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!

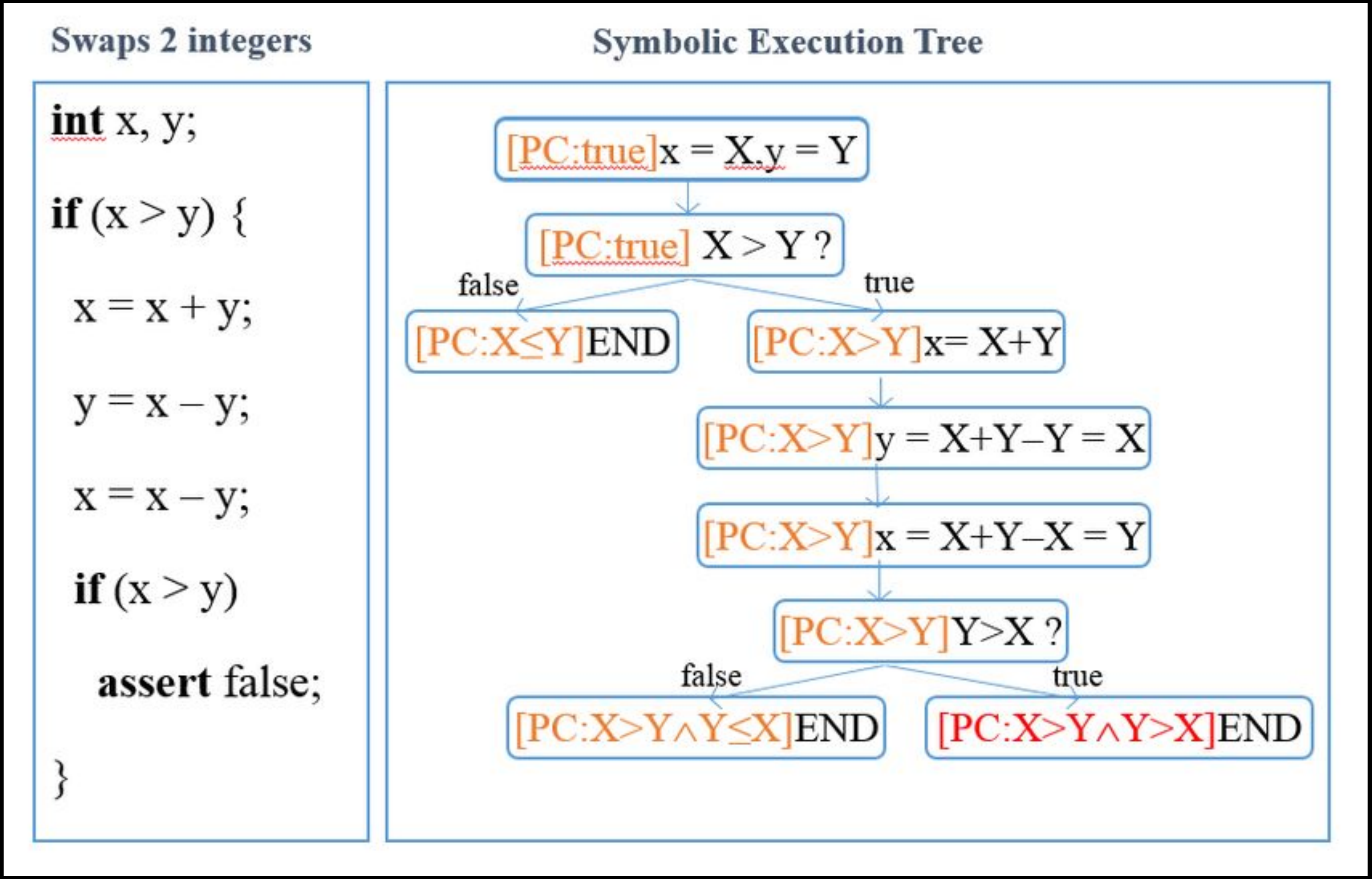
Background: Manual bug-finding

- 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?

Background: Symbolic Execution

- Concrete execution: run program with concrete inputs
- Symbolic execution: run program with symbolic inputs
    - "Emulate" effects of each instruction on program state
- Program state contains symbolic values
    - For binary-only analysis program state = registers + memory
- Have to look at all possible paths
    - For each program state, track "path condition" that led to that path
- Why do we care?
    - "Exhaustively" explores the state space of the program, can then solve for conditions implying bugs/vulnerabilities
    - Is my instruction pointer a symbolic value that can be affected by the input?
- Learn more: https://www.youtube.com/watch?v=yRVZPvHYHzw

Background: Symbolic Execution


Swaps 2 integers

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

Symbolic Execution Tree

Symbolic Execution Challenges

- State explosion
  - In a complex program, number of paths/states can blow up very quickly
  - What if you have loops?
- Modeling the Environment
  - How to model library calls (e.g. malloc), system calls, etc.
- Symbolic memory
  - Let's say I read/write memory with a symbolic address
  - What to do?
  - Memory reasoning is essential because many bugs are related to memory corruption

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
- No debugging information, large and annoying (hard to manually exploit)
- Want to automatically find exploitable bugs and generate exploits
- This means we don’t want false positives
- Ideally we don’t want false negatives either :)

Unleashing MAYHEM on Binary Code

- Symbolic execution system for automatically finding vulnerabilities with proof-of-vulnerability (exploit)
- Shows that symbolic execution is practical for finding bugs and generating exploits on real-world programs
  - "Hybrid symbolic execution" to address state explosion issues
  - "Index-based memory modeling" to address memory modeling issues
- "Every bug reported by MAYHEM is accompanied by a working shell-spawning exploit"
- Modified? version of MAYHEM system won the 2016 DARPA Cyber Grand Challenge (CGC)

MAYHEM System Design

- Usage on orzhttpd HTTP server
  - mayhem -sym-net 80 400 ./orzhttpd

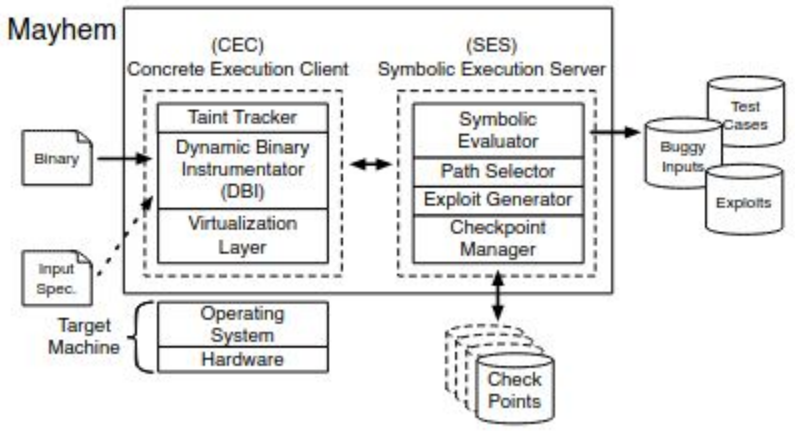


Figure 2: MAYHEM architecture

“Online” vs “Offline” Execution

- Online execution forks the symbolic executor at each branch
- Offline explores a single path at a time and reruns the symbolic executor from the beginning on each path
- What are the tradeoffs?

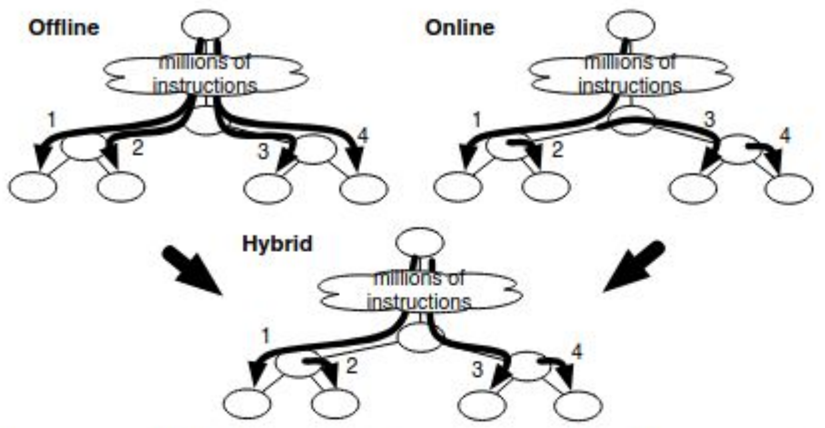


Figure 3: Hybrid execution tries to combine the speed of online execution and the memory use of offline execution to efficiently explore the input space.

Hybrid Execution

- Tries to strike a middle ground between online and offline execution
- Execute online until we hit memory cap
- Then save “checkpoints” for all active executor states
  - “Checkpoints” throw out the concrete state and so compress the memory usage over an active symbolic executor state
- Restart online execution from one of these checkpoints (chosen according to certain heuristics)

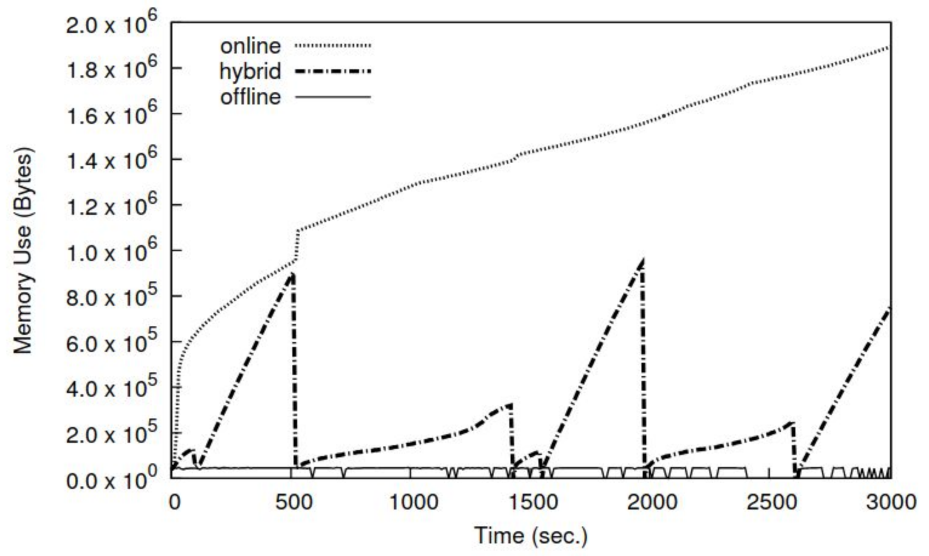


Figure 7: Memory use in online, offline, and hybrid mode.

CEC + SES Design

- CEC = “Concrete Executor Client”
  - natively executes program fragments with purely concrete state
  - performs taint analysis to detect symbolic values
  - sends tainted blocks to SES for symbolic execution
- SES = “Symbolic Executor Server”
  - symbolically executes tainted blocks and sends paths to be executed to CEC according to path prioritization heuristics
- Tries to optimize over purely symbolic emulation of all instructions
- Path prioritization
  - Prioritize paths with more coverage a la coverage-guided fuzzing
  - Prioritize paths with symbolic instruction pointer

Index-Based Memory Modeling

- One approach to model memory: concretize every memory access
  - If I access M[i] where i is symbolic, just pick some concrete value that satisfies the symbolic expression
  - What are the tradeoffs?
- “More than 40% of the programs required symbolic memory modeling”
- Another approach: fully symbolic memory
  - What are the tradeoffs?
- MAYHEM approach: writes are concretized, reads are not
  - load at address i generates a memory object M that contains all values i could refer to
  - But try to do some optimizations that speed up reads
  - Try to put some bounds on a pointer (“value set analysis”)
  - See paper for gory details

Bug Example

```c
#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);
}
```

```c
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

- 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

- Is it reasonable to care about no false positives so much, as MAYHEM does?
- Can we extend the MAYHEM idea to 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?
- Is there a better way to model memory than MAYHEM's approach?
- Do you see any problems with this approach?
- Do you have any ideas to make this approach better?

Discussion

- 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