CSE 484 / CSE M 584: Software Security

Winter 2023

Tadayoshi (Yoshi) Kohno
yoshi@cs.Washington.edu

UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...
Off-by-One Overflow

• Home-brewed range-checking string copy

```c
void mycopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
}

void main(int argc, char *argv[]) {  
    if (argc==2)
        mycopy(argv[1]);
}
```

This will copy 513 characters into buffer. Oops!

• 1-byte overflow: can’t change RET, but can change pointer to previous stack frame...
Frame Pointer Overflow

State includes:
- Instruction pointer
- Frame pointer

Imagine:
- Foo() calls Bar()
- Bar() calls mycopy()

Overwrite saved frame pointer means that adversary’s code will start executing after Bar() returns.
Another Variant: Function Pointer Overflow

• C uses function pointers for callbacks: if pointer to F is stored in memory location P, then one can call F as (*P)(...)

Diagram:
- Buffer with attacker-supplied input string
- Callback pointer
- attack code
- overflow
- Legitimate function F (elsewhere in memory)
Other Overflow Targets

• Format strings in C
  – We’ll walk through this later

• Heap management structures used by malloc()
  – More details in section
  – Techniques have changed wildly over time

• These are all attacks you can look forward to in Lab #1 😊
Review: Printf() and the Stack

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

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

Printf’s internal stack pointer starts here

Printf’s internal stack pointer starts here

CSE 484 / CSE M 584 - Winter 2023
Summary of Printf Risks

• Printf takes a variable number of arguments
  – E.g., `printf(“Here’s an int: %d”, 10);`
• Assumptions about input can lead to trouble
  – E.g., `printf(buf)` when `buf=“Hello world”` versus when `buf=“Hello world %d”`
  – Can be used to advance printf’s internal stack pointer
  – Can read memory
    • E.g., `printf(“%x”)` will print in hex format whatever printf’s internal stack pointer is pointing to at the time
  – Can write memory
    • E.g., `printf(“Hello%n”);` will write “5” to the memory location specified by whatever printf’s internal SP is pointing to at the time
foo() {
    char buf[...];
    strncpy(buf, readUntrustedInput(), sizeof(buf));
    printf(buf);  //vulnerable
}

What should the string returned by readUntrustedInput() contain??
Canvas -> Quizzes -> January 13

Note: Different compilers / compiler options / architectures might vary

If format string contains % then printf will expect to find arguments here...
Using %n to Overwrite Return Address

"... attackString%n...", attack code

Buffer with attacker-supplied input "string"

&RET

SFP

RET

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

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

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

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.

In foo()’s stack frame:
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+
Question

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
  – … Right?
  – 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
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`
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

**Diagram:**

1. **Normal Pointer Dereference**
   - Fetch pointer value: 0x1234
   - Access data referenced by pointer: 0x1234

2. **Corrupted Pointer Dereference**
   - Fetch pointer value: Corrupted pointer
   - Access attack code referenced by corrupted pointer: 0x1340

---

[Cowan]
PointGuard Dereference

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

CPU

Memory

Encrypted pointer 0x7239

Data 0x1234

Corrupted pointer 0x7239 0x1340

Data 0x1234

Attack code 0x9786

[Random address; segmentation fault and crash]
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