CSE 484: Computer Security and Privacy

Software Security: A few more defenses and attacks

Spring 2024

David Kohlbrenner
dkohlbre@cs

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

• Lab 1a due Wednesday
  • Lab1a writeups are individual, and are a textbox on gradescope, rather than a pdf upload.

• We do update the SSH guide and such if there are common challenges
  • Take a look at that and the lab FAQs if you run into problems first
Printf exploitation explanation not clicking?

• I’ve uploaded two short exercises for starting to write printf exploits to ed
  • Give them a try if you are a bit lost, or even if you aren’t 😊
return-to-libc

• Overwrite saved ret (IP) with address of any library routine
  • Arrange stack to look like arguments

• Does not look like a huge threat
  • ...
  • We can call *any* function we want!
  • Say, exec 😊
return-to-libc++

• Insight: Overwritten saved EIP need not point to the *beginning* of a library routine

• **Any** existing instruction in the code image is fine
  • Will execute the sequence starting from this instruction

• What if instruction sequence contains RET?
  • Execution will be transferred... to where?
  • Read the word pointed to by stack pointer (SP)
    • Guess what? Its value is under attacker’s control!
  • Use it as the new value for IP
    • Now control is transferred to an address of attacker’s choice!
  • Increment SP to point to the next word on the stack
Chaining RETs

• Can chain together sequences ending in RET
  • Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)

• What is this good for?

• Answer [Shacham et al.]: everything
  • Turing-complete language
  • Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  • Attack can perform arbitrary computation using no injected code at all – return-oriented programming

• Truly, a “weird machine”
Return-Oriented Programming
**Defense: Run-Time Checking: StackGuard**

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary
Defense: Run-Time Checking: StackGuard

- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary

- Choose random canary string on program start
  - Attacker can’t guess what the value of canary will be

- Canary contains: “\0”, newline, linefeed, EOF
  - String functions like strcpy won’t copy beyond “\0”
StackGuard Implementation

• StackGuard requires code recompilation

• Checking canary integrity prior to every function return causes a performance penalty
  • For example, 8% for Apache Web server at one point in time
Defeating StackGuard

- StackGuard can be defeated
  - A single memory write where the attacker controls both the value and the destination is sufficient

- Suppose program contains `copy(buf,attacker-input)` and `copy(dst,buf)`
  - Example: `dst` is a local pointer variable
  - Attacker controls both `buf` and `dst`
Defenses so far

• **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
Pointer integrity protections (e.g. PointGuard, PAC, etc.)

• Attack: overwrite a pointer (heap date, ret, function pointer, etc.)

• Idea: encrypt all pointers while in memory
  • Generate a random key when program is executed
  • Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    • Pointers cannot be overflowed while in registers

• Attacker cannot predict the target program’s key
  • Even if pointer is overwritten, after XORing with key it will dereference to a “random” memory address
Normal Pointer Dereference

**CPU**
1. Fetch pointer value
2. Access data referenced by pointer

**Memory**
- Pointer 0x1234
- Data

**CPU**
1. Fetch pointer value
2. Access attack code referenced by corrupted pointer

**Memory**
- Corrupted pointer 0x1234
- 0x1340
- Data
- Attack code

0x1234
0x1340

[Cowan]
4/5/2024
PointGuard Dereference

1. Fetch pointer value
2. Access data referenced by pointer

1. Fetch pointer value
2. Access random address; segmentation fault and crash

CPU

Memory

Encrypted pointer 0x7239

Data

0x1234

Memory

Corrupted pointer 0x7239 0x1340

Data

Attack code

0x1234 0x1340 0x9786
PointGuard Issues

• Must be very fast
  • Pointer dereferences are very common

• Compiler issues
  • Must encrypt and decrypt only pointers
  • If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there

• Attacker should not be able to modify the key
  • Store key in its own non-writable memory page

• PG’d code doesn’t mix well with normal code
  • What if PG’d code needs to pass a pointer to OS kernel?
Defense: Shadow stacks

• Idea: don’t store return addresses on the stack!

• Store them on... a different stack!
  • A hidden stack

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

• Or store on both main stack and shadow stack, and compare for equality at function return

• 2020/2021 Hardware Support emerges (e.g., Intel Tiger Lake, AMD Ryzen PRO 5000)
Challenges With Shadow Stacks

• Where do we put the shadow stack?
  • Can the attacker figure out where it is? Can they access it?

• How fast is it to store/retrieve from the shadow stack?

• How big is the shadow stack?

• Is this compatible with all software?

• (Still need to consider data corruption attacks, even if attacker can’t influence control flow.)
What does a modern program do?

(Mostly normal x86_32)

080491f6 <foo>:
80491f6:   f3 0f 1e fb  endbr32
80491fa:   55                    push %ebp
80491fb:   89 e5                   mov %esp,%ebp
80491fd:   81 ec c0 01 00 00       sub $0x1c0,%esp
8049203:   8b 45 08                mov 0x8(%ebp),%eax
8049206:   89 85 40 fe ff ff       mov %eax,-0x1c0(%ebp)
804920c:   65 a1 14 00 00 00       mov %gs:0x14,%eax
8049212:   89 45 08                mov 0x8(%,ebp),%eax
8049215:   80 45 fc                xor %eax,%eax
8049217:   80 85 40 fe ff ff       mov -0x1c0(%ebp),%eax
804921d:   65 a1 14 00 00 00       mov %gs:0x14,%eax
8049222:   50                    push %eax
8049223:   8d 85 44 fe ff ff       lea -0x1bc(%ebp),%eax
8049229:   50                    push %eax
804922a:   e8 81 00 00 00         call 80490b0 <strcpy@plt>
804922f:   83 c4 08               add $0x8,%esp
8049232:   90                    nop
8049233:   8b 55 fc                mov -0x4(%ebp),%edx
8049236:   65 33 15 14 00 00 00   xor %gs:0x14,%edx
804923d:   74 05                   je 8049244 <foo+0x4e>
804923f:   e8 4c fe ff ff         call 8049090 <__stack_chk_fail@plt>
8049244:   c9                    leave
8049245:   c3                    ret
Other Big Classes of Defenses

• Use safe programming languages, e.g., Java, Rust
  • What about legacy C code?
  • (Though Java doesn’t magically fix all security issues 😊)
• Static analysis of source code to find overflows
• Dynamic testing: “fuzzing”
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
Other Common Software Security Issues...
Another Class of Vulnerability: (Gradescope)

```c
char buf[80];
void vulnerable() {
    long long len = get_int_from_user();
    char *p = get_string_from_user();
    int32_t buflen = sizeof buf;
    if (len > buflen) {
        error("length too large");
        return;
    }
    memcpy(buf, p, len);
}
```

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

Snippet 1

Snippet 2
Implicit Cast

• Consider this code (x86_32bit):

```c
char buf[80];
void vulnerable() {
    long long len = read_int_from_network();
    char *p = read_string_from_network();
    int32_t buflen = sizeof buf;
    if (len > buflen) {
        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;
```

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

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

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

(from www-inst.eecs.berkeley.edu—implflaws.pdf)
Another Type of Vulnerability

• Consider this code:

```c
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)

- **TOCTOU** = “Time of Check to Tile of Use”

```c
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`:
  ```c
  symlink("/etc/passwd", "file");
  ```
Something Different: Password Checker

• Functional requirements
  • $\text{PwdCheck}(\text{RealPwd, CandidatePwd})$ 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)

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

• Clearly meets functional description
Attacker Model

- Attacker can guess CandidatePwds through some standard interface
- Naive: Try all $256^8 = 18,446,744,073,709,551,616$ possibilities
- Is it possible to derive password more quickly?

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

dkohlbre.com/cew