Today’s Class

1. **Memory Safety Attacks:**
   How can attackers exploit software bugs to force a program to run code or commands they want?

2. **Memory Safety Protections:**
   How can we prevent these kinds of software attacks or minimize the damage they can do?
Outline: Memory Safety: Attacks & Defenses

1. Review: Memory layout and function calls in a process

2. Attacks:
   1. Stack-based buffer overflow attacks
   2. Heap vulnerabilities (briefly)

3. Defenses:
   1. Stack Canaries
   2. Address-Space Layout Randomization (ASLR)
   3. W^X and ROP
   4. Fuzzing and Memory Safe Languages
The Stack and Calling a Function in C

What happens to memory when you call \texttt{foo(a,b)}?
- A “stack frame” is added (\texttt{esp} & \texttt{ebp} move up)
- Instruction pointer \texttt{eip} moves to code for \texttt{foo}

```c
int foo(int a, int b) {
    int d = 1;
    return a+b+d;
}

int main(...) {
    ... int x = foo(5, 6);
    ...
}
```

```
Virtual Memory

<table>
<thead>
<tr>
<th>Stack</th>
<th>Env</th>
</tr>
</thead>
<tbody>
<tr>
<td>prev arg</td>
<td>prev local</td>
</tr>
<tr>
<td>saved ebp</td>
<td>saved eip</td>
</tr>
<tr>
<td>arg a</td>
<td>arg b</td>
</tr>
<tr>
<td>saved ebp</td>
<td>saved eip</td>
</tr>
<tr>
<td>new frame</td>
<td>current frame</td>
</tr>
<tr>
<td>main</td>
<td>foo</td>
</tr>
</tbody>
</table>
```

```
000...0
```

```
eip
esp
ebp
```
Returning from a function

What happens after code of `foo(a, b)` is finished?
- Pop the function’s stack frame (move esp to ebp)
- Pop (moves) saved ebp into ebp register
- RET: Pop saved eip into eip register
  (CPU assumes ebp was pointing right above the saved eip)
- Caller (main) pops foo’s arg from the stack

```c
int foo(int a, int b) {
    int d = 1;
    return a+b+d;
}

int main(...) {
    ...
    int x = foo(5, 6);
    ...
}
```
Returning from a function

What happens after code of \( \text{foo}(a, b) \) is finished?
- Pop the function’s stack frame (move esp to ebp)
- Pop (moves) saved ebp into ebp register
- RET: Pop saved eip into eip register
- Caller (main) pops foo’s arg from the stack

**Key Point:**
The CPU determines what code & data to execute next, based on values stored on the stack
Outline: Memory Safety: Attacks & Defenses

1. Review: Memory layout and function calls in a process

2. Attacks:
   1. Stack-based buffer overflow attacks
   2. Heap vulnerabilities (briefly)

3. Defenses:
   1. Stack Canaries
   2. Address-Space Layout Randomization (ASLR)
   3. \( W^X \) and ROP
   4. Fuzzing and Memory Safe Languages
Classic Attack: Overflowing a buffer on the stack

Function `bad` copies a string into a 64 character buffer.
— `strcpy` continues copying until it hits NULL character!
— If `s` points to longer string, this overwrites rest of stack frame.
— Most importantly saved `eip` is changed, altering control flow.

```c
void bad(char *s) {
  char buf[64];
  strcpy(buf, s);
}
```
Classic Attack: Overflowing a buffer on the stack

Function `bad` copies a string into a 64 character buffer.
— `strcpy` continues copying until it hits NULL character!
— If `s` points to longer string, this overwrites rest of stack frame.
— Most importantly saved `eip` is changed, altering control flow.

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

`s=“AAAA...AAAA”` (70 or more characters)

Frame before `strcpy` Frame after `strcpy`

saved `eip` should be here!
AAAA=0x41414141 will be used as return address
Classic Attack: Overflowing a buffer on the stack

Function `bad` copies a string into a 64 character buffer.
- `strcpy` continues copying until it hits NULL character!
- If `s` points to longer string, this overwrites rest of stack frame.
- Most importantly saved `eip` is changed, altering control flow.

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

Frame before `strcpy`  Frame after `strcpy`

```
local buf
    <buf cont.>
    <buf cont.>
    ...
    <buf cont.>
saved ebp
saved eip
arg s
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
```

`s = "AAAA...AAAA"` (70 or more characters)

```
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
```

Virtual Memory

```
000...0
main
foo

virtual memory

fff...
stack
env

414...1
```

What will happen? SEGFAULT!

Saved `eip` should be here! `AAAA=0x41414141` will be used as return address.

Segmentation fault!
How to exploit a stack buffer overflow

Suppose attacker can cause bad to run with an `s` it chooses.
- Step 1: Set correct bytes to *point back to input*(!)

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

$s$ = “AAAA…AAAA\x24\xf6\xff\xbfAAA…”

Frame before `strcpy` Frame after `strcpy`

Well-chosen characters used as an address when overwriting the saved `eip`!

What will happen? Illegal instruction!
How to exploit a stack buffer overflow

Suppose attacker can cause bad to run with an s it chooses.

- Step 1: Set correct bytes to *point back to input*(!)
- Step 2: Make input *executable machine code*(!)

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

s=“<machine code>\x24\xf6\xff\xbfAAA...”

![Frame before strcpy](chart1)

![Frame after strcpy](chart2)

Frame before strcpy

<table>
<thead>
<tr>
<th>local buf</th>
<th>&lt;buf cont.&gt;</th>
<th>&lt;buf cont.&gt;</th>
<th>...</th>
<th>&lt;buf cont.&gt;</th>
<th>saved ebp</th>
<th>saved eip</th>
<th>arg s</th>
</tr>
</thead>
</table>

Frame after strcpy

<table>
<thead>
<tr>
<th>&lt;code&gt;</th>
<th>&lt;code&gt;</th>
<th>&lt;code&gt;</th>
<th>&lt;code&gt;</th>
<th>&lt;code&gt;</th>
<th>&lt;code&gt;</th>
<th>&lt;code&gt;</th>
<th>0xbfffff624</th>
</tr>
</thead>
</table>

Program runs attacker’s code once the function (bad) returns!
What to put in for `<code>`?

The possibilities are endless!
— Spawn a shell
— Spawn a new service listening to network
— Change files
— ...

But wait... what about NULL bytes?

**Solution:** Find machine instructions with no NULLs!
— Can even find machine code with all alpha bytes.

```
s="<machine code>\x24\xf6\xff\xbfAAA..."  
(code contains 0x0)
```

Frame after `strcpy`

```
<code>
<code>
...00
<unchanged>
<unchanged>
saved ebp
saved eip
AAAA
```

`strcpy` stopped here, saving victim :(

Example Shellcode

```c
char shellcode[] =
"\xeb\x1f\x5e\x89\x76\x31\xc0\x88\x46\x08\x90\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xcd\xff\xff\xff\bin/sh";
```

Basically equivalent to:

```c
#include <stdio.h>
void main() {
    char *name[2];
    name[0] = "\bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```
Finally, where did that magic address come from?

Assignment: GDB is your friend 😊

Two challenges:
— Need that address to jump to beginning of shellcode
— Need to precisely place it to overwrite saved EIP

```c
void bad(char *s) {
    char buf[64];
    strcpy(buf, s);
}
```

```c
s="\x24\xf6\xff\xbfAAA..."
```

![Diagram showing memory layout with addresses and saved EIP]
Technique #1: NOP Sleds

— Instruction 0x90 is “xchg eax, eax”, i.e. does not thing. This is a “No Op” or “NOP”.
— Just add a ton of NOPs (as many as you can, even many MB) and hope pointer lands there
Technique #2: Placing malicious EIP

— Simple: Just copy it many times

```
0xbffff624
0x90909090
0x90909090...
0x90909090
0xbffff624
0xbffff624
```

saved eip

```
0xbffff624
0xbffff624
0xbffff624
...```
Brief Recap: Stack Buffer Overflows

- Bugs in code can allow attackers to bypass OS security and access control policies

- The CPU stores critical “control flow” information on the stack
  - Saved EIP & Saved EBP: controls what the CPU does after a function returns
  - Buffer overflow attack: vulnerable program doesn’t check if a (stack) buffer has enough space to hold copied data
  - Attacker can provide input that overflows buffer & has: {malicious code} + {new return address, that points to the malicious code}
  - After returning from current function, the CPU will run the attacker’s code, instead of the program’s actual code
Heap Memory: Many Kinds of Vulnerabilities

Initially, the program has:
- A heap variable (heap_buf)
- A function pointer allocated on the heap that points to foo(...)

Attack:
- Overflowing heap_buf can overwrite the heap func ptr
- Later, when program calls the func ptr, it will execute the attacker’s code in heap_buf

Heap overflow attacks can also overwrite variables that get used later in code (e.g., admin = False -> admin = True)

Many other heap bugs:
- Use-after-free,
- Double Free,
- Corrupting metadata…
Outline: Memory Safety: Attacks & Defenses

1. Review: Memory layout and function calls in a process

2. Attacks:
   1. Stack-based buffer overflow attacks
   2. Heap vulnerabilities (briefly)

3. Defenses:
   1. Stack Canaries
   2. Address-Space Layout Randomization (ASLR)
   3. W^X and ROP
   4. Fuzzing and Memory Safe Languages
Countermeasure #1: Stack Canaries
Stack Canaries (a.k.a. Stack Protectors)

- **Idea**: Try to detect if stack data is corrupted, before using it after the function returns.

- **Compiler** inserts additional instructions (code) to each function:
  - At the start of every function, push a “canary” value onto stack between local variables and saved ebp/eip
  - Before returning, additional code checks if canary value is still correct; If not, ABORT.

---

**Standard frame**

- local d
- saved ebp
- saved eip
- arg b
- arg a

**Frame with canary**

- local d
- canary
- saved ebp
- saved eip
- arg b
- arg a

**After overflow**

- AAAA
- AAAA
- AAAA
- ...
How should we (defender) pick the canary value?

**Null**: Set to 0x00000000. Hard for attacker to copy NULLs onto stack.

**Terminator**: 0x000d0aff (for example.) 0x0d=CR, 0x0a=LF, 0xff=EOF. Some buggy code will stop at these characters.

**Random**: Process chooses random value at start, uses same value in every call.
## Stack Canaries in gcc

<table>
<thead>
<tr>
<th>Flag</th>
<th>Default?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fno-stack-protector</td>
<td>No</td>
<td>Turns off protections</td>
</tr>
<tr>
<td>-fstack-protector</td>
<td>Yes</td>
<td>Adds to funcs that call <code>alloca()</code> &amp; w/ arrays larger than 8 chars (--param=ssp-buffer-size changes 8)</td>
</tr>
<tr>
<td>-fstack-protector-strong</td>
<td>No</td>
<td>Also funcs w/ any arrays &amp; refs to local frame addresses. Introduced by ChromeOS team.</td>
</tr>
<tr>
<td>-fstack-protector-all</td>
<td>No</td>
<td>All funcs</td>
</tr>
</tbody>
</table>

- With `-fstack-protector`, 2.5% of functions in kernel covered, 0.33% larger binary
- With `-fstack-protector-strong`, 20.5% of functions in kernel covered, 2.4% larger binary
Related ProPolice Feature: Rearranging Locals

- gcc puts local arrays below other locals, even if declared in other order

```c
int foo(...) {
    char *p;
    char buf[64];
    ...
}
```

```c
int foo(...) {
    char buf[64];
    char *p;
    ...
}
```

**vs**

```c
local buf[]
...
local buf[]
local *p
canary
saved ebp
saved eip
arg b
arg a
```

```c
local *p
local buf[]
...
local buf[]
canary
saved ebp
saved eip
arg b
arg a
```
Bypassing Canaries via Complex Bugs

```c
int foo(char *s1, char *s2) {
    char *p;
    char buf[64];
    p = buf;
    strcpy(p, s1); // oh no :(
    ...            
    strncpy(p, s2, 16);
    ...            
}
```
Bypassing Canaries via Complex Bugs

```c
int foo(char *s1, char *s2) {
    char *p;
    char buf[64];
    p = buf;
    strcpy(p, s1); // oh no :( 
    ... 
    strncpy(p, s2, 16);
    ...
}
```

Attacker crafts s1 to:

1) Fill buff with shellcode
2) Overwrite p to point to the saved eip
   (by overflowing one word longer than buf)
Bypassing Canaries via Complex Bugs

```c
int foo(char *s1, char *s2) {
    char *p;
    char buf[64];
    p = buf;
    strcpy(p, s1); // oh no :(
    ...
    strncpy(p, s2, 16);
    ...
}
```

**Attacker crafted s2 to:**

- Point into `buf` (where shellcode was copied)
Bypassing Canaries via “Reading the Stack”

Request that contains overflow

Web server `fork()`'s child to handle request

Child inherits same random canary value 0xXXYYZZWW.

Response or crash

Overflow 1 byte and observe if process crashes.

- If no crash: we guessed that canary byte value correctly!
- Learn byte xx after max of 256 tries! Repeat for rest.
Another Similar Countermeasure: Shadow Stacks

**Idea:** Have the compiler add additional code to each function that:

- Makes a copy of func’s saved eip in separate memory segment (outside stack)
- Checks whether func’s saved eip on the stack matches this “shadow” copy before returning
Outline: Memory Safety: Attacks & Defenses

1. Review: Memory layout and function calls in a process

2. Attacks:
   1. Stack-based buffer overflow attacks
   2. Heap vulnerabilities (briefly)

3. Defenses:
   1. Stack Canaries
   2. Address-Space Layout Randomization (ASLR)
   3. W^X and ROP
   4. Fuzzing and Memory Safe Languages
Address-Space Layout Randomization (ASLR)

**Idea:** OS makes it hard to know / guess function return addresses (what value the attacker should overwrite the saved eip with)

Linux PaX implementation:

- OS adds random offsets in green areas (location of stack, heap and text)
- 16 bits, 16 bits, 24 bits or randomness respectively

Possible attacks:

- Huge NOP sleds + Copy shellcode many times in heap.
- Side channels (or printf bugs) can leak random choice
- Brute force with large number of forks

Modern machines have 64-bit addresses, making ASLR stronger.
Outline: Memory Safety: Attacks & Defenses

1. Review: Memory layout and function calls in a process

2. Attacks:
   1. Stack-based buffer overflow attacks
   2. Heap vulnerabilities (briefly)

3. Defenses:
   1. Stack Canaries
   2. Address-Space Layout Randomization (ASLR)
   3. W ^ X and ROP Attacks
   4. Fuzzing and Memory Safe Languages
## W ^ X ("Write XOR Execute")

### Virtual Memory

**Perms**
- \( r, x \) .text
- \( r \) .data
- \( r, w \) .bss
- \( r, w \) heap
- \( r, x \) libc
- \( r, w \) stack

#### Idea:

Code should not be writable & Data should not be executable
- e.g., stack memory = writable, but not executable

*OS* will mark each memory segment* as either writeable or executable, but never both.

- Modern hardware support: x64 (the x86 successor)
- All major OS implement (PaX/ExecShield - Linux, DEP - Windows, …)
- Also used in virtual machine / sandboxes
Bypassing $W \land X$: Return-to-libc

Virtual Memory

<table>
<thead>
<tr>
<th>Perms</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>r,x</td>
<td>.text</td>
</tr>
<tr>
<td>r</td>
<td>.data</td>
</tr>
<tr>
<td>r,w</td>
<td>.bss</td>
</tr>
<tr>
<td>r,w</td>
<td>heap</td>
</tr>
<tr>
<td>r,x</td>
<td>libc</td>
</tr>
<tr>
<td>r,w</td>
<td>stack</td>
</tr>
</tbody>
</table>

- local buf[]
- ...
- local buf[]
- saved ebp
- saved eip
- arg s2
- arg s1
Bypassing W ^ X: Return-to-libc

Virtual Memory

Perms | .text | .data | .bss | heap | libc | stack
---|---|---|---|---|---|---
$r,x$ | $r$ | $r,w$ | $r,w$ | $r,x$ | $r,w$ | 

```
local buf[]

xxxx
xxxx
xxxx
new eip
xxxx
target args
```

The Attack:
- Overwrite eip to point to target func in libc (`system`)
- Overwrite stack to setup args for the target func
- Result: Function is called w/ specific args!
Bypassing W ^ X: Return-to-libc Details

Attack Goal: Spawn shell for attacker, e.g., system("/bin/sh")
- Overwrite eip to point to target func in libc (system)
- Overwrite stack to setup args for target func ("/bin/sh")
Bypassing W ^ X: Return-to-libc Details

Attack Goal: Spawn shell for attacker, e.g., `system("/bin/sh")`
- Overwrite eip to point to target func in libc (`system`)
- Overwrite stack to setup args for target func ("/bin/sh")
- Result: `system("/bin/sh")` is called!
Return-to-libc enables attacker to call existing functions (e.g., from libc)

Going further: Why not “return” into the middle of functions, and only execute final instructions?
  - Finer-grain control: can execute a few select instructions, rather than entire predefined functions

General ROP attack (Shacham 2008):
  - Search through common library code (e.g., libc) for functions that end in useful instructions.
  - Build shellcode as a series of “return addr’s” that point to useful instructions.
    (RET instruction pops next word on the stack into %eip)
Outline: Memory Safety: Attacks & Defenses

1. Review: Memory layout and function calls in a process

2. Attacks:
   1. Stack-based buffer overflow attacks
   2. Heap vulnerabilities (briefly)

3. Defenses:
   1. Stack Canaries
   2. Address-Space Layout Randomization (ASLR)
   3. \( W^X \) and ROP
   4. Fuzzing and Memory Safe Languages
Program Fuzzing: Find bugs before release

Idea: Developer runs their program on huge number of automatically-generated inputs, searches for crashes, and fixes bugs before releasing software.

Linux Mint fixes screensaver bypass discovered by two kids

Two children playing on their dad's computer accidentally found a way to bypass the screensaver and access locked systems.

"A few weeks ago, my kids wanted to hack my Linux desktop, so they typed and clicked everywhere while I was standing behind them looking at them play," wrote a user identifying themselves as robo2bobo.

According to the bug report, the two kids pressed random keys on both the physical and on-screen keyboards, which eventually led to a crash of the Linux Mint screensaver, allowing the two access to the desktop.

"I thought it was a unique incident, but they managed to do it a second time," the user added.
Types of Fuzzing

**Mutation-based (dumb):** Take an initial set of examples (program inputs) and make random changes to them.

- Millions of inputs (can run fuzzing forever)
- Possibly lower quality, unlikely to find certain bugs / types of inputs

**Generative (smart):** Describe inputs to fit format/protocol, then generate inputs from that grammar with changes.

- Run with fewer inputs, which can be directed to certain bug types or code logic
Problems with Fuzzing

**Mutation-based (dumb):** How long to run? And we need a strong server.

**Generative (smart):** Run out of test cases. A lot more work.

**General problems:**

- Need to identify when bug/crash occurs automatically.
- Don’t want to report same bug 1000s of times.
- How do we prioritize bugs?
Fuzzing in Production

AFL: Popular open-source fuzzer released by Google

Google/Microsoft constantly fuzz products with dedicated servers/VMS.

Anecdote: Found 95 vulnerabilities in Chrome during 2011.

OneFuzz

A self-hosted Fuzzing-As-A-Service platform

Project OneFuzz enables continuous developer-driven fuzzing to proactively harden software prior to release. With a single command, which can be baked into CI/CD, developers can launch fuzz jobs from a few virtual machines to thousands of cores.
Many of our problems can be solved by using “memory-safe” languages.

- The programming model for these languages *does not allow* for such bugs (e.g., no access to pointers / mem addr’s and built-in object bounds checking).

<table>
<thead>
<tr>
<th>Not Memory-Safe</th>
<th>Memory Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Java</td>
</tr>
<tr>
<td>C++</td>
<td>Python</td>
</tr>
<tr>
<td>Assembly</td>
<td>Javascript</td>
</tr>
<tr>
<td></td>
<td>Rust, Go, Haskell, …</td>
</tr>
</tbody>
</table>

Ideally, we’d avoid writing programs in unsafe languages, but lots of legacy code (and low-level stuff) are written in C/C++.
Software Defenses

Pre-deployment, before the program runs: find or prevent bugs
- Fuzzing: proactively finding & fixing bugs by testing many program inputs
- Memory safe languages: automatically avoid exploitable memory bugs
- Done by the application developer

Program runtime: stopping exploits / violations of program’s memory
- Stack Canaries, ASLR, DEP/W+X, etc.
- Implemented by the compiler (stack canary) or operating system (ASLR, W+X)
- Attacks adapt & evolve (Stack reading, ROP attacks, etc.)

Post-exploitation (not covered today): limit possible damage from compromise
- Sandboxing and VMs
- Done by user/admin of the system or the app developer (e.g., web browsers)
The End