Announcements

• Lab 1
First, Review slides from Friday and Wednesday
Memory Layout

• **Text region:** Executable code of the program
• **Heap:** Dynamically allocated data
• **Stack:** Local variables, function return addresses; grows and shrinks as functions are called and return

![Memory Layout Diagram]

<table>
<thead>
<tr>
<th>Text region</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr 0x00...0</td>
<td>Heap</td>
<td>Stack</td>
</tr>
</tbody>
</table>
Stack Buffers

• Suppose Web server contains this function:

```c
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

• When this function is invoked, a new **frame** (activation record) is pushed onto the stack.
What if Buffer is Overstuffed?

• Memory pointed to by str is copied onto stack...

```c
void func(char *str) {
    char buf[126];
    strcpy(buf, str);
}
```

• If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

`strcpy` does NOT check whether the string at *str contains fewer than 126 characters

This will be interpreted as return address!
Executing Attack Code

• Suppose buffer contains attacker-created string
  • For example, str points to a string received from the network as the URL
  
  Attacker puts actual assembly instructions into his input string, e.g., binary code of \texttt{execve(“/bin/sh”)}

  In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

• When function exits, code in the buffer will be executed, giving attacker a shell (\texttt{“shellcode”})
  • Root shell if the victim program is setuid root
Closer Look at the Stack

```c
printf("Numbers: %d,%d", 5, 6);
```

```c
printf("Numbers: %d,%d");
```

strokeoreal
Writing Stack with Format Strings

- `%n` format symbol tells `printf` to write the number of characters that have been printed

  ```c
  printf("Overflow this!\%n", &myVar);
  ```

- Argument of `printf` is interpreted as destination address
- This writes 14 into `myVar` ("Overflow this!" has 14 characters)

- What if `printf` does **not** have an argument?

  ```c
  char buf[16]="Overflow this!\%n";
  printf(buf);
  ```

- Stack location pointed to by `printf`’s internal stack pointer will be interpreted as address into which the number of characters will be written.
How Can We Attack This? Breakout -> In-Class Activity

```c
foo() {
    char buf[...];
    strncpy(buf, readUntrustedInput(), sizeof(buf));
    printf(buf); //vulnerable
}
```

What should the string returned by `readUntrustedInput()` contain??

Different compilers / compiler options / architectures might vary
Using %n to Overwrite Return Address

In foo()'s stack frame:

Buffer with attacker-supplied input “string”

This portion contains enough % symbols to advance printf’s internal stack pointer

“... attackString%n...”, attack code

Number of characters “in” attackString must be equal to ... what?

&RET

When %n happens, make sure the location under printf’s internal stack pointer contains address of RET; %n will write the number of characters printed so far into RET

SFP

Return execution to this address

RET

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte.

(4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

Why is “in” in quotes? C allows you to concisely specify the “width” to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf(“%5d%n”, 10) will print three spaces followed by the integer: “   10”

That is, the %n will write 5, not 2.
Review: “End”
Recommended Reading

• It will be hard to do Lab 1 without:
  • Reading (see course schedule):
    • Smashing the Stack for Fun and Profit
    • Exploiting Format String Vulnerabilities
  • Attending section this week and next
Buffer Overflow: Causes and Cures

• Classical memory exploit involves code injection
  • Put malicious code at a predictable location in memory, usually masquerading as data
  • Trick vulnerable program into passing control to it

• Possible defenses:
  1. Prevent execution of untrusted code
  2. Stack “canaries”
  3. Encrypt pointers
  4. Address space layout randomization
  5. Code analysis
  6. ...
Defense: Executable Space Protection

• Mark all writeable memory locations as non-executable
  • Example: Microsoft’s Data Execution Prevention (DEP)
  • This blocks many code injection exploits

• Hardware support
  • AMD “NX” bit (no-execute), Intel “XD” bit (executed disable) (in post-2004 CPUs)
  • Makes memory page non-executable

• Widely deployed
  • Windows XP SP2+ (2004), Linux since 2004 (check distribution), OS X 10.5+ (10.4 for stack but not heap), Android 2.3+
Canvas In-Class Activity

• What might an attacker be able to accomplish even if they cannot execute code on the stack?
What Does “Executable Space Protection” Not Prevent?

• Can still corrupt stack ...
  • ... or function pointers
  • ... or critical data on the heap

• As long as RET points into existing code, executable space protection will not block control transfer!
  ➔ return-to-libc exploits
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
  • …
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`

---

![Diagram](image.png)

- Overwrite destination of `strcpy` with RET position
- `strcpy` will copy `BadPointer` here
- Return execution to this address
ASLR: Address Space Randomization

• Randomly arrange address space of key data areas for a process
  • Base of executable region
  • Position of stack
  • Position of heap
  • Position of libraries
• Introduced by Linux PaX project in 2001
• Adopted by OpenBSD in 2003
• Adopted by Linux in 2005
ASLR: Address Space Randomization

• Deployment (examples)
  • Linux kernel since 2.6.12 (2005+)
  • Android 4.0+
  • iOS 4.3+ ; OS X 10.5+
  • Microsoft since Windows Vista (2007)

• Attacker goal: Guess or figure out target address (or addresses)

• ASLR more effective on 64-bit architectures
Attacking ASLR

- **NOP sleds** and **heap spraying** to increase likelihood for adversary’s code to be reached (e.g., on heap)
- **Brute force attacks or memory disclosures** to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation
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

Memory

CPU

-pointer
0x1234

Data

0x1234

CPU

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

Memory

-pointer
0x1234

Data

0x1234

Attack code

0x1340

[Cowan]
PointGuard Dereference

1. Fetch pointer value
   - Encrypt 0x7239
   - Decrypt

2. Access data referenced by pointer
   - Decrypt 0x1234

Memory

CPU

Corrupted pointer
- 0x7239
- 0x1340

Data

Attack code

0x1234

0x1340

0x9786

CSE 484 - Winter 2022
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 emerged (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.)
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