SIGPwny

General FA2025 • 2025-11-02

Mayhem

Ronan Boyarski, Nikhil Date

Announcements

- We are playing **BuckeyeCTF 2025** hosted by OSU!
    - This **Friday (11/7)** at **7:00pm** (room TBD, likely Siebel 2406)
    - Unlike most CTFs, Buckeye offers prizes to the top 3 undergraduate teams
    - No graduate students are allowed to play
    - Try hard and win that prize!

Finding bugs in binaries

- Traditionally, you would do this by hand
- Start by using a debugger and hook the main way you write input (usually socket **`recv()`** function)
- Sync your debugger with your disassembler and check the control flow graph to find calls to unsafe functions
    - For example, looking for calls to **`memcpy()`** where you can specify the wrong input size
    - Can be subtle, like finding multiple chained memcpy calls where size is fixed but you can get some fixed overflow into a dangerous region
- Reliable, but slow and depends on a skilled reverse engineer
- How much of this can we automate?

Fuzzing

- A sort of randomized testing designed to find memory bugs
- Fuzzer tries to select inputs that will explore as much of the program as possible
- The idea is that, the more of the program we explore, the higher likelihood we have of discovering a bug
- Fuzzing is very successful in practice
  - Example fuzzers include: AFL, libfuzzer, syzkaller

"More bugs than eyes. Setup Syzkaller on a junk pc tonight, by Sunday you will have unique, likely exploitable kernel bugs" -Ravi

Symbolic Execution

- Normal execution is known as "concrete" execution
- Symbolic execution: make the program inputs symbolic values
- Track symbolic expressions representing program state
- Use this to look at all possible paths
- Using symbolic execution, we can solve for whether we can mess with important bits of program state (like the instruction pointer)
- Problems:
  - Path explosion: symbolic execution does not scale well with big programs
  - Environment: how are you tracking heap allocations, syscalls, etc.
  - Reasoning about memory: how do you track memory accesses, the most important part of our manual bug finding method?

Problem Setting

- We have a binary we want to exploit
    - Could be a really fun target, like Windows SMB server, iOS messaging app, or kernel driver
- Said binary has no debugging information, and is large and annoying
- We would like to automate finding bugs in *all* binaries of that form
    - For example, try to find exploitable bug in **every** kernel driver
- We want to automatically generate patches for all of our exploits
- We want to make sure that what the machine gives us is reliable and verifiable (no false positives > no false negatives)

MAYHEM

- 2016 DARPA Cyber Grand Challenge winner, developed by CMU
- "Every bug reported by MAYHEM is accompanied by a working shell-spawning exploit"
- "To make exploit generation possible at the binary-level, MAYHEM addresses two major technical challenges: actively managing execution paths without exhausting memory, and reasoning about *symbolic memory indices*, where a load or a store address depends on user input. To this end, we propose two novel techniques: 1) hybrid symbolic execution for combining online and offline (concolic) execution to maximize the benefits of both techniques, and 2) index-based memory modeling."

High-Level Idea

- What if we combine concrete execution with symbolic execution, so we can reason about real program state using symbolic execution?
- Model a formula for whether we can control the instruction pointer, put attacker supplied data in memory, and have a memory protection primitive to execute said code
  - If symbolic execution returns SAT, we can exploit the program
- How do they handle path explosion?
  - No repeated work
  - No ballooning higher than the current machine's RAM at any one time
  - Reason about symbolic memory where a load or store address depends on user input

Hybrid Symbolic Execution

- Execution alternates between online and offline symbolic runs
- Acts like a virtual memory manager in an OS, where we swap out symbolic execution engines
  - When we are overburdened, we swap out the current symbolic state and cache it, like a swapfile for virtual memory
  - Caching formulas prevents re-execution, ensuring forward progress
- Concretization constraints prevent false positives
- Pure symbolic execution is hard
  - For example, memcmp creates a new branch for each character in the input buffer
  - No need to do symbolic execution for the setup before we are opening the program to any attacker input

Bug Example

```
#define BUFSIZE 4096
typedef struct {
	char buf[BUFSIZE];
	int used;
} STATIC_BUFFER_t;
typedef struct conn {
	STATIC_BUFFER_t read_buf;
	… // irrelevant
} CONN_t;
static void serverlog(LOG_TYPE_t type, const char* format, …)
{
	… // irrelevant
	if (format) {
		va_start (ap, format);
		vsprintf(buf, format, ap);
		va_end(ap);
	}
	fprintf(log, buf); // format string bug
	fflush(log);
}
```

```
HTTP_STATE_t http_read_request(CONN_t *conn) {
     … // irrelevant
     while (conn->read_buf.used < BUFSIZE) {
       sz = static_buffer_read(conn, &conn->read_buf);
       if (sz < 0) {
         …
         conn->read_buf.used += sz;
         if (memcmp(&conn->read-buf.buf[conn->read_buf.used]
              - 4, "\r\n\r\n", 4) == 0 { break; }
       }
     if (conn->read_buf.used >= BUFSIZE) {
       conn->status.st = HTTP_STATUS_400;
       return HTTP_STATE_ERROR;
     }
     serverlog(ERROR_LOG, "%s\n", conn->read_buf.buf);
} // serverlog user input comes from the outside (HTTP)
```

How it works

- There's a concrete runner (client) and symbolic executor (server)
- Client has a taint tracker to see if user input does bad things
  - If so, pass it to the symbolic executor, and see if we can solve to do even worse things
  - Works by checking if any basic blocks are corrupted
- Server decides tasking for the client and where to explore
- So, taint tracker would see that we are calling `fprintf` with a user-specified string, which lets us control how we access stack memory
  - At this point, we would suspend concrete execution, and determine where we can branch to
  - Pathing to new places is modeled as additional constraints (one per jump)

How it works

- When we hit any important parts of the program, the SES will try to build a solution to its symbolic equation
  - If it can find one, then any solution is, by necessity, a working exploit
  - It will crawl through each of these possibilities by context switching back to the client
- MAYHEM will continue to explore until either a solution is returned, we hit a user-specified maximum runtime, or all execution paths have been exhausted

Results

- 29 exploitable vulnerabilities (2 0-days)
- Relatively limited bug classes (buffer overflow, format string, function pointer overwrite)
- Won in the 2016 DARPA Cyber Grand Challenge, although those binaries were relatively simplified compared to hardened real-world targets
- Darpa had the AIxCC challenge this year, which was like the 2016 Cyber Grand Challenge but using LLMs on top of these techniques

Discussion

- Do you think symbolic execution can handle finding more complex bug classes, like heap exploits and logic bugs? How would you go about modeling a heap allocator or important logical state?
- Are there other new techniques that would help amplify the effectiveness of fuzzing or symbolic execution, or otherwise win back some of the intuition and creativity of a human exploit developer? LLMs have been surprisingly middling thus far…
- Which do you think moves faster, automated bug finding, or exploitation mitigation techniques? How would a modern redesign of this concept fare against modern security mitigations?

Next Meetings

2025-11-06 • This Thursday

- Game Hacking
- Go write some cheats and finally learn how Windows works

2025-11-09 • Next Sunday

- Movie Social
- We have a movie in mind but we're keeping it secret 🙂

2025-11-13 • Next Thursday

- Rubber Ducky / Bad USB
- Turn physical access into RCE with this one simple trick