Lec10: Fuzzing and Symbolic Execution

Taesoo Kim

Administrivia

- In-class CTF on Dec 4 (24 hours)!
- Submit your team's challenge by Nov 20
- But submit it early for our feedback!
- Or you can brainstorm your challenge during the office hours!

Emphasis on Exploitation (so far)

- In practice, it's more important to ask: how to find bugs?
  - With source code
  - With only binaries

Two Pre-conditions for Exploitation

1. Locating a bug (i.e., bug finding)
2. Triggering the bug (i.e., reachability)

```
1   // Q2. How to reach this path?
2   if (magic == 0xdeadbeef) {
3       // Q1. Is this buggy?
4       memcpy(dst, src, len)
5   }
```

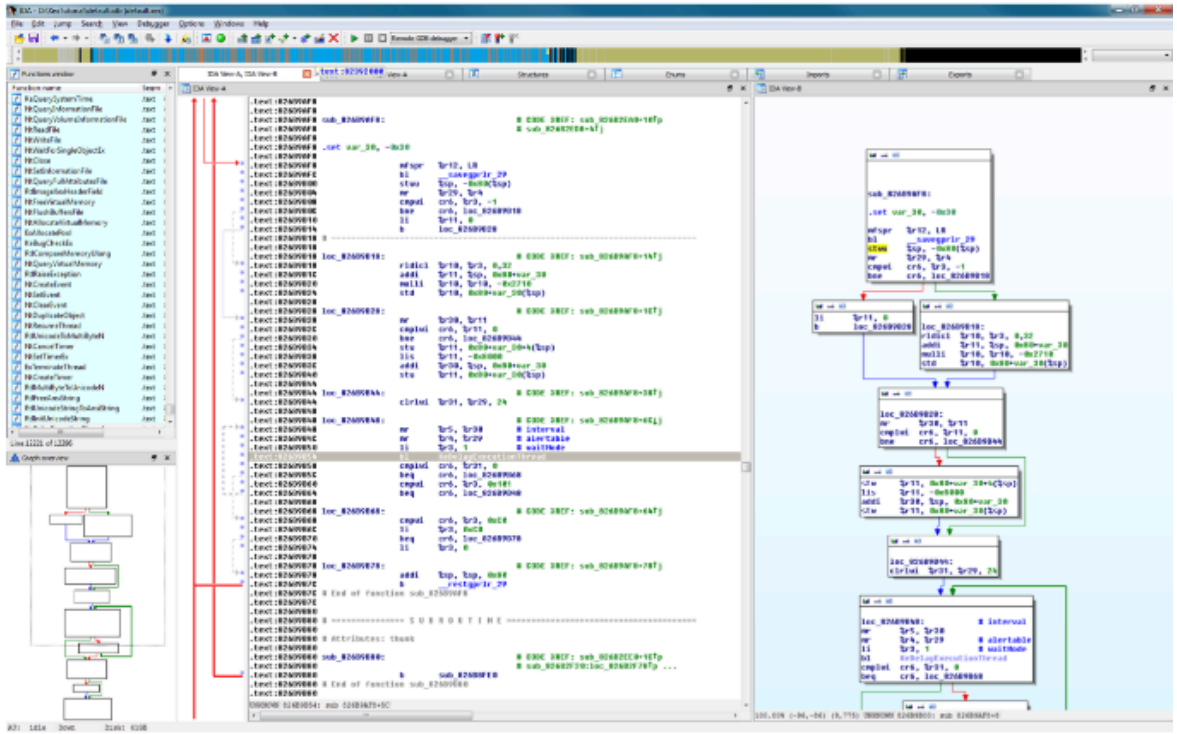
Solution 1: Code Auditing (w/ code)


```
1  static OSStatus SSLVerifySignedServerKeyExchange(...) {
2   ...
3   if (err = SSLHashSHA1.update(&hashCtx, &clientRandom))
4       goto fail;
5   if (err = SSLHashSHA1.update(&hashCtx, &serverRandom))
6       goto fail;
7   if (err = SSLHashSHA1.update(&hashCtx, &signedParams))
8       goto fail;
9       goto fail;
10  if (err = SSLHashSHA1.final(&hashCtx, &hashOut)
11      goto fail;
12
13  err = sslRawVerify(...);
14 fail:
15     return err;
16 }
```

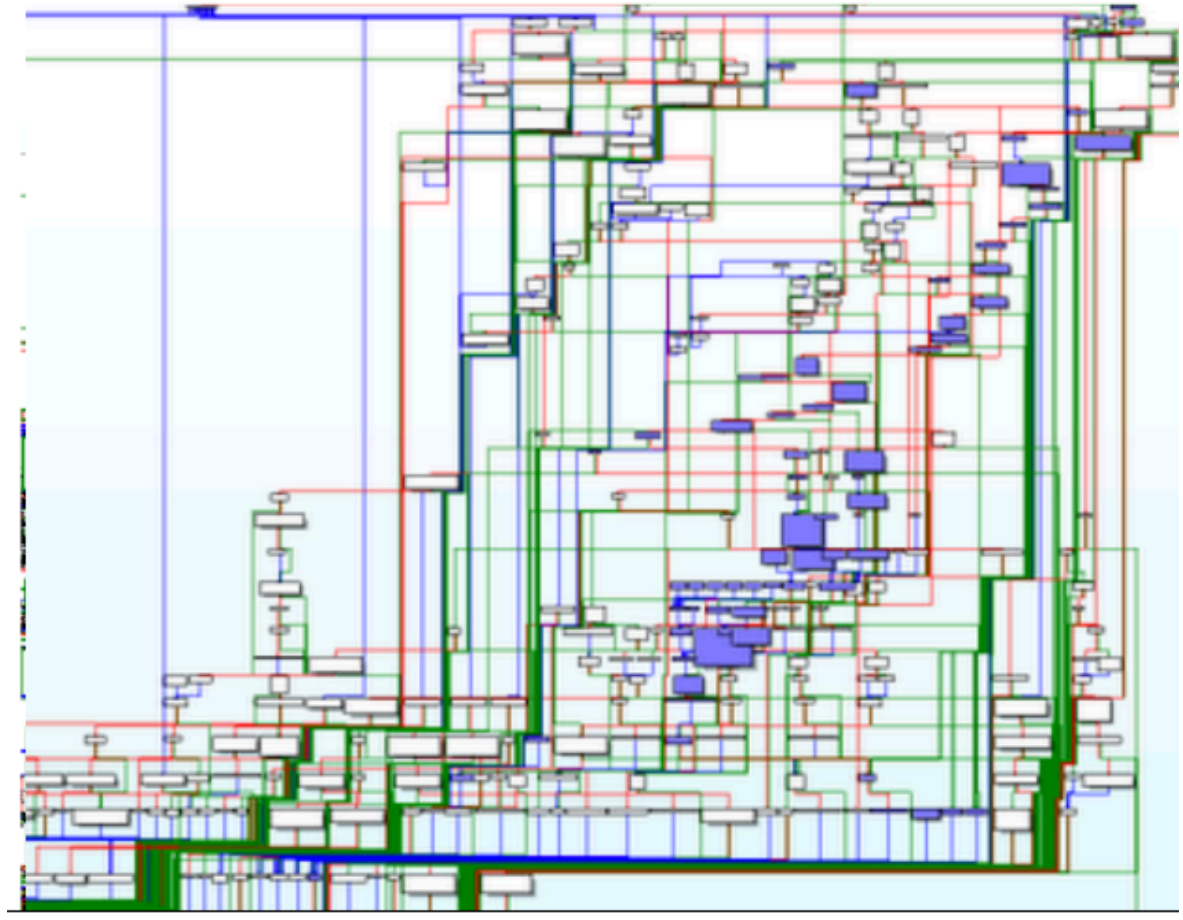
Ref. CVE-2014-1266, Apple sslKeyExchange.c

Solution 2: Static Analysis (on binary)

- Reverse Engineering (e.g., IDA or Ghidra)



Problem: Too Complex (e.g., browse)



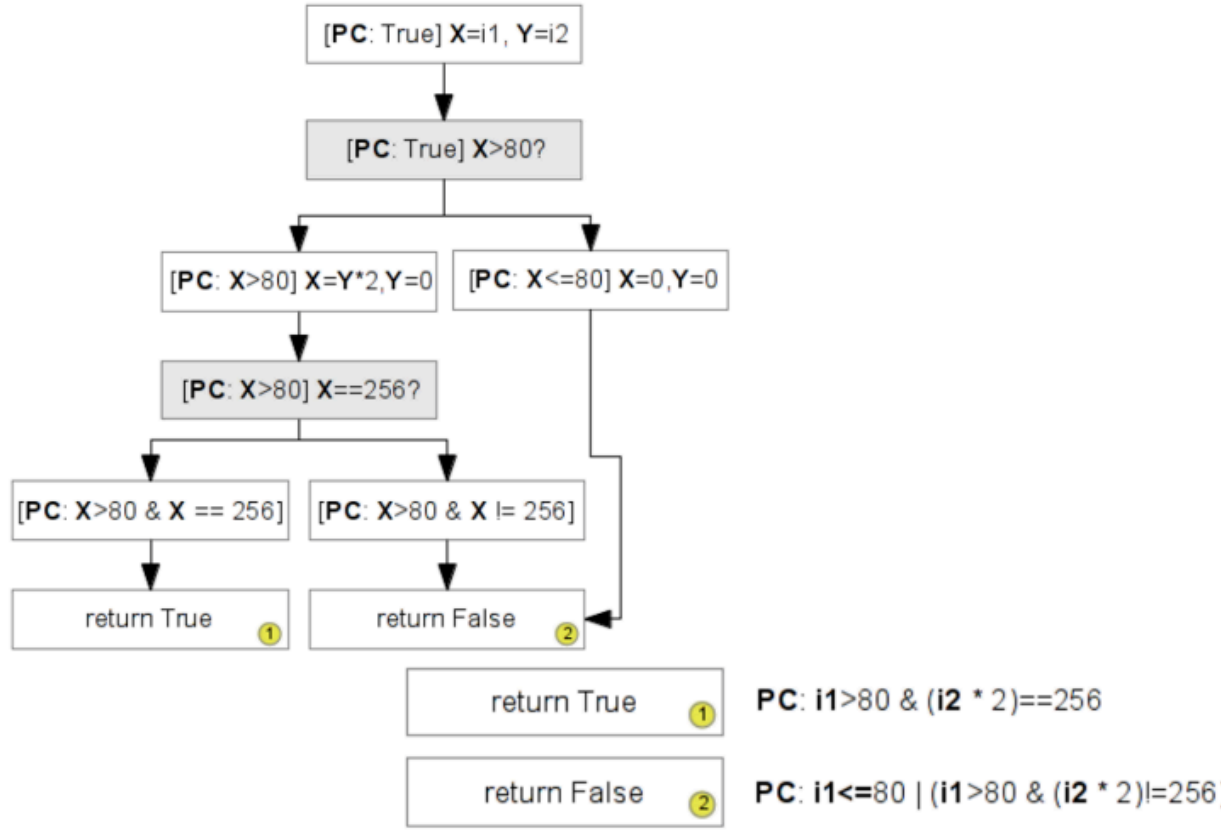
Two Popular Directions

- Symbolic execution (static)
- Fuzzing (dynamic)

Symbolic Execution


```
int foo(int i1, int i2)
{
    int x = i1;
    int y = i2;

    if (x > 80){
        x = y * 2;
        y = 0;
        if (x == 256)
            return True;
    }
    else{
        x = 0;
        y = 0;
    }
    /* ... */
    return False;
}
```



Problem: State Explosion

- Too many path to explore (e.g., strcmp(input1, input2))
- Too huge input/state space (e.g., browser? OS?)
- Solving constraints is a hard problem (i.e., slow)



Today's Topic: Fuzzing

- Key ideas
  1. Execute the program with pseudo-random inputs (i.e., input corpus)
  2. Check if the input crashes (i.e., crashing input)
  3. Observe the program execution (i.e., code coverage)
  4. Make a better guess for the input (i.e., input mutation)

→ Reachability is free since we are executing with a concrete input!

How well fuzzing can explore all paths?

```
1  int foo(int i1, int i2) {
2    int x = i1;
3    int y = i2;
4
5    if (x > 80) {
6      x = y * 2;
7      y = 0;
8      if (x == 256) {
9  *     __builtin_trap();
10       return 1;
11     }
12   } else {
13     x = 0; y = 0;
14   }
15   return 0;
16 }
```

Let’s Compare Two Approaches

- To crash this example, both need to meet `i1 > 80` and `i2 == 128`:
  - Symbolic execution needs to resolve just two conditions (sounds easy)
  - Fuzzing needs to scan an entire **int** to find a proper `i2` (`i1` is easy)
- However, LibFuzzer/AFL/AFLplusplus are much faster thanks to numerous heuristics!
- Also, testing one fuzzing input is x10k faster!

→ Q. what about `strcmp(buf, "250382")` in `crackme0x00`?

Importance of High-quality Corpus

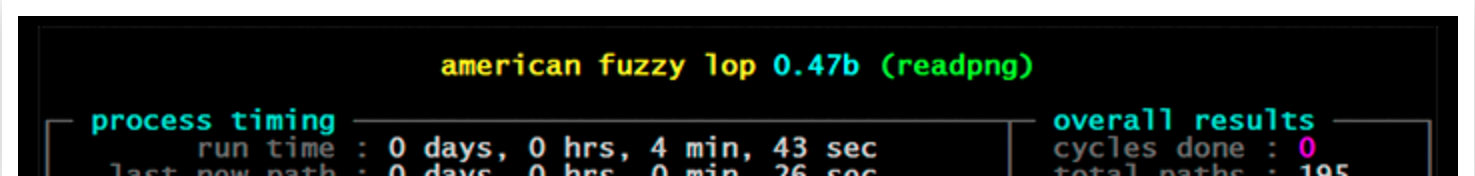
- In fact, fuzzing is really bad at exploring paths
  - e.g., if (a == 0xdeadbeef)
- So, paths should be (or mostly) given by corpus (sample inputs)
  - e.g., pdf files utilizing full features
  - but, not too many! (do not compromise your performance)
- A fuzzer will trigger the exploitable state
  - e.g., len in malloc()

AFL (American Fuzzy Lop)

- VERY well-engineered fuzzer w/ lots of heuristics

american fuzzy lop (2.53b)

American fuzzy lop is a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary. This substantially improves the functional coverage for the fuzzed code. The compact synthesized corpora produced by the tool are also useful for seeding other, more labor- or resource-intensive testing regimes down the road.



Examples of Mutation Techniques

- interest: -1, 0x8000000, 0xffff, etc
- bitflip: flipping 1,2,3,4,8,16,32 bits
- havoc: random tweak in fixed length
- extra: dictionary, etc
- etc

Idea 1: Map Input to State Transition

- Input → [IPs] (problem?)

Idea 1: Map Input to State Transition

- Input → [IPs] (problem?)
- Input → map[blockIPs % len] (problem? A→B vs B→A)

Idea 1: Map Input to State Transition

- Input → [IPs] (problem?)
- Input → map[blockIPs % len] (problem? A→B vs B→A)
- Input → map[((prevBlockIP >> 1) ^ curBlockIP) % len] (problem?)

Idea 1: Map Input to State Transition

- Input → [IPs] (problem?)
- Input → map[blockIPs % len] (problem? A→B vs B→A)
- Input → map[((prevBlockIP >> 1) ^ curBlockIP) % len] (problem?)
- Input → map[((rand1 >> 1) ^ rand2) % len] (problem?)

Idea 1: Map Input to State Transition

- Input → [IPs] (problem?)
- Input → map[blockIPs % len] (problem? A→B vs B→A)
- Input → map[((prevBlockIP >> 1) ^ curBlockIP) % len] (problem?)
- Input → map[((rand1 >> 1) ^ rand2) % len] (problem?)

→ It also normalize (or bucketize) the map to handle loops better!

Idea 2: Avoiding Redundant Paths

- If you see the duplicated state, throw out
  - e.g., i1 = 1, 2, 3
- If you see the new path, keep it for further exploration
  - e.g., i1 = 81

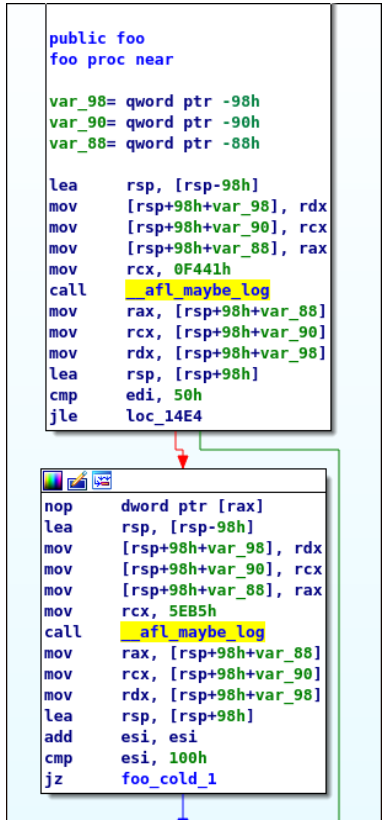
How to Create Mapping?

- Instrumentation
  - Source code → by compiler (e.g., gcc, clang)
  - Binary → via binary instrumentation (e.g., QEMU)

```
1  if (block_address > elf_text_start
2      && block_address < elf_text_end) {
3    cur_location = (block_address >> 4) ^ (block_address << 8);
4    shared_mem[cur_location ^ prev_location] ++;
5    prev_location = cur_location >> 1;
6  }
```

Source Code Instrumentation

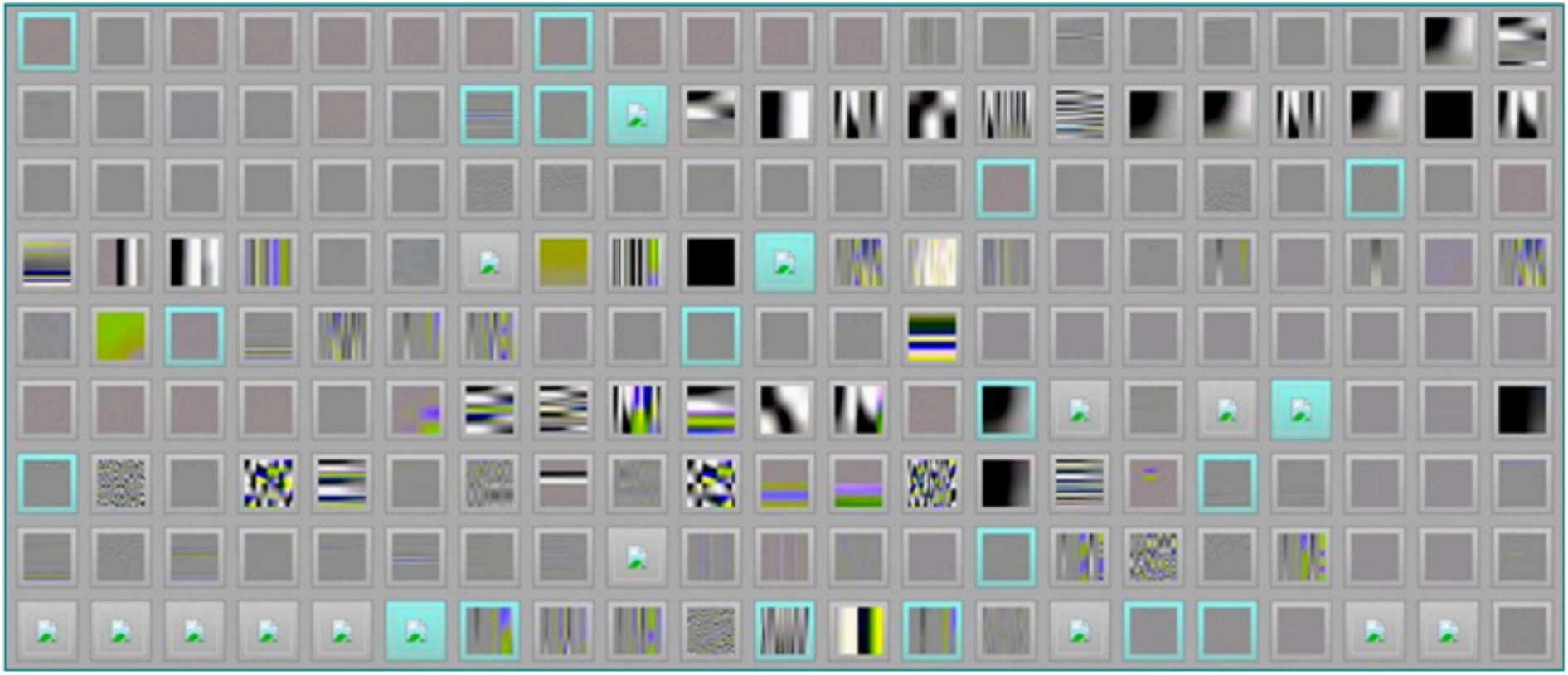
```
1436: mov   rcx,0xf441
143d: call  1520 <__afl_maybe_log>
```



afl_maybe_log()

```
0000000000001440 <__afl_store>:
  # rcx = __afl_cur_loc
  xor    rcx,QWORD PTR [rip+0x2c51]   # rcx = __afl_cur_loc ^ __afl_prev_loc
  xor    QWORD PTR [rip+0x2c4a],rcx   # __afl_prev_loc = __afl_cur_loc
  shr    QWORD PTR [rip+0x2c43],1     # __afl_prev_loc = __afl_prev_loc >> 1
  add    BYTE PTR [rdx+rcx*1],0x1     # __afl_area_ptr[rcx] += 1
```

AFL Arts



Ref. http://lcamtuf.coredump.cx/afl/

Problem: Not All Bugs Crash

```
1  char *buf = malloc(10);
2  buf[10] = 'A';    // 1-byte heap overflow: no crash!
3  free(buf);
4  buf[0] = 'B';     // use-after-free: no crash!
```

- Fuzzers only “see” crashes, so silent memory corruption is missed
- Sanitizers: compiler instruments every load/store with a check
  - e.g., AddressSanitizer (ASan): `clang -fsanitize=address`

→ Q. How can we make every invalid access crash?

Recall: Stack Canaries?

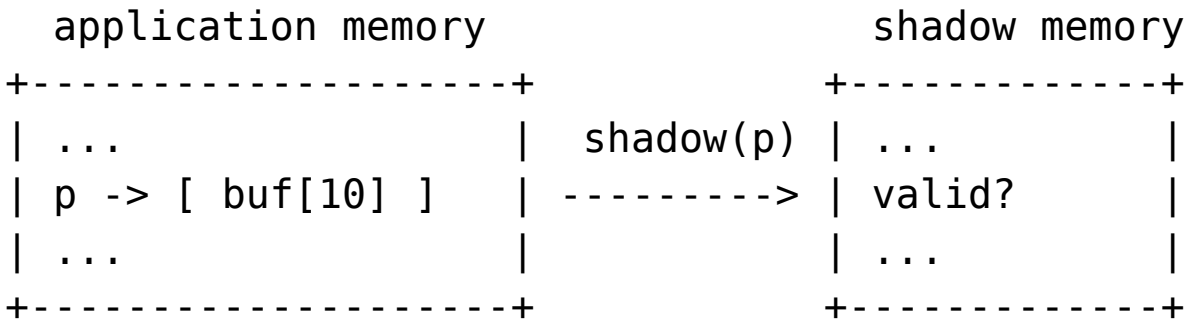
```
1  char buf[16] = "guest";   // global
2  char cmd[16] = "ls";      // global
3
4  int main() {
5    gets(buf);   // overflow into cmd: no canary at all!
6    system(cmd);
7  }
```

- Only protect one region (i.e., the return address)
- Cannot detect overflows into global objects
- Cannot detect use-after-free (or any heap bugs)

→ Q. Can we protect every memory access?

Idea: Shadow Memory

- Keep metadata (“is it valid?”) for memory in a separate region



- Compiler inserts a check before every memory access

```
1 *p = x;   // becomes:
2 if (!valid(shadow(p))) report(p);   // crash on invalid access!
3 *p = x;
```

How to Track Valid Memory?

- Track each object (e.g., a list of [start, end)) (problem?)

How to Track Valid Memory?

- Track each object (e.g., a list of [start, end)) (problem?)
- Track each byte w/ 1 shadow byte (problem?)

How to Track Valid Memory?

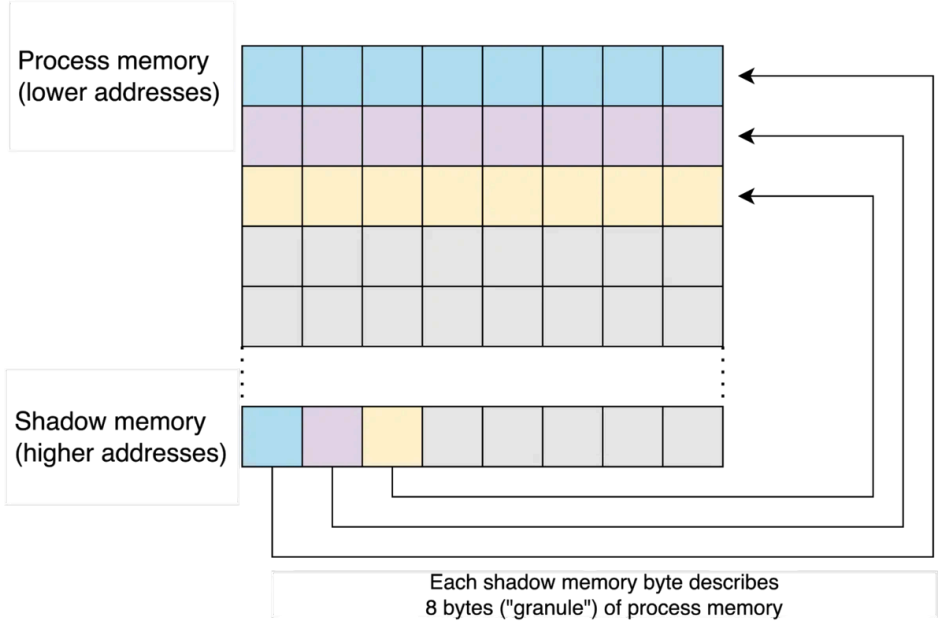
- Track each object (e.g., a list of [start, end)) (problem?)
- Track each byte w/ 1 shadow byte (problem?)
- Track each byte w/ 1 shadow bit (problem?)

How to Track Valid Memory?

- Track each object (e.g., a list of [start, end)) (problem?)
- Track each byte w/ 1 shadow byte (problem?)
- Track each byte w/ 1 shadow bit (problem?)
- Track an 8-byte range w/ 1 shadow byte (problem?)

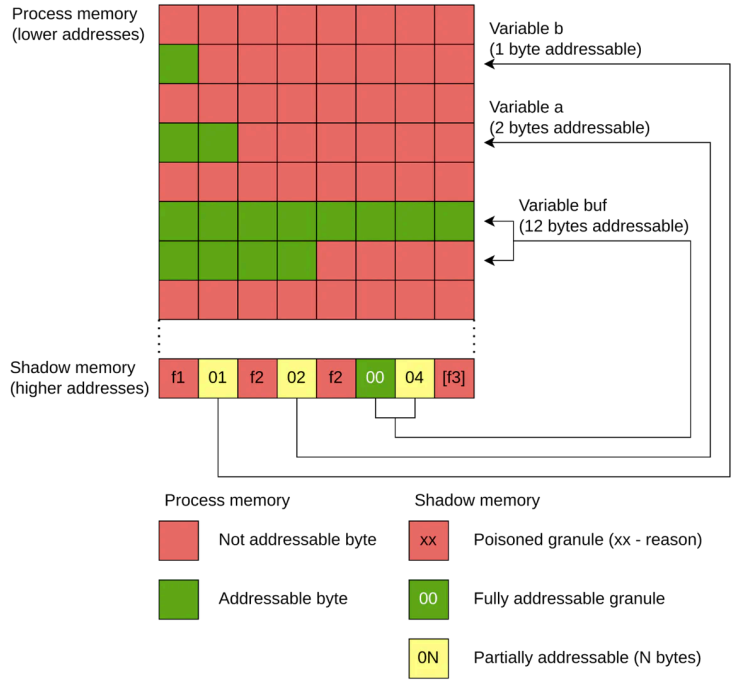
ASan: Shadow Memory

- 1 shadow byte describes an 8-byte granule: `shadow = (addr >> 3) + 0x7fff8000`
- `malloc()` is 8-byte aligned, so only the first k bytes can be addressable



ASan: Redzones

- Poison redzones around every object (e.g., `f1` - `f3` for stack, `fa` for heap)



Ref. Trail of Bits: Understanding AddressSanitizer

ASan: Shadow Check

```
memory:  [AAAAAAAA][AA......][........]  // buf = malloc(10)
shadow:  [   00   ][   02   ][   fa   ]
```

```
1  // instrumented before every N-byte load/store (N = 1, 2, 4)
2  int8_t k = *(int8_t *)((addr >> 3) + 0x7fff8000);
3  if (k != 0 && (int8_t)((addr & 7) + N - 1) >= k)
4    __asan_report_load(addr);   // buf[10]: 2 + 0 >= 2 -> report!
```

- Use-after-free: poison freed memory (`fd`) and delay its reuse (quarantine)

→ ~2x slowdown, ~3x memory: great for fuzzing, not for production!

References

- CVE-2014-1266, Apple sslKeyExchange.c
- LibFuzzer
- AFL (American Fuzzy Lop)
- AFLplusplus
- Trail of Bits: Understanding AddressSanitizer
- Sanitize, Fuzz, and Harden Your C++ Code