible to launch any sort of AC attack by ordering a pizza.

In the benign version, the size of the LFSRs is ≤ 22, and therefore the memory consumption never exceeds 125 MB.

All of the blue teams correctly identified this variant as benign.

6.14.2 Unintended Vulnerabilities in InAndOut

6.14.2.1 Modified Confirmation Number (AC-T)

All of the blue teams identified an unintended vulnerability that allowed an attacker to prevent a benign user from picking up their pizza by attempting to pick up a pizza order with a modified confirmation number. This was caused by a failure to fully check the validity of the input confirmation number—the application only checks whether its corresponding binary code (based on the pattern of zero versus non-zero entries) matches an existing order. If it does, it will wait until a pizza with the input confirmation number is ready. So, an attacker can simply place an order, get a valid confirmation number, modify any of its non-zero entries to another non-zero number (it is guaranteed to have at least one non-zero entry), and then attempt to pick up a pizza with that modified confirmation number. The application then loops forever waiting for a pizza with the requested confirmation number to be ready, which will never happen.

6.15 LitMedia

LitMedia is a web-based application where clients can log in and view media consisting of open-source poems and pieces of literature. As a client views media, their media preferences are updated based on the categories and tags of the media it views. The application keeps track of preferences in order to present users with media they are more likely to view. Clients are also able to browse media incognito, i.e., client preferences will not be updated.

6.15.1 Motivation for LitMedia

There are no intentional vulnerabilities in LitMedia. The motivation for this challenge program was the potential for a SC leaking a client's top media preferences; however, to increase variation in our Engagement 7 questions, LitMedia was made to contain no vulnerabilities.

6.15.2 Benign Questions in LitMedia

In Engagement 7, various null questions were asked for two benign versions of this challenge program. These null questions include an AC memory vulnerability, a SC-T vulnerability to discover a user's preferences, a SC-ST vulnerability to discover a user's bookmarks list, and a SC-T vulnerability to discover which article a user was viewing. For the AC memory vulnerability null question, there was an unintended vulnerability found by collaborative teams and Utah, another unintended found by the control team, and there were two questions, the bookmarks SC-ST vulnerability and the article viewing SC-T vulnerability, where one of the collaborative teams incorrectly answered that there was a vulnerability present.

6.15.3 Unintended Vulnerabilities in LitMedia

The collaborative teams reported an unintended AC memory vulnerability in LitMedia. This vulnerability was triggered by submitting a POST to the `MediaViewerHandler` about 80 times quickly and consecutively to cause the buffer stream for the associated images to exceed the memory resource budget.

The control team found another unintended AC memory vulnerability where the memory resource budget is exceeded by submitting a POST to add a large number of bookmarks at once.

6.16 Roulette

Roulette is a multiplayer, internet-based version of the real-life casino game of roulette, wherein players have a variety of betting options. Players can place bets by selecting the exact number of the pocket the ball will land in; or they can place their bets on larger groupings of pockets, for example, the pocket color, the parity of the winning number (even or odd number), the dozen to which the winning number belongs (first, second, or third), etc. The variations are manifold [35].

In the real-life game, to determine the winning number and color a croupier spins a wheel in one direction, then spins a ball in the opposite direction around a circular track running around the circumference of the wheel. The ball eventually loses momentum and falls onto the wheel and into one of 37 (in French-European roulette) or 38 (in American roulette) colored and numbered pockets on the wheel.

Payouts for each type of bet are based on its win probability. For example, because there are only two colors, a winning color-bet pays twice the bet amount, whereas a winning bet on a dozen pays thrice the bet amount.

In this application, only a subset of a full roulette game is implemented. The restrictions are the following:

- Bets: Only the following types of bets are allowed:
  - Number: single number in the interval [0,36] (French layout; in American layout betting on 00 is allowed).
  - Parity: even or odd (zero has no parity).
  - Color: red or black (zero's color is green).
  - Dozen: first (1-12), second (13-24), or third (25-36)
- Bet amount: Only integer dollar amounts are permitted.

Remark:
The players play against the House and not against each other.

Remark:
When a player places a bet, the bet amount is immediately deducted from their available bankroll.

Remark:
When a player loses all their bankroll it is evicted from the game. If it reconnects, their bankroll is reinitialized to the initial bankroll amount.

Also, in this application we do not perform a full simulation of the wheel spinning, but make the winning slot a deterministic, ad-hoc function of a randomly generated spin time. Hereafter we refer to the calculation of this function as wheel spinning.

The game sequence is the following:

1. The House announces that bets are allowed (bets-on announcement).
2. Towards the end of the betting period, the wheel is spun.
3. After the wheel has been spun, bets are not allowed. A bets-off announcement is broadcast.
4. The winning number is announced.
5. The House pays off the winning bets.
6. Repeat as long as there are connected players.

6.16.1 Vulnerabilities and Mitigations

6.16.1.1 Blocking Queue (AC-T)

During a betting period (known as the bets-on period), the accepted bets are stored in an array (the bets array) with finite capacity.

When a bet is to be added to the bets array and the array is full, the adding method waits for a certain time until there is room in the array. If the timeout period has elapsed and no room was made in the array, the bet is discarded, and an error message is sent to the player.

However, the bets array is not cleared until the very end of the bets-off period; therefore, if the array is full and a user attempts to add an additional bet the application will block for the timeout specified. After the timeout has elapsed, the application will move to the bets-off period, and the additional bet that blocked the application (and any additional bets that other users attempted to place after the application blocked) will be discarded. Because request processing within the application is single-threaded, the block will cause the application to be non-responsive until the timeout.

We also provided a benign version that sets the blocking queue's capacity to 10,000 bets, preventing an attacker from causing the queue to block within the input budget.

The blue teams all correctly identified this variant as benign. We observed that the collaborative teams noticed the vulnerability in the other variant and they seemed to exert extra effort looking for a vulnerability in this version because we hadn't asked about the other one.

6.16.1.2 Winning Slot (SC-ST)

Once a user has attempted a bet that exceeds their bankroll, a reminder message will be sent out to remind players to bet during each betting period. The amount of elapsed time between the bets-on message (start of the betting period) and the reminder message is randomly generated at the beginning of each betting period. The wheel spin time is equal to the aforementioned elapsed time, and the winning slot is calculated as a function of the spin time and the previous winning slot. Therefore, the winning slot can be guessed before the bets-off period by observing the time between the bets-on message and the bet reminder message, both of which have unique sizes compared to the other Roulette messages.

The sending of reminder messages is triggered by a user attempting to place a bet that exceeds their bankroll; we introduced this trigger so it is not obvious from observing the normal behavior of the application that a side channel is present.

All of the blue teams correctly detected this vulnerability. Although not all of the collaborative teams mentioned the trigger for the bet reminder message.

We also provided a benign version that randomly generates the spin time, so that there is no relation between the time the wheel spins and the timing of the bet reminder message.

All of the blue teams correctly identified this variant as benign.

6.16.1.3 Bet CBC (SC-S)

In this version of roulette, network traffic (beyond the initial session establishment) is encrypted using CBC mode encryption without a MAC. This is vulnerable to what is known as a "padding oracle attack" [36]. In short, the encryption can be broken simply by an attacker being able to determine whether a submitted message is correctly padded. In this version, the size of the response message reveals whether the padding was correct.

The encryption method encrypts plaintext in blocks of fixed length, with the last block of the message padded to the block length. In particular, we use PKCS5 padding, in which blocks are of length 16 and each byte in the padding is the total number of padding bytes. For example, if the last block has 10 bytes of data, then there are 6 bytes of padding, and the padded data would be:

$$[x_0, x\ 1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, 6, 6, 6, 6, 6, 6]$$

If the last block has only 1 byte of data, then there are 15 bytes of padding, and the padded data would be:

$$[x_0, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15]$$

In the event that the last block is exactly 16 bytes, an extra block of only padding:

$$[16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16]$$

is added.

If the application attempts to decrypt a message where the plaintext is not correctly padded, an exception is caught and it sends an error response which has a length of 288 bytes. If not, decryption succeeds and the message is passed to the `SuspicionDispatcher` for handling. If its contents represent a legitimate Suspicion message, it will be processed accordingly, but, if not, it will fail with a protocol buffer exception. In both of these cases, the response message will always have a size other than 288, so an attacker can easily determine whether their message was correctly padded from the response packet size.

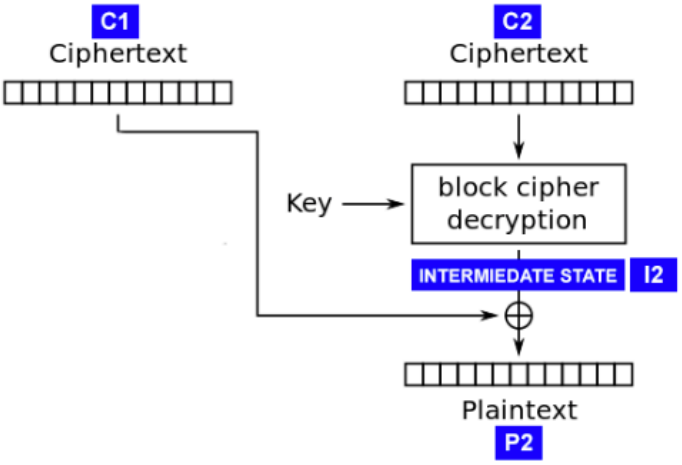


Figure 8: The intermediate state in CBC mode decryption.

Say the padded plaintext message contains consecutive blocks P_1, P_2 and corresponding ciphertext C_1, C_2. As shown in figure 6-6, CBC mode encryption works in such a way that there exists an intermediate block I_2, such that $I_2 = C_1 \oplus P_2$ and $P_2 = C_1 \oplus I_2$, where $\oplus$ denotes bitwise exclusive or.

An attacker, *M*, can intercept a message from benign user *A* to benign user *B*. It then repeatedly modifies and re-sends parts of the message to *B*, updating it as the attack progresses, and making it look like it came from *A* so that *B* will continue using the corresponding cryptographic key. Given two consecutive blocks C_1 and C_2 of the ciphertext, it modifies C_1 and sends $C_1' || C_2$ (where || denotes concatenation), in order to obtain P_2. Starting from the last byte, for each index *i*, it tries all possible values for that byte, noting (as determined by timing) which one receives a response other than the padding error. For the last byte only, it needs to verify that the valid padding is actually [1], and not [2, 2], [3, 3, 3], etc. In most cases it will be [1], but it can verify this by modifying the byte at the next-to-last index of C_1' and verifying that the padding is still correct. (If not, it should continue modifying the last byte of C_1' until it obtains another message with valid padding.) Once it has obtained the desired padding at index *i*, it can determine the actual plaintext byte for that index, $P_2[i]$.

As an example, suppose a value of 27 yielded correct padding (verified as padding of 1) at index 16. First, the attacker can obtain the intermediate value,

$$I_2[16] = C_1'[16] \oplus P_2'[16] = 27 \oplus 1 = 26.$$

Then it can determine the actual plaintext value at the 16th index in P_2,

$$P_2[16] = C_1[16] \oplus I_2[16] = C_1[16] \oplus 26.$$

Note that it has the ciphertext, so $C_1[16]$ is known.

Furthermore, it can deduce what byte to use for $C_1'[16]$ to increase that padding by 1 and begin experimenting with the next index. For index 15, it wants $P_2'[15] = P_2'[16] = 2$ to obtain valid padding, i.e., a plaintext ending with [2, 2]. So it sets

$$C_1'[16] = P_2'[16] \oplus I_2[16] = 2 \oplus 26 = 32$$

and proceeds to try all values for $C_1'[15]$ until it finds one that yields valid padding. And so on for the remainder of the block, and any additional blocks.

He is thus able to undo the encryption of all but the first block of the message (which does not contain the secret of interest anyway).

Attack optimization: Given that the attacker knows the structure of the bet message it is decrypting, it can limit their efforts to a small portion of the message.

From the value in the message, the type of bet can be deduced, so only the value and the amount need to be decrypted. The structure of the end of the bet message is:

```
| value header | value length | value bytes (at most 6) |
amount header | amount bytes (at most 2) | padding bytes |
```

The last two bytes of the value are sufficient to determine the value, which is either a one- or two- digit number, or “first”, “second”, “third”, “red”, “black”, “even”, or “odd”. So it is sufficient to decrypt the last byte, which reveals the amount of padding, and then at most 5 more bytes, for a total of 6 bytes to decrypt.

Further, for many of these locations, the options are much more restricted than all 256 possible bytes. The amount header is always 24, the padding is between 10 and 15, the first amount byte is at most 128, and the last amount byte (if there are two) is at most 78 (variant representation of a number that is at most 10000.)

Additionally, the value bytes are limited to digits and characters that appear in the last two characters of the value names — 22 options for the first value byte, and 18 for the second. So, the attack requires at most 6 + 128 + 78 + 22 + 18 = 252 active operations, plus one passive operation to obtain the packet to decrypt.

None of the blue teams detected this vulnerability. The teams focused primarily on the code where the secret appeared and on passive observation of related packets. The collaborative teams did, in passing, consider whether they might break the encryption, but they only looked at the RSA encryption which is used for sharing a symmetric key, and did not consider the encryption used for the remainder of the communication.

We also provided two mitigated versions of this vulnerability. The first is vulnerable to a padding oracle attack, but is not exploitable to the required probability of success within the resource budget. In this version the size of the response message that reveals that the padding was incorrect is occasionally (0.8% of the time) the same size as one of the possible (and fairly common) response messages when decryption is successful, but the packet is not a valid application message. Since the question provided to blue teams requires 100%probability of success, this version is benign, since, in the worst case, an attacker could get many messages of conflicting size and use up their allowed operations eliminating the uncertainty they cause.

Having failed to detect the vulnerable variant, all of the blue teams correctly believed the variant with this mitigation to be benign.

The other mitigated version uses a MAC to prevent an attacker from being able to replay modified messages to learn about message contents, thus eliminating the opportunity for a Padding Oracle Attack. This version was not included in the engagement, for fear that blue teams would notice the difference, which would point them directly to the vulnerability in the other variant (e.g. by performing an internet search for CBC without a MAC.)

There were no unintended vulnerabilities detected in Roulette.

6.17 Suspicion

Suspicion, which appeared in Engagement 7, is a World War II themed peer-to-peer multi-player game, based on the game SpyFall [37].

There are n players, where n is at least 3. In each game, a player is randomly selected to be the spy and is informed of such, while the other $n - 1$ players are informed of a random common (secret) location from a fixed set of possible locations, as well as a password. The goal of the non-spies is to determine who the spy is, and the goal of the spy is to determine the secret location.

In the spirit of the theme, messages are sent in Morse code, and the users have the option of viewing them in Morse code. The default is that the client automatically decodes them before displaying them to the user.

The game proceeds as follows: A single player picks another player of whom to ask a question, with the intent of demonstrating that it knows the secret location and/or determining whether the other player knows the secret location, all without revealing the location to the spy. That player answers the question, and then it is their turn to ask a question to another player. All players are able to view all of the questions and answers. The game ends after the spy chooses to guess the location or a non-spy chooses to accuse someone of being the spy. To prevent the spy from winning by accusing itself, the password is required when accusing someone of being the spy.

The game client of the player who initiates a game serves as the communications hub and ensures players take turns appropriately. We refer to this player as the game leader.

Example questions: Do you come here often? What brings you here today? Do you like coming here? What's the average age of those here with us? Would you bring your family here? How did you get here?

6.17.1 Intended Vulnerabilities in Suspicion

6.17.1.1 Location SC (SC-S)

This side channel allows the spy to learn the location of the game.

There are 26 possible locations in which each game can take place. The location is known to all players except the spy, whose goal is to determine the location. Recall that players attempt to learn the location or the identity of the spy by asking each other questions. The location is leaked via the sizes of the first three answer messages sent by the game leader in the game, each of which reveals one trit of the base-3 representation of the location index. Note that all answer messages in the game are sent through the game leader, who is the communications hub. The game leader's client enforces that the game cannot be terminated prior to 3 rounds of questions and answers in the game, ensuring that the game does not end before the spy has a chance to learn the location.

Messages are sent as Google protocol buffers [38], but contained within a longer byte array, where the first two bytes indicate the actual size of the protocol buffer. For a trit of 0, this longer byte array will have size 1002, for a trit of 1 it will have size 1003, and for a trit of 2 it will have size 1004. So, if the first three answer messages have size s_1, s_2, and s_3, then the location index is $i = (s_1 - 1002) * 9 + (s_2 - 1002) * 3 + (s_3 - 1002)$. The location is then the i^{th} element in SuspicionGame.possibleLocations.

Questions and answers are limited to 40 characters each by the sender's client, and user IDs are limited to 25 characters, guaranteeing that 1000 bytes is sufficient for the answer messages (even after encoding the answer in Morse code).

In E7, only Utah correctly answered this question. Both the control team and some of the collaborative teams identified the potential vulnerability in the code. However, after running the program, they observed that the packet sizes on the wire didn't change, and decided that there must be some padding added at another stage that mitigated the vulnerability. While it is true that the encryption padded the packets, what they missed was that the attacker was a player in the game and could see the decrypted packets, whose sizes revealed the secret.

(We suspect that Utah benefited from not having looked at packet sizes, so their correct answer was likely the result of a less complete analysis, rather than more.)

This vulnerability is mitigated by incorrectly updating the round number, which is used to determine which trit (the first, second, or third) should be sent. In this version, the round number is updated every time `getRounds`() is called, so that by the time the first answer is sent, the round number has already passed the trits of the location, and the size of the message – 1002, 1003, or 1004 – is randomly determined.

In Engagement 7, both Utah and the control team answered this question correctly, but our mitigation misled the collaborative teams. They did observe that there was a difference in how the rounds were updated, but they failed to realize that it interfered with the side channel. Further, dynamic analysis revealed that, in this version of Suspicion, the packet sizes vary. A difference in the encryption algorithm allowed the actual (random) packet sizes to be viewed, but the collaborative teams assumed the difference was due to the side channel. They did not bother to verify that the changing packet sizes correlated with the location.

6.17.1.2 Morse Code Integer Overflow (AC-Disk)

Suspicion sends questions and answers encoded in Morse code, and "compressed" using run length encoding. When processing a received question or answer, it decompresses the answer to disk. There is a guard to ensure that this does not cause excessive disk space usage, however, as shown in snippet 9, the guard uses an integer to hold the total bytes written, and it can overflow, causing the guard to miss that the limit was exceeded. This is a re-creation of a vulnerability from WithMi (see section 6.8.1) that none of the blue teams detected in Engagements 3 and 4.

```
int totalSize = 0;
while ((b = inStream.read()) != -1) { // read byte
    int num = readInt(inStream); // read count for that byte
    totalSize += num;
    // attempt to make sure we haven't exceeded MAX_SIZE
    if (totalSize > MAX_SIZE) {
       throw new IOException("Data exceeded maximum allowed size ");
   };
   writeRepeated(b, num, outStream);
}
```

Snippet 9: Integer Overflow Allows Excessive Disk Space Usage.

All seven blue teams identified the vulnerability in this code; however, they exploited it by sending a negative count, rather than by overflowing the integer. (Utah did notice and comment on the potential for integer overflow.)

Rather than simply mitigating the vulnerability by properly using a long to hold the total number of bytes read/written (and risk the blue teams noticing the difference, pointing directly to the vulnerability), we instead added an additional check, in a separate method, which checks whether the total number of bytes has decreased.

All seven blue teams correctly observed that the vulnerability was mitigated in this version of Suspicion.

6.17.1.3 CBC Padding Oracle (SC-ST)

This version of Suspicion contains a side channel in time and space that allows a third party to decrypt application messages, in particular the game assignment message, which contains the password. The space component is used only to identify that message, and the decryption is done using timing information only.

In this version of Suspicion, network traffic (beyond the initial session establishment) is encrypted using CBC (Cipher-Block Chaining) mode encryption, without a MAC. This is vulnerable to what is known as a padding oracle attack [36]. In short, the encryption can be broken with a chosen ciphertext attack, simply by an attacker being able to determine whether a submitted message is correctly padded. In this implementation, the timing of the response message reveals whether the padding was incorrect.

If the application attempts to decrypt a message where the plaintext is not correctly padded, an exception is caught and it sends an error response. If not, decryption succeeds and the message is passed to the `SuspicionDispatcher` for handling. If it is a valid Suspicion message, it will be processed accordingly; otherwise it will fail with a protocol buffer exception. Naturally, the padding failure is faster than successfully completing decryption and parsing the message, so an attacker can determine whether the message was correctly padded from the response time.

A slight variation on this vulnerability appeared in Roulette (section 6.17.1). There, the response *size* revealed whether the padding was correct, while, in Suspicion, the side channel is in the timing. See the discussion in section 6.17.1 for further details of the vulnerability and how it can be exploited.

Note that the timings are affected by Just-in-Time (JIT) compilation, so the attacker should ensure that the server performs some warm-up decryptions before using any timing information for the attack. We found that 650 warm-up operations were sufficient, with the attacker then using four timings for each

guess on the first byte decrypted, and three timings on each subsequent byte. The warm-up operations can be done before the game is accepted.

None of the blue teams detected this vulnerability. The teams focused primarily on the code where the secret appeared and on passive observation of related packets. The collaborative teams did, in passing, consider whether they might break the encryption, but they only looked at the RSA encryption which is used for sharing a symmetric key, and did not consider the encryption used for the remainder of the communication.

This vulnerability is mitigated in Suspicion by not using CBC encryption mode at all, using CTR (Counter) mode instead.

Having not detected the vulnerability in the other version, the blue teams correctly identified this version as benign.

6.17.2 Unintended Vulnerabilities in Suspicion

All seven blue teams found an unintended memory vulnerability in Suspicion, which simply used their disk space exploit. Had they used an overflow exploit, it would have instead caused the JVM to throw an exception on an attempt to create a large array, rather than actually using excessive memory.

7 Results and Discussion

7.1 Tests and Vulnerability Proofs

For each challenge program, we developed a set of benign tests that showcase the benign functionality of the program. Additionally, for each vulnerability we developed a test that demonstrated the presence or absence of that vulnerability.

7.2 Benign Tests

Benign tests were implemented with a combination of bash scripts, Python, and Expect scripts, and they were shared with the blue teams to demonstrate usage of the application. For the purpose of running these tests in our automated nightly builds, we normally verified success by confirming that the output of the test was exactly as expected; in a few cases we simplified by searching for a particular chunk of output that matched what was expected.

7.3 Side Channel Vulnerability Proofs

Our approach to side channel proofs in the networked reference environment may have been overly complicated due to a misunderstanding regarding whether the attacker is allowed to do anything beyond observing network traffic on the masterNUC. For side channel proofs involving a third-party observer, we had an attacker on the clientNUC who would send start/stop messages to a listener on the masterNUC, which would send back the relevant information about the network packets seen during the collection period. (In most cases where the side channel was exploitable by an active participant in an application session, it was unnecessary to use the masterNUC and the attacker could simply observe the packets itself sent and received.)

7.4 AC-T Vulnerability Proofs

In the first engagements, our proofs for AC-T vulnerabilities would simply check the amount of time it took for the attacker to receive a response to their (last) attack request. However, later on, we improved our testing mechanism to verify that a benign user's request was impacted (and not just the attacker's request). Thus, we had a benign user that continuously submitted requests during and after the attack, and watched whether any of these requests failed to receive a response within the time bound. This worked much like the side channel proofs, employing a listener on the masterNUC to observe the benign user's requests and responses.

7.5 AC Memory and AC-Disk Vulnerability Proofs

Our proofs for AC Memory and AC-Disk vulnerabilities simply continuously monitor the memory or disk usage in the server docker container after sending malicious input from the attacker. We also eventually included verifying the output of the test, to ensure that, when we found memory or disk space not exceeded, it wasn't merely because the attack had failed to run.

8 Difficulties

In this section, we discuss some of the challenges we faced as an adversarial challenger in the DARPA STAC program.

8.1 Unintended Vulnerabilities

Our greatest enemy in this program quickly became unintended vulnerabilities. They plagued us at every engagement, with AC vulnerabilities posing a significantly greater threat. Given that the goal of the program was to develop tools for detecting such vulnerabilities, there were obviously no off-the-shelf tools we could use to detect them so that they could be eliminated. We did invest some effort into trying to use some of the better blue team tools to see what they might find, but we were not able to get any usable results out of them. What we found most helpful was Apogee's request that we document the guards on user inputs and where they are implemented. We did catch several omissions in the process of writing those detailed descriptions. However, many unintended vulnerabilities still marred the engagements.

This issue was seriously exacerbated by the programmatic decision to only ask blue teams to find one vulnerability per question. (We referred to this as the "any versus all" debate.) Had blue teams been required to find *all* relevant vulnerabilities, the unintended ones would have been a nice bonus. However, Apogee felt strongly that it was unreasonable to have to rule on what constituted distinct vulnerabilities in order to fairly score teams in the presence of multiple vulnerabilities. Thus, much of our hard work on creating interesting and subtle vulnerabilities went down the drain when blue teams found lower-hanging unintended vulnerabilities instead. (CyberPoint did not provide additional clarification why it would be reasonable that all vulnerabilities were not known.)

Further, CyberPoint's challenges were more severely impacted than BBN's by this decision, due to our approach of having multiple variants of an application, which often left multiple challenges vulnerable to the same unintended exploit (which the blue teams could easily just throw at it, rather than exercising their tools). Limiting challenges to two variants per engagement slightly reduced this impact, and allowed us the opportunity to eliminate unintended vulnerabilities between engagements. We are pleased that in Engagement 6 we were able to present a variant of BattleBoats that was free of unintended vulnerabilities (or at least free of unintended vulnerabilities that any of the blue teams were

able to detect). Finally, in Engagement 7, blue teams didn't find any unintended vulnerabilities in either variant of Roulette or CyberWallet, and there was likewise one variant each of LitMedia and Suspicion that was free of unintended vulnerabilities.

8.2 Just-In-Time Compilation

While more readily known as a source of side channel vulnerabilities [39, 40], JIT compilation actually served as a hindrance to several of our timing side channels.

The JIT compiler dynamically optimizes the code as it's run, focusing on portions of the code that are responsible for a significant amount of execution time. Java has two separate JIT compilers, C1 and C2. C1 performs fast, lightweight compilations of bytecode to native code, while C2 is slow and sustains a high memory overhead, but creates highly optimized compilations from bytecode to native code. JIT compilation combines both of these compilers, starting with C1, and moving to C2 in cases where that seems insufficient.

The effect of JIT compilation is that the time that elapses between any two given events may vary significantly from one call to another, with a general decreasing trend, usually stabilizing once the running application has been sufficiently exercised. This was particularly relevant in STAC because vulnerability proofs (i.e. attacks/exploits) are typically run against a server that has just started up, when these effects are typically most dramatic. In most cases, we had to allow an attacker many extra operations to ensure that the server/victim was "warmed up" before beginning the attack.

8.3 Unexplained Behavior

We encountered an interesting obstacle when developing the confirmation code side channel in InAndOut (section 6.14.1). This side channel reveals a customer's pizza order confirmation code via the order response delay. We needed to gather data on many orders to craft an exploit that would prove the side channel's existence. We confirmed via VisualVM [41] that JIT compilation completed optimization after approximately 1000 orders, but after about 3000 orders we noticed a decrease in response delay, with increasing scatter in response delays for a given confirmation number.

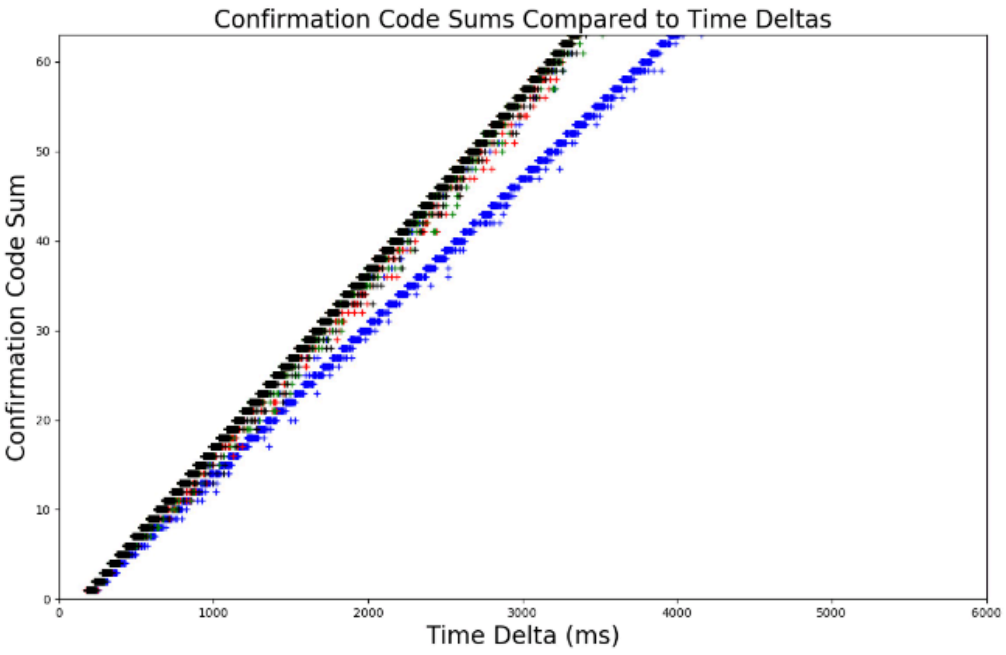


Figure 9: InAndOut timing side channel inconsistencies: confirmation number vs response delay (milis). The graph consists of 11988 orders. The first 3000 orders placed are shown in blue.

We were unable to determine why so many "warm up" orders were required in order for the response times to stabilize. This inconsistency in execution times eventually led us to implement a server shutdown after 1000 orders to confine the relationship between confirmation number and response delay to a single band.

8.4 The Switch to Java 8

While switching to Java 8 was clearly the correct thing to do (and long overdue), it had the undesirable side effect of breaking at least one of the vulnerabilities we had already developed but not yet presented in an engagement. The cache-miss vulnerability in SimpleVote (section 6.11.1) had originally been implemented and tested in Java 7, where it relied on the slowness of Java 7's BigDecimal implementation. When we upgraded to Java 8, the vulnerability was no longer present. We created our own, slow, implementation of the BigDecimal arithmetic used, in order to revive the vulnerability for Java 8.

8.5 Reasoning about Attack Budgets

Reasoning about attack budgets is a tricky business. In order to test the blue teams' ability to do this, we asked a few questions that were intended to be benign by virtue of the attack budget being insufficient. However, that required us to be able to correctly reason about the necessary attack budget.

As an example of the challenge involved in reasoning about attack budgets, in our CBC vulnerability in Engagement 7, intuitively, the attack would require some 8000 active operations. However, using only simple optimizations (see section 6.17.1) relating to the potential message contents, we were able to reduce it to roughly 250. Unfortunately, none of the blue teams noticed the CBC vulnerability at all, so we were unable to see whether the surprising budget interfered with their ability to determine its viability.

The blue teams were able to out-reason us on the budget for a SimpleVote vulnerability in Engagement 6. There was a known vulnerability that required submission of 8 registration keys to exploit. In addition to 462 bytes to log in, we believed each key submission to require 510 bytes, which included a TLS handshake, and therefore set the total input limit to 4000 bytes, believing that the challenge would be benign at that budget. However, one blue team believed during the take-home portion that they could reduce the POST header sizes to achieve the exploit within the budget. At the live engagement, the other teams needed convincing, and they were eventually able to exploit the vulnerability within the given budget, not by reducing the header sizes, but by maintaining a TLS session to avoid repeated TLS handshakes.

Even more precarious was reasoning about attack budgets on our implementations of RSA cryptography using the Chinese Remainder Theorem. We had an attack that succeeded in under 62,000 operations. However, in the literature [21], this implementation is exploitable in only 300 operations, albeit with timing operations on a smart card. We used some statistical tricks from [22], as well as some tricks of our own that we stumbled onto, in order to make it feasible in the noisier Java setting. It was only after we started seeing blue team responses that we realized we had no way of knowing whether an attack was possible with fewer operations and whether all of our mitigations were sufficient. The Blue teams seemed to be equally at a loss, citing attacks on RSA in the literature without discussing the attack budget at all.

One mitigation kicks the user out of the connection handshake if they fail to respond correctly to a connection challenge, rather than allowing them to try again, quadrupling the number of operations needed for the attack. However, we cannot guarantee that there is no attack more efficient than ours that would be able to succeed in spite of this.

Another mitigation does not use the Chinese Remainder Theorem. In the literature, attacks on RSA implemented this way require 5000 operations for a 512-bit key when timing operations on a smart card. Given that our key is twice as long, and the environment much less accommodating, we assume that no exploit exists within an attack budget of 75,000 operations.

In another, the decrypt algorithm decrypts a random multiple of the actual message. Then, it uses basic modular inverse operations to get the message. Since the attacker does not know what value is actually being decrypted, it cannot use our attack to determine the key. We believe this to be safely benign, but one never knows when a new cryptographic attack will be discovered.

Finally, another mitigation balances the vulnerable branch, performing the extra reduction repeatedly regardless of whether it's needed. We believe this to be the most secure of our mitigations.

8.6 NUC Availability

Although we possessed 12 NUCs (likely significantly more than any other STAC performer), there were occasions when developers were stuck waiting for a NUC cluster to become available so they could complete a task. The development of timing side channels required much heavier NUC use than any other vulnerability type, due to both the difficulty of developing the vulnerabilities and attacks (which could only legitimately be tested on the reference platform), and the typical longer duration of attacks. However, proofs for all types of vulnerabilities had to be tested on the reference platform. With five developers and only four network clusters, this occasionally led to some contention, particularly with multiple variants of each challenge program to be tested. (CyberPoint did not provide additional clarification regarding the issues derived from the limited amount of NUCs and how to resolve these issues.)

8.7 Platform-Dependent Timing Side Channels

Due to limited NUC availability, our code was developed on machines that were faster and more powerful than the reference platform. Because of the difference in performance, we found that timing SCs that worked on our development machines needed to be adjusted to work on the NUCs.

A case in point is the timing side channel in InAndOut (section 6.14.1). In this side channel, the confirmation number was leaked through the different response delays when placing a pizza order.

Figure 8-2 is a comparison of the relationship between confirmation number and response delay in the development platform vs. the reference platform.

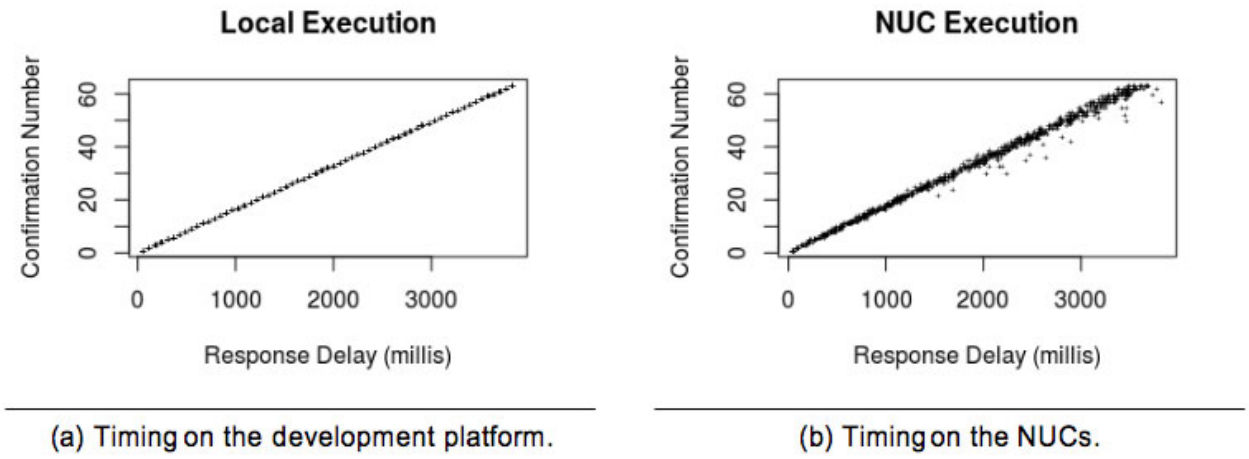


Figure 10: The Confirmation Number Timing Side Channel in InAndOut.

While we expected to see an increase in the time differentials, we were surprised that, in going from the development platform to the reference platform, a one-to-one relationship became a one-to-many one. In this case, on the reference platform a single confirmation number mapped to, at most, five different timings. (A simple solution was to allow the blue teams up to four oracle queries.)

We investigated two possible sources for this kind of discrepancy: JIT and garbage collection.

In many cases, as discussed in section 8.2, we surmounted the changes in timing introduced by JIT with warm-up operations. To overcome the noise introduced by garbage collection, it was necessary for us to locate the constructs in the code that triggered garbage collections, and make the appropriate changes (for example, changing a local variable to a class variable reduces garbage collection at the expense of a larger memory footprint).

Sometimes, however, as in this example, we could not find an explanation for the deterioration of the SC on the reference platform, and we had to resort to changing the attack budget.

8.8 IntelliJ Transformations

While the use of IntelliJ's PSI library for code transformations brought great benefits, it was also fraught with difficulties. While their code transformations may work well when used manually and intentionally in the IDEA development environment, blindly applying them at a large scale revealed many edge cases that IntelliJ didn't handle correctly. Due to the nature of our transformation system, minor code changes could cause significant changes in what transformations were applied. Unfortunately, this would

sometimes cause us to encounter these bugs. In most cases, the bugs yielded code that didn't compile, with refactoring issues such as pulling code into a separate class that requires access to a private class field, failure to include the necessary imports when moving a code block to a separate class, or creating a new variable or method with a name that was already used in the class. However, there were also bugs that caused things like re-ordering of operations in code where the order affects the outcome of a computation, which yielded compilable code containing subtle errors. While the JetBrains team was generally quite responsive in getting bugs fixed, we could not afford to rely on that, and there were many times when we disabled transformations on a code block, modified our transformation code to prevent a mishandled case, or changed our source code or transformation seeds to prevent these bugs from introducing errors in our generated source code.

8.9 Side Effects of Vulnerabilities

8.9.1 TCP Flood from CBC Attack

The cipher block chaining (CBC) attack is unusual in that it requires spoofing packets to appear as part of another user's established connection. We accomplished this using Scapy [42], crafting TCP packets that were successfully received as legitimate and as coming from another user on another machine. However, this had a devastating side effect of causing a TCP flood because the victim remains connected at the same time. The TCP protocol reacts on behalf of the victim when the server acknowledges packets allegedly from the victim that it had never sent. Inexplicably, we had one network of NUCs where the attack was able to succeed in spite of this, while our three other NUC networks became overwhelmed and caused the packets needed for the attack to get frequently dropped. We invested some effort in investigating what might have caused this difference and how we might alleviate the problem. But in the end, we had to abandon our use of TCP and modify our communications framework to work with UDP as well.

8.9.2 Vulnerability Overlap

It often occurs that an AC-Disk or AC Memory vulnerability also manifests as an AC-T vulnerability. This is not too surprising, as it may take some time to exceed a memory or disk usage threshold. As a result, we had to be careful to ensure that our AC-Disk and AC Memory vulnerabilities did not get interpreted as unintended AC-T vulnerabilities. In earlier engagements, we could get around this issue by adding more variants to separate time and space AC questions. However, large numbers of variants were problematic due to other unintended vulnerabilities being applicable across variants, so, starting from Engagement 5, we were limited to two variants per application per engagement. This kept us from asking about a benign variant of one of the vulnerabilities in BattleBoats in Engagement 5.

Similarly, Calculator had two vulnerabilities — the padding vulnerability in FFT, and the parsing vulnerability — that also required a vulnerable version of the webserver to allow posts of the size needed to exploit them. Fortunately, there were several distinct contributing factors to the webserver vulnerability, so it was possible to separate them and only use the increased allowed POST size to accommodate these vulnerabilities, while not having the actual webserver vulnerability present in these variants.

9 Takeaways

9.1 Avoiding Unintended Vulnerabilities

Throughout the program, there was much emphasis on encouraging the Red teams to avoid unintended vulnerabilities in their challenge programs. However, the point of the program was for the blue teams to develop the ability to detect these vulnerabilities because state-of-the-art methodologies and technologies could not. Thus, it would have made more sense to ask the blue teams to find all vulnerabilities—intended *and* unintended—in the challenge programs than to expect the Red teams to detect and prevent them. Given that the blue teams were expected to be able to determine whether a challenge program was free of vulnerabilities, they could just as easily determine whether a given vulnerability was the only one or if there were others. The only reason they were not asked to do so was because it was anticipated that this might cause difficulty in scoring: how would it be determined whether two claimed vulnerabilities were truly distinct or actually just different ways of exploiting a single vulnerability, in order to correctly award points for vulnerabilities identified? While we appreciate this concern, we feel that the purpose of the program was not to definitively determine which research group could find the most vulnerabilities, but to encourage the development of the strongest tools possible. We believe that, even if there had been some challenges in scoring due to multiple vulnerabilities, it would have had limited effect on the ranking of the teams, and, more importantly, would not have hindered the program from reaching its goals in any way. In fact, it would have further exercised the research tools and granted more insight into their capabilities and limitations.

That said, we offer some insights into the problem of preventing unintended vulnerabilities.

One useful measure is to identify, for all code locations where user input is processed, what guards are in place there, what assumptions the developer is making regarding the input, and whether they are enforced. Many unintended vulnerabilities resulted from incorrect assumptions regarding user input.

Another useful step is to study specific previous unintended vulnerabilities and make sure they are not applicable. This can help identify assumptions that developers don't even realize they're making, for example, that an input is positive, not provided in scientific notation, etc.

In retrospect, however, the best thing we could have done to prevent unintended vulnerabilities would have been to have our own, full-time, internal blue team, focused on finding unintended vulnerabilities and developing advanced tools for doing so. Unfortunately, the blue teams' tools that were available to us were not in a state that facilitated their use by outsiders, even though we invested a good amount of time in trying to benefit from them. And, while the very basic fuzzers that we wrote did find a few unintended vulnerabilities, in spite of running them nightly for an extended period of time, they failed to find many of them.

9.2 Collaborative Engagements

The idea of collaborative engagements was an interesting one. We expected that whichever team had found a valid vulnerability (when there was one) would quickly show it to the other teams and a consensus would be reached. In many cases, the discussion did take that route. However, this was a social experiment as well, and we saw cases where the most confident *personality* in the group would take the lead, even when their answer was clearly incomplete, and it could take quite some time for another team to speak up with a better answer. Rare (if existent at all) were the cases where the

collaboration went deeper than sorting out which team had the most convincing answer. Cases where the teams failed to reach a consensus were also rare, and were often due to someone missing something, rather than genuine disagreement.

In Engagement 7, the effects of the significant increase in the number of questions was quite apparent. We saw several cases of the collaborative teams lacking thoroughness in their analysis, particularly on the Suspicion location side channel (section 6.18.1), where seeing that packets had different sizes was sufficient to convince them that a side channel was present, and seeing that they did not was sufficient to override the fact that they had seen a side channel in the code; they did not verify that the changing sizes correlated with the secret, nor did they look to see where the padding was coming from and whether they had access to the unpadded sizes.

9.3 Blue team Capabilities

9.3.1 Research Tool Weaknesses

Over the course of the program, the research teams made much progress in terms of handling challenges of a realistic size, and greatly improved their accuracy in detecting vulnerabilities. Some clear weaknesses remain, however, particularly with regard to detecting side channels. Their focus remains on code where the secret is directly accessed in the code, leaving detection of cryptographic vulnerabilities to manual inspection with knowledge of previously published vulnerabilities. Additionally, even when the vulnerability is tied to the use of the secret in the code, their analysis is rather primitive, with an inability to look for correlations between the secret and a *combination* of timings, sizes, or both. (They may look at more than one of these things, but only sequentially, not in combination.) This was demonstrated with the InAndOut confirmation number side channel (section 6.14.1), where the amount of random noise that had been added to the timing was revealed by a packet size, and every single blue team missed it entirely. We feel that the analysis this requires is a capability that could readily be added to their tools.

We also noted apparent difficulties in identifying code that writes to disk. None of the research teams even looked at the code related to the disk write vulnerability in Calculator in Engagement 7 (section 6.12.1). Rather, their analysis of possible disk space vulnerabilities in this application was limited to what went into the log file. Our initial conjecture was that they only looked for uses of packages java.io and java.nio. However, our MultipartHelper class imports both java.io.File and java.nio.file.Files. It even uses File.copy(). The actual vulnerable disk write, however, is called from within FileUpload.parseParameterMap(). We would have expected the blue teams' pre-engagement analysis of APIs used and methods called to have highlighted this as a method to watch for disk writes.

9.3.2 Control team Versus Research Teams in the Final Engagement

It is worth noting that two of the questions on which the collaborative teams outperformed the control team in Engagement 7 were due to the control team's misinterpreting a question about CyberWallet, which spoke about a target account, as if there was a specific bank account number being targeted. (For the record, this was not our wording.) This led them away from looking for the intended side channel, which leaked the account number. We feel that this does not reflect in any way on the superiority of the research teams' tools, and we believe that their capabilities are more closely matched than the Engagement 7 results might suggest. Putting those two questions aside, there was only one CyberPoint question on which the research teams outperformed the control team, while we had two questions that the control team answered correctly where the collaborative teams did not. It is interesting to note that

the research teams outperformed the control team on BBN's questions more significantly than on ours. Unfortunately, we are not familiar enough with their challenges to be able to draw any useful conclusions from that observation.

9.4 Miscellaneous

9.4.1 Other JVM languages

According to the original Broad Agency Announcement (BAA) for the STAC program [43], the blue teams were required to analyze Java bytecode, which indicated that any JVM language could be used in developing the challenge programs. Java was the most popular option and it was used in developing all of our programs. This was due to a programmatic decision by the program manager and Engagement Lead to not allow other languages because the blue teams were having enough difficulty with Java.

Recall that the blue teams receive a compiled JAR file of each challenge program, which they run through their tools and decompile to inspect. Because all JVM languages are compiled to Java bytecode, it was of interest to us to investigate the differences between compiled bytecodes of the same logic in both Java and another JVM language to determine if there would be a worthwhile benefit to developing a challenge program using the latter.

We compared Java to Scala. Scala is an object-oriented JVM language that supports functional programming by design. Some other features that distinguish Scala apart from Java are more concise code (the same task can usually be done with less code in Scala than in Java), immutable variables by default, higher level pattern matching, support for tail call recursion optimization, simplified thread communication and control, etc. Snippet 10 and snippet 11 show two code blocks that perform the exact same task following the same logic (with obvious syntax differences).

```
public static void main(String[] args) throws Exception{
    System.out.println("Starting program!");
    int x = 10;
    double N = 10000000005.0;
    double z = 0;
    int i;
    for(i = 0; i<x; i++){
        z+=N;
    }
    double w = z/x;
    if((long)Math.abs(N - w) != 0){
        Thread.sleep(10000);
    }
}
```

Snippet 10: Java code example.

For reference, the bytecode corresponding to snippet 10 and snippet 11 are provided in appendix B.

There are noticeable differences between the two sets of bytecode. One difference is that the Scala bytecode is longer than the Java bytecode by about 30 lines. Another visual difference is that the Java bytecode doesn't have as many calls or functions, making the Scala version look more scattered and

complex. When decompiled back to Java and Scala, respectively, the Java code looked the same as it originally was while the Scala code included some additions: mainly using `predef` to provide function definitions such as `print` and `intWrapper` and adding static members for default objects.

```
def main(args: Array[String]): Unit = {
    println("Starting program!")
    var x: Int = 10
    var N: Double = 10000000005.0
    var z: Double = 0
    var i = 0
    for (i <- 0 until x) {
        z += N
    }
    var w: Double = z/x
    if (Math.abs(N - w).toLong != 0) {
        Thread.sleep(10000)
    }
}
```

Snippet 11: Scala code example.

A comparison between the Java and Scala bytecodes provided evidence that using Scala may have been advantageous against the blue teams. Though the blue teams were made aware that other JVM languages may be used, it is uncertain how well their tools would be able to work on Scala bytecode and if they would have run into problems. It is possible that the blue teams may have had trouble with it if their analysts weren't familiar with Scala. However, given that no other JVM language currently approaches Java's level of popularity, we don't feel that the program lost out significantly by not including other JVM languages, and we feel that the program likely benefited from allowing blue teams to focus on a single language.

10 Conclusion

We have described the applications and vulnerabilities that CyberPoint developed for the DARPA STAC program, the thought processes that motivated them, the tools we used and created to assist in this endeavor, and the results and lessons learned in the process. We are hopeful that our contribution will be a useful asset in the ongoing effort to develop tools capable of detecting a wide variety of software vulnerabilities.

11 Processes and Procedures

11.1 Overview

The approach used to develop and deliver code for this project follows well-defined practices. These include:

- Agile planning and development using Jira.
- Code management using a Bitbucket Git repository.
- Code reviews through pull requests.
- Unit testing.
- Project building using Gradle.
- Continuous and nightly builds using Jenkins.
- Staged delivery for quality assurance.
- Documentation using Confluence.
- Team communication with Slack.

11.2 Agile Planning and Development

The term “agile” is overused and not often correctly applied. Our approach to agile development is no exception in that we have customized the practices to our particular needs. We have adopted some of the best practices of agile development and trimmed or omitted practices that do not add value to our particular project:

Sprints typically last two weeks. This duration is adjusted according to specific needs at the time of sprint planning. Tasks and bugs are added to the backlog and are prioritized prior to sprint planning. Backlog items are assigned to a sprint during sprint planning. The notion of a fixed sprint (once a sprint has started, items in a sprint cannot be added or removed) is not required. It is acceptable for a backlog item to be promoted to the current sprint if a more immediate need arises. Likewise, items in a sprint can be demoted out of the sprint to the backlog. Sprint tasks do not contain time or point estimates nor is the time spent on a task required to be recorded. This project does not need to account for velocity, so this provides more flexibility on the number of tasks assignable to a sprint and frees the developer from having to track time. Tasks are primarily selected by the developer; not assigned.

Prior to sprint planning, a retrospective of the previous sprint is made to review the work accomplished and to understand if there are improvements that can be made to the process.

Daily stand-ups are used to indicate what work has been accomplished since our last meeting, what work will be done that day, and what obstacles, if any, are blocking progress for the developer.

11.3 Code Management

All source code is maintained in a Git repository. No developer may directly merge (push) changes into the main repository. Instead, the developer creates a repository on their Git server that is a fork of the main repository. It is the developer’s server repository that they clone to their development environment.

Changes made to the project are pushed to the developer's server repository and a pull request is created to start the process of merging into the main repository. The pull request indicates the sprint task that initiated the change and is sent to the developers who will perform a review of all code in the pull request. The reviewers provide feedback on errors or issues in the code review and only approve it after the request is satisfactory. After the pull request is approved, the changes are merged into the main repository.

11.4 Software Development

Development of software was done either on a development workstation or on a VM. The choice of an IDE was left up to the developer, with both IntelliJ and Eclipse supported.

Software development always follows the feature-branch-workflow pattern:

4. A sprint ticket is selected from the To Do list and moved to the In Progress state.
5. A pull from the master branch is made to development environment. Remember, this pull will originate from the developer's Git server repository.
6. A new branch is created and named after the sprint ticket.
7. Changes are made to the source code to accomplish the sprint task and committed to the local branch. All commit messages should begin with sprint ticket name.
8. After completing the task and verifying that all tests successfully pass, it is important to ensure that the latest master branch has been merged into the local branch.
9. The local branch is pushed to the developer's remote Git repository.
10. A pull request is created assigning other developers who will perform that code review.
11. Any pull request comments are addressed by the developer and any modifications committed and pushed.
12. After the code reviewers have approved the pull request and the code has been merged into the main repository, the developer moves the In Progress sprint task to the Done state.

The traditional roles of Product Owner, Scrum Master, and Development Team are not differentiated as the development team satisfies all three roles.

11.5 Vulnerability Development

Often, vulnerabilities were explored and developed as a proof of concept within a small dummy application that did nothing more than expose the vulnerability, and only later integrated into a realistic application.

In many cases, an application was designed simply to showcase a particular vulnerability. For example, the concept of the BattleBoats application (section 6.9) was developed as an opportunity to use Newton's method (for computing the flight time of cannon shots.) Similarly, PowerBroker (section 6.7) and BidPal (section 6.6) were designed to showcase a timing side channel in a secret bidding scheme. To a lesser extent, the CyberWallet application (section 6.13) was developed due to its potential for harboring lots of realistically secret data. Less often, vulnerabilities were tailored to an existing application; for example, the recursive answer vulnerability took advantage of data persistence implemented with a visitor pattern to cause infinite recursion in SimpleVote (section 6.11). Other more general vulnerabilities, such as those in cryptography, hash tables, and sorting were readily placed in several different applications.

11.6 Coding Standard

Coding standards can produce readable and understandable code. However, this can also create an additional burden on the developer, distracting from the actual task of developing code. To minimize this, basic standards based on the Google Java style guide [44] were utilized. The requirements for strict coding practices were relaxed, since the code developed for the STAC project does not have to live a long life or be maintained by a large team.

11.7 Testing

Testing is a way to verify that code written works as expected. Testing can be generally divided into unit tests and integration tests. Unit testing verifies the behavior of individual parts of code and also provides a way to verify that edge-cases are handled properly. There are several well-established unit testing frameworks available for Java: JUnit, TestNG, Spock (most preferred due to testing capabilities and ease; however, it requires writing tests in Groovy).

Integration testing verifies that components, when built into a system, behave as expected; testing end to-end. Testing of this type typically takes longer and often involves more resources than unit testing. Of particular note are the tests that verify the existence of intended vulnerabilities in an application by exploiting them and validating the output, and similarly the tests that verify that a vulnerability has been successfully mitigated. Additionally, several fuzzers were implemented to bombard certain applications with input, in order to hopefully stumble onto any unintended algorithmic complexity vulnerabilities.

For the STAC project, unit tests were written using JUnit, and Nose was used for executing the integration tests since they are written using Python. Additionally, all tests of the upcoming engagement were performed nightly on the reference platform to verify all tests were successful on the project architecture.

11.8 Building the Project

Gradle is the tool used to build the project. The same build commands are used by the developer and the continuous build system. All of the tasks needed to build the system are defined in the build script. Due to the particular needs of our project, the build system is far more complex than that of a typical software project. See section 5 for further details.

11.9 Continuous Integration

A continuous integration system, Jenkins, was used to ensure that the challenge programs and tests build and run as expected on the reference platform. Whenever Jenkins builds the system, the latest source from the Git repository is pulled, the source code is built, and then both unit and integration tests are executed. Jenkins is scheduled to automatically build and run the entire system nightly.

Additionally, whenever a pull request is merged into the main Git repository, Jenkins is triggered automatically to start a full build/test. This can also be manually triggered by any developer. In addition to ensuring the main repository builds correctly, there is a nightly build that ensures the latest release candidate still builds/runs correctly. Custom Jenkins tasks also allow a developer to test their own branch on the reference platform. This is helpful to ensure that all tests run successfully prior to creating a pull request.

11.10 Staged Delivery

While the main repository is the location where the most current development is maintained, there is also a need to track previous deliveries. The Git model provides tags and branches to accomplish this.

The most recent release candidate is maintained (and built nightly) and can be pulled on demand. When a new release candidate is ready, the same process for code approval is followed: A pull request is created requesting that the main repository be merged into the current release.

Likewise, once a current release is identified as ready for delivery, a Jenkins task tags the release candidate, builds the full system, and pushes a copy of the binaries to a local delivery Git repository. The local delivery Git repository is then copied to the remote STAC delivery Git repository.

11.11 Documentation

It is important to document all development discussions, design decisions, communications, engagement results, and team information. This provides a document trail but, more importantly, provides knowledge of the project structure, background, purpose, and goals useful to educate new team members. We used the Confluence collaboration platform from Atlassian to document the project.

11.12 Team Communication

While documentation is critical for the project, there is also a need for daily communication between team members. Slack was used to provide team-wide and member-to-member communication, allowing brief messages to be exchanged and also allowing snippets of code or text to be easily shared. In addition, Slack is not limited to team members, providing the advantage to add web-hooks for the build system to send notifications on success/failure of continuous integration builds, as well as pull requests which can be sent to the entire team.

12 Scala vs. Java Bytecode

12.1. Java Bytecode for Code Snippet 10

```
public class HelloJ
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #12.#25        // java/lang/Object."<init>":()V
   #2 = Fieldref           #26.#27        //
java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #28            // Starting program!
   #4 = Methodref          #29.#30        //
java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Double             1.0000000005E10d
   #7 = Methodref          #31.#32        // java/lang/Math.abs:(D)D
   #8 = Long               10000l
  #10 = Methodref          #33.#34        // java/lang/Thread.sleep:(J)V
  #11 = Class              #35            // HelloJ
  #12 = Class              #36            // java/lang/Object
  #13 = Utf8               <init>
  #14 = Utf8               ()V
  #15 = Utf8               Code
  #16 = Utf8               LineNumberTable
  #17 = Utf8               main
  #18 = Utf8               ([Ljava/lang/String;)V
  #19 = Utf8               StackMapTable
  #20 = Class              #37            // "[Ljava/lang/String;"
  #21 = Utf8               Exceptions
  #22 = Class              #38            // java/lang/Exception
  #23 = Utf8               SourceFile
  #24 = Utf8               HelloJ.java
  #25 = NameAndType        #13:#14        // "<init>":()V
  #26 = Class              #39            // java/lang/System
  #27 = NameAndType        #40:#41        // out:Ljava/io/PrintStream;
  #28 = Utf8               Starting program!
  #29 = Class              #42            // java/io/PrintStream
  #30 = NameAndType        #43:#44        // println:(Ljava/lang/String;)V
  #31 = Class              #45            // java/lang/Math
  #32 = NameAndType        #46:#47        // abs:(D)D
  #33 = Class              #48            // java/lang/Thread
  #34 = NameAndType        #49:#50        // sleep:(J)V
  #35 = Utf8               HelloJ
  #36 = Utf8               java/lang/Object
  #37 = Utf8               [Ljava/lang/String;
  #38 = Utf8               java/lang/Exception
  #39 = Utf8               java/lang/System
  #40 = Utf8               out
  #41 = Utf8               Ljava/io/PrintStream;
  #42 = Utf8               java/io/PrintStream
  #43 = Utf8               println
  #44 = Utf8               (Ljava/lang/String;)V
  #45 = Utf8               java/lang/Math
  #46 = Utf8               abs
```

```
  #47 = Utf8               (D)D
  #48 = Utf8               java/lang/Thread
  #49 = Utf8               sleep
  #50 = Utf8               (J)V
{
  public HelloJ();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
        line 1: 0
  public static void main(java.lang.String[]) throws java.lang.Exception;
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=4, locals=9, args_size=1
         0: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3                  // String Starting program!
         5: invokevirtual #4                  // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
         8: bipush        10
        10: istore_1
        11: ldc2_w        #5                  // double 1.0000000005E10d
        14: dstore_2
        15: dconst_0
        16: dstore        4
        18: iconst_0
        19: istore        6
        21: iload         6
        23: iload_1
        24: if_icmpge     39
        27: dload         4
        29: dload_2
        30: dadd
        31: dstore        4
        33: iinc          6, 1
        36: goto          21
        39: dload         4
        41: iload_1
        42: i2d
        43: ddiv
        44: dstore        7
        46: dload_2
        47: dload         7
        49: dsub
        50: invokestatic  #7                  // Method
java/lang/Math.abs:(D)D
        53: d2l
        54: lconst_0
```

```
        55: lcmp
        56: ifeq          65
        59: ldc2_w        #8                  // long 10000l
        62: invokestatic  #10                 // Method
java/lang/Thread.sleep:(J)V
        65: return
      LineNumberTable:
        line 4: 0
        line 6: 8
        line 7: 11
        line 8: 15
        line 10: 18
        line 11: 27
        line 10: 33
        line 13: 39
        line 14: 46
        line 15: 59
        line 18: 65
      StackMapTable: number_of_entries = 3
        frame_type = 255 /* full_frame */
          offset_delta = 21
          locals = [ class "[Ljava/lang/String;", int, double, double, int ]
          stack = []
        frame_type = 17 /* same */
        frame_type = 252 /* append */
          offset_delta = 25
          locals = [ double ]
    Exceptions:
      throws java.lang.Exception
}
SourceFile: "HelloJ.java"
```

12.2. Scala Bytecode for Code Snippet 11

```
public final class HelloS$$anonfun$main$1 extends
scala.runtime.AbstractFunction1$mcVI$sp implements scala.Serializable
  minor version: 0
  major version: 50
  flags: ACC_PUBLIC, ACC_FINAL, ACC_SUPER
Constant pool:
   #1 = Utf8               HelloS$$anonfun$main$1
   #2 = Class              #1             // HelloS$$anonfun$main$1
   #3 = Utf8               scala/runtime/AbstractFunction1$mcVI$sp
   #4 = Class              #3             //
scala/runtime/AbstractFunction1$mcVI$sp
   #5 = Utf8               scala/Serializable
   #6 = Class              #5             // scala/Serializable
   #7 = Utf8               HelloS.scala
   #8 = Utf8               HelloS$
   #9 = Class              #8             // HelloS$
  #10 = Utf8               main
  #11 = Utf8               ([Ljava/lang/String;)V
  #12 = NameAndType        #10:#11        // main:([Ljava/lang/String;)V
  #13 = Utf8               serialVersionUID
  #14 = Utf8               J
  #15 = Long               0l
```

```
  #17 = Utf8               N$1
  #18 = Utf8               Lscala/runtime/DoubleRef;
  #19 = Utf8               z$1
  #20 = Utf8               apply
  #21 = Utf8               (I)V
  #22 = Utf8               apply$mcVI$sp
  #23 = NameAndType        #22:#21        // apply$mcVI$sp:(I)V
  #24 = Methodref          #2.#23         //
HelloS$$anonfun$main$1.apply$mcVI$sp:(I)V
  #25 = Utf8               this
  #26 = Utf8               LHelloS$$anonfun$main$1;
  #27 = Utf8               i
  #28 = Utf8               I
  #29 = NameAndType        #19:#18        // z$1:Lscala/runtime/DoubleRef;
  #30 = Fieldref           #2.#29         //
HelloS$$anonfun$main$1.z$1:Lscala/runtime/DoubleRef;
  #31 = Utf8               scala/runtime/DoubleRef
  #32 = Class              #31            // scala/runtime/DoubleRef
  #33 = Utf8               elem
  #34 = Utf8               D
  #35 = NameAndType        #33:#34        // elem:D
  #36 = Fieldref           #32.#35        // scala/runtime/DoubleRef.elem:D
  #37 = NameAndType        #17:#18        // N$1:Lscala/runtime/DoubleRef;
  #38 = Fieldref           #2.#37         //
HelloS$$anonfun$main$1.N$1:Lscala/runtime/DoubleRef;
  #39 = Utf8               (Ljava/lang/Object;)Ljava/lang/Object;
  #40 = Utf8               scala/runtime/BoxesRunTime
  #41 = Class              #40            // scala/runtime/BoxesRunTime
  #42 = Utf8               unboxToInt
  #43 = Utf8               (Ljava/lang/Object;)I
  #44 = NameAndType        #42:#43        // unboxToInt:(Ljava/lang/Object;)I
  #45 = Methodref          #41.#44        //
scala/runtime/BoxesRunTime.unboxToInt:(Ljava/lang/Object;)I
  #46 = NameAndType        #20:#21        // apply:(I)V
  #47 = Methodref          #2.#46         //
HelloS$$anonfun$main$1.apply:(I)V
  #48 = Utf8               scala/runtime/BoxedUnit
  #49 = Class              #48            // scala/runtime/BoxedUnit
  #50 = Utf8               UNIT
  #51 = Utf8               Lscala/runtime/BoxedUnit;
  #52 = NameAndType        #50:#51        // UNIT:Lscala/runtime/BoxedUnit;
  #53 = Fieldref           #49.#52        //
scala/runtime/BoxedUnit.UNIT:Lscala/runtime/BoxedUnit;
  #54 = Utf8               v1
  #55 = Utf8               Ljava/lang/Object;
  #56 = Utf8               <init>
  #57 = Utf8
(Lscala/runtime/DoubleRef;Lscala/runtime/DoubleRef;)V
  #58 = Utf8               ()V
  #59 = NameAndType        #56:#58        // "<init>":()V
  #60 = Methodref          #4.#59         //
scala/runtime/AbstractFunction1$mcVI$sp."<init>":()V
  #61 = Utf8               ConstantValue
  #62 = Utf8               Code
  #63 = Utf8               LocalVariableTable
```

```
  #64 = Utf8               LineNumberTable
  #65 = Utf8               SourceFile
  #66 = Utf8               EnclosingMethod
  #67 = Utf8               InnerClasses
  #68 = Utf8               Scala
{
  public static final long serialVersionUID;
    descriptor: J
    flags: ACC_PUBLIC, ACC_STATIC, ACC_FINAL
    ConstantValue: long 0l
  public final void apply(int);
    descriptor: (I)V
    flags: ACC_PUBLIC, ACC_FINAL
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: iload_1
         2: invokevirtual #24                 // Method apply$mcVI$sp:(I)V
         5: return
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       6     0  this   LHelloS$$anonfun$main$1;
            0       6     1     i   I
      LineNumberTable:
        line 10: 0
  public void apply$mcVI$sp(int);
    descriptor: (I)V
    flags: ACC_PUBLIC
    Code:
      stack=5, locals=2, args_size=2
         0: aload_0
         1: getfield      #30                 // Field
z$1:Lscala/runtime/DoubleRef;
         4: aload_0
         5: getfield      #30                 // Field
z$1:Lscala/runtime/DoubleRef;
         8: getfield      #36                 // Field
scala/runtime/DoubleRef.elem:D
        11: aload_0
        12: getfield      #38                 // Field
N$1:Lscala/runtime/DoubleRef;
        15: getfield      #36                 // Field
scala/runtime/DoubleRef.elem:D
        18: dadd
        19: putfield      #36                 // Field
scala/runtime/DoubleRef.elem:D
        22: return
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      23     0  this   LHelloS$$anonfun$main$1;
            0      23     1     i   I
      LineNumberTable:
        line 11: 0
  public final java.lang.Object apply(java.lang.Object);
    descriptor: (Ljava/lang/Object;)Ljava/lang/Object;
```

```
    flags: ACC_PUBLIC, ACC_FINAL, ACC_BRIDGE, ACC_SYNTHETIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: aload_1
         2: invokestatic  #45                 // Method
scala/runtime/BoxesRunTime.unboxToInt:(Ljava/lang/Object;)I
         5: invokevirtual #47                 // Method apply:(I)V
         8: getstatic     #53                 // Field
scala/runtime/BoxedUnit.UNIT:Lscala/runtime/BoxedUnit;
        11: areturn
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      12     0  this   LHelloS$$anonfun$main$1;
            0      12     1    v1   Ljava/lang/Object;
      LineNumberTable:
        line 10: 0
  public HelloS$$anonfun$main$1(scala.runtime.DoubleRef,
scala.runtime.DoubleRef);
    descriptor: (Lscala/runtime/DoubleRef;Lscala/runtime/DoubleRef;)V
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=3
         0: aload_0
         1: aload_1
         2: putfield      #38                 // Field
N$1:Lscala/runtime/DoubleRef;
         5: aload_0
         6: aload_2
         7: putfield      #30                 // Field
z$1:Lscala/runtime/DoubleRef;
        10: aload_0
        11: invokespecial #60                 // Method
scala/runtime/AbstractFunction1$mcVI$sp."<init>":()V
        14: return
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      15     0  this   LHelloS$$anonfun$main$1;
            0      15     1   N$1   Lscala/runtime/DoubleRef;
            0      15     2   z$1   Lscala/runtime/DoubleRef;
      LineNumberTable:
        line 10: 0
}
SourceFile: "HelloS.scala"
EnclosingMethod: #9.#12                 // HelloS$.main
InnerClasses:
     public final #2; //class HelloS$$anonfun$main$1
Error: unknown attribute
  Scala: length = 0x0
```

13 References

[1] Public release items for the DARPA Space/Time Analysis for Cybersecurity (STAC) program. Web page. https://github.com/Apogee-Research/STAC.

[2] E. Milanov. The RSA algorithm, 2003. https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf.

[3] STAC canonical examples public release. Web page. https://github.com/Apogee-Research/STAC/tree/master/Canonical_Examples.

[4] Visitor pattern. Web page. https://en.wikipedia.org/wiki/Visitor_pattern.

[5] Gradle build tool. Web page. https://gradle.org.

[6] Apache Groovy. Web page. http://groovy-lang.org/.

[7] Riva Borbely et al. CASCAID User's Guide, 2019.

[8] Jinja. Web page. http://jinja.pocoo.org/.

[9] JavaParser. Web page. http://javaparser.org.

[10] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In POPL '09 Proceedings of 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 127–139. ACM, 2009.

[11] IntelliJ platform SDK devguide. Web page. https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html.

[12] Line breaking. Web page. https://xxyxyz.org/line-breaking.

[13] Diffie-Hellman key exchange. Web page. https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange.

[14] Exponentiation by squaring. Web page. https://en.wikipedia.org/wiki/Exponentiation_by_squaring#Montgomery%27s_ladder_technique.

[15] Thomas H. Cormen, Charles E. Leiserson, et al. Introduction to Algorithms, chapter Getting Started. MIT Press, 1989.

[16] Thomas H. Cormen, Charles E. Leiserson, et al. Introduction to Algorithms, chapter Quicksort. MIT Press, 1989.

[17] Thomas H. Cormen, Charles E. Leiserson, et al. Introduction to Algorithms, chapter Maximum Flow. MIT Press, 1989.

[18] Thomas H. Cormen, Charles E. Leiserson, et al. Introduction to Algorithms, chapter Single-Source Shortest Paths. MIT Press, 1989.

[19] Alberto Martelli. On the complexity of admissible search algorithms. Artificial Intelligence, 8(1):1–13, 1997.

[20] Billion laughs attack. Web page. https://en.wikipedia.org/wiki/Billion_laughs_attack.

[21] Werner Schindler. A timing attack against RSA with the Chinese Remainder Theorem. In International Workshop on Cryptographic Hardware and Embedded Systems, pages 109–124. Springer, 2000.

[22] Timothy D. Morgan and Jason W. Morgan. Web timing attacks made practical. In Black Hat USA 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Morgan-Web-Timing-Attacks-Made-Practical-wp.pdf.

[23] Battleship. Web page. https://en.wikipedia.org/wiki/Battleship_(game).

[24] Newton's method. Web page. https://en.wikipedia.org/wiki/Newton%27s_method.

[25] Secant method. Web page. https://en.wikipedia.org/wiki/Secant_method.

[26] Patrick Dehornoy. A fast method for comparing braids. Advances in Mathematics, 125.2:200–235, 1997.

[27] J Rizzo and T Duong. The CRIME attack. In ekoparty Security Conference, 2012. https://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf.

[28] Yoel Gluck, Neal Harris, and Angelo Prado. BREACH: reviving the CRIME attack. In Black Hat USA 2013. https://media.blackhat.com/us-13/US-13-Prado-SSL-Gone-in-30-seconds-A-BREACH-beyond-CRIME-WP.pdf.

[29] Savan Oswal, Anjali Singh, and Kirthi Kumari. DEFLATE compression algorithm. International Journal of Engineering Research and General Science, 4(1):430–436, 2016.

[30] Linear Feedback Shift Register. Web page. https://en.wikipedia.org/wiki/Linear-feedback_shift_register.

[31] Solomon W. Golomb et al. Shift register sequences. Aegean Park Press, 1967.

[32] Primitive polynomial (field theory). Web page. https://en.wikipedia.org/wiki/Primitive_polynomial_(field_theory).

[33] Primitive polynomial list. Web page. http://www.partow.net/programming/polynomials/index.html#deg12.

14 Acronyms

Acronym	Description
AC	algorithmic complexity
AC-Disk	algorithmic complexity vulnerability in disk usage
AC Memory	algorithmic complexity vulnerability in memory usage
AC-T	algorithmic complexity vulnerability in time
CASCAID	Complexity and Side-Channel Adversarial Integrated Defects
BSSID	Basic Service Set Identifier
BAA	Broad Agency Announcement
CBC	cipher block chaining
FFDH	Finite-Field Diffie-Hellman
FFT	Fast Fourier Transform
GCD	greatest common divisor
JIT	Just-in-Time
LFSR	linear feedback shift register
NUC	Next Unit of Computing
SC	side channel
SC-S	side channel in space
SC-ST	side channel in space and time
SC-T	side channel in time
STAC	Space/Time Analysis for Cybersecurity
TLS	Transport Layer Security