CSE 484: Computer Security and Privacy

Software Security (Misc)

Winter 2021

David Kohlbrenner

dkohlbre@cs.washington.edu

Thanks to Franzi Roesner, Dan Boneh, Dieter Gollmann, Dan Halperin, Yoshi Kohno, Ada Lerner, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Admin

- Lab 1 checkpoint *next Wednesday night!*
  - That is, sploits 1-3
  - When you are ‘done’ you stop changing those files.

```
chmod -w sploit0.c
      +w
```

Last Words on Buffer Overflows…

Defenses

- ASLR – Randomize where the stack/heap/code starts
  - Counters: Information disclosures, sprays and sleds
- Canaries – Put a value on the stack, see if it changes
  - Counters: Arbitrary writes
- DEP – Mark sections of memory as non-executable, e.g. the stack
  - Counters: ROP, JOP, Code-reuse attacks in general, JIT

W ⊕ X | write xor execute

Defense: Shadow stacks

- Idea: don't store return addresses on the stack!

- Store them on… a different stack! returns go here
  - A *hidden* stack

- On function call/return instrument the function
  - Store/retrieve the return address from shadow stack

- Maybe encrypt/randomize the shadow stack data?

Challenges With Shadow Stacks

- Where do we put the shadow stack?
  - Can the attacker figure out where it is?
- How fast is it to store/retrieve from the shadow stack?

adds 2-4 stores + loads

- How *big* is the shadow stack?

is it a full stack? just rets?

- Is this compatible with all software?

Other Possible Solutions

- Use safe programming languages, e.g., Rust (or Java?)
  - What about legacy C code?
  - (Though Rust doesn't magically fix all security issues ☺)
- Static analysis of source code to find overflows
- Dynamic testing: "fuzzing"

remove entire vuln classes

Other Common Software Security Issues…

Another Type of Vulnerability

- Consider this code:

```
char buf[80];
void vulnerable() {
     int len = read_int_from_network();
     char *p = read_string_from_network();
     if (len > sizeof buf) {
          error("length too large, nice try!");
          return;
     }
     memcpy(buf, p, len);
}
```

```
void *memcpy(void *dst, const void * src, size_t n);
```

```
typedef unsigned int size_t;
```

Another Example

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

Breakout Groups: January 15th on Canvas

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Implicit Cast

int signed 32b
unsigned 32b

- Consider this code:

```
char buf[80];
void vulnerable() {
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if (len > sizeof buf) {
        error("length too large, nice try!");
        return;
    }
    memcpy(buf, p, len);
}
```

If len is negative, may copy huge amounts of input into buf.

-INT_MAX

int > int(80

size_t

ssize_t

UNSIGNED_MAX

```
void *memcpy(void *dst, const void * src, size_t n);
```

```
typedef unsigned int size_t;
```

Integer Overflow

if max-5 > len 5 < 0

```
size_t len = read_int_from_network();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

4

if ((len+5) < len)

error!

- What if len is large (e.g., len = 0xFFFFFFFF)? +5
- Then len + 5 = 4 (on many platforms)
- Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

(from www-inst.eecs.berkeley.edu—implflaws.pdf)

Another Type of Vulnerability

- Consider this code:

Check

```
if (access("file", W_OK) != 0) {
   exit(1); // user not allowed to write to file
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- **Goal:** Write to file only with permission
- What can go wrong?

TOCTOU (Race Condition)

confused deputy
(sort of)

- TOCTOU = “Time of Check to Ti~~l~~me of Use”

```
if (access(“file”, W_OK) != 0) {
   exit(1); // user not allowed to write to file
}

fd = open(“file”, O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- **Goal:** Write to file only with permission
- Attacker (in another program) can change meaning of “file” between access and open:

```
symlink("/etc/passwd", "file");
```

file myfile.txt
check
rm myfile
ln myfile.txt

Password Checker

- Functional requirements
  - PwdCheck(RealPwd, CandidatePwd) should:
    - Return TRUE if RealPwd matches CandidatePwd
    - Return FALSE otherwise
  - RealPwd and CandidatePwd are both 8 characters long

Password Checker

- Functional requirements
  - PwdCheck(RealPwd, CandidatePwd) should:
    - Return TRUE if RealPwd matches CandidatePwd
    - Return FALSE otherwise
  - RealPwd and CandidatePwd are both 8 characters long
- Implementation (like TENEX system)

```
PwdCheck(RealPwd, CandidatePwd)   // both 8 chars
    for i = 1 to 8 do
        if (RealPwd[i] != CandidatePwd[i]) then
            return FALSE
    return TRUE
```

- Clearly meets functional description

Attacker Model

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars
   for i = 1 to 8 do
      if (RealPwd[i] != CandidatePwd[i]) then
         return FALSE
   return TRUE
```

- Attacker can guess CandidatePwds through some standard interface
- Naive: Try all 256^8 = 18,446,744,073,709,551,616 possibilities

a a a a a a a a F - 3ms

b a a a a a a a F - 3ms

c a a a a a a a F - 4ms

4 4

c b a a a a a a 5m

Timing Attacks

subset of side-channels

- Assume there are no "typical" bugs in the software
  - No buffer overflow bugs ←
  - No format string vulnerabilities ←
  - Good choice of randomness ←
  - Good design ←
- The software may still be vulnerable to timing attacks
  - Software exhibits input-dependent timings
- Complex and hard to fully protect against

Other Examples

- Plenty of other examples of timings attacks
  - Timing cache misses
    - Extract cryptographic keys…
    - Recent Spectre/Meltdown attacks
  - Duration of a rendering operation
    - Extract webpage information
  - Duration of a *failed* decryption attempt
    - Different failures mean different thing (e.g. Padding oracles)

in cache 4ns

*ptr

not in cache 100s

privatecontent.com/$(username)

Side-channels

- Timing is only one possibility

- Consider:
  - Power usage —
  - Sensors — accelerometers
  - EM Outputs — TEMPEST

Software Security: So what do we do?

Fuzz Testing

- Generate “random” inputs to program
  - Sometimes conforming to input structures (file formats, etc.)
- See if program crashes
  - If crashes, found a bug
  - Bug may be exploitable
- Surprisingly effective

- Now standard part of development lifecycle

General Principles

- Check inputs ← injections, checking lengths
- Check all return values ← malloc
- Least privilege
- Securely clear memory (passwords, keys, etc.) ← memset(∅)
- Failsafe defaults
- Defense in depth
  - Also: prevent, detect, respond

- NOT: security through obscurity

General Principles

- Reduce size of trusted computing base (TCB) – least priv.
- Simplicity, modularity
  - But: Be careful at interface boundaries! – almost all bugs
- Minimize attack surface
- Use vetted components
- Security by design
  - But: tension between security and other goals
- Open design? Open source? Closed source?
  - Different perspectives

Does Open Source Help?

- Different perspectives…

- Happy example?
  - Linux kernel backdoor attempt thwarted (2003)
    (http://www.freedom-to-tinker.com/?p=472)

- Sad example?
  - Heartbleed (2014)
    - Vulnerability in OpenSSL that allowed attackers to read arbitrary memory from vulnerable servers (including private keys)

Vulnerability Analysis and Disclosure

- What do you do if you've found a security problem in a real system?
- Say
  - A commercial website?
  - UW grade database?
  - Boeing 787?
  - TSA procedures?