Warmup:
Logistics

- Lab 1 due Friday
  - If you are having any problems, please read the SSH guide and instructions closely!
- In general, post _text_ not screenshots of text for questions on ed
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)

- Attack: overflow a function pointer so that it points to attack code
- 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

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

CPU

Memory

Pointer
0x1234

Data
0x1234

CPU

Memory

Corrupted pointer
0x1234
0x1340

Data
0x1234

Attack code
0x1340

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

[Cowan]
PointGuard Dereference

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

Memory

CPU

Decrypted
0x1234

Encrypted
0x7239

0x1234

0x1340

Data

1. Fetch pointer value

Decrypted
0x1340

Corrupted pointer
0x7239

0x1340

Data

0x1234

0x9786

Attack code

2. Access random address; segmentation fault and crash
What might be a challenge of adding pointguard (or generally a pointer-encryption scheme) to code?

Consider how it would work with libraries, the operating system, etc.
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
80491fb:  89 e5
80491fd:  81 ec c0 01 00 00 00
8049203:  8b 45 08
8049206:  89 85 40 fe ff ff
8049208:  65 a1 14 00 00 00
8049212:  89 45 fc
8049215:  31 c0
8049217:  8b 85 40 fe ff ff
804921d:  83 c0 04
8049222:  50
8049223:  8d 85 44 fe ff ff
8049229:  50
804922a:  e8 81 fe ff ff
804922f:  83 c4 08
8049232:  90
8049233:  8b 55 fc
8049236:  65 33 15 14 00 00 00
804923d:  74 05
804923f:  e8 4c fe ff ff
8049244:  c9
8049245:  c3

(Lab 1 version)
08049196 <foo>:
8049196:  55
8049197:  89 e5
8049199:  81 ec b8 01 00 00 00
804919f:  8b 45 08
80491a2:  83 c0 04
80491a5:  8b 00
80491a7:  50
80491a8:  8d 85 48 fe ff ff
80491ae:  50
80491af:  e8 9c fc fe ff ff
80491b4:  83 c4 08
80491b7:  90
80491b8:  c9
80491b9:  c3
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...
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");
        return;
    }
    memcpy(buf, p, len);
}

void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;

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

• Consider this code:

```c
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`.

```c
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```
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](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:
  symlink("/etc/passwd", "file");
Something Different: Password Checker

• Functional requirements
  • \texttt{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
  • $\text{PwdCheck}(\text{RealPwd}, \text{CandidatePwd})$ should:
    • Return TRUE if RealPwd matches CandidatePwd
    • Return FALSE otherwise
  • RealPwd and CandidatePwd are both 8 characters long

• Implementation (like TENEX system)

```plaintext
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?

```python
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