Defending against Memory-Corruption Vulnerabilities and Advanced Attacks

Gang Tan
Penn State University
Spring 2019

CMPSC 447, Software Security

* some slides adapted from those by Trent Jaeger
Overflow Vulnerabilities

- Despite knowledge of buffer overflows for over 40 years, they have not been eliminated.
- This is partly due to the wide variety of exploit options:
  - Variety of targets: can exploit more than return addresses – any code addresses or data
  - Variety of uses: can exploit on read and write
  - Variety of exploits: can inject or reuse code
  - Variety of workarounds: current defenses are incomplete
Defense

- We can take countermeasures at different points in time
  - before we even begin programming
  - during development
  - when testing
  - when code is running

- Next we will discuss mostly two kinds
  - Detection and mitigation at runtime
  - Prevention during development
    - Defensive programming
  - Prevention during compilation
Detection: Stack Canaries

- AKA stack cookies
- Introduced in StackGuard in gcc
- A dummy (or random) value is written on the stack in front of the return address and checked when function returns
- A careless stack overflow will overwrite the canary, which can then be detected
  - Assume sequential stack smashing
Ways of Defeating StackGuard

- Overwrite the canary with the correct value
  - If the range of the random value is small, just try every possibility
  - Or perform a memory disclosure attack first
    - To learn the canary value on the stack
StackGuard Limitations

- **Big limitation**: Disclosure attacks
  - By performing a buffer “overread”
  - Example is the famous Heartbleed attack against SSL
  - Why is this a problem for Stackguard canaries?

```c
char packet[10];
...
// suppose len is adversary controlled
strncpy(buf, packet, len);
send(fd, buf, len);
```
StackGuard Limitations

- **Big limitation**: Disclosure attacks
  - By performing a buffer “overread”
  - Example is the famous Heartbleed attack against SSL
  - Why is this a problem for Stackguard?

```c
char packet[10];
...
// suppose len is adversary controlled
strncpy(buf, packet, len);
send(fd, buf, len);
```
StackGuard Limitations

- **Big limitation**: disclosure attacks
  - By performing a buffer “overread”
  - One may extract the canary values by reading beyond the end of stack buffers
  - Which would enable the adversary to learn the (supposedly secret) canary value
More Ways of Defeating StackGuard

- Sometimes no need to overwrite the return address
  - Can overflow
    - Security-sensitive local variables
    - Heap overflow
    - Global data overflow
    - ...
  - Ultimately, an attacker only needs to hijack a function pointer
Example: Hijacking a function pointer

void foo () {...}
void bar () {...}
int main() {
    char buf [16];
    void (*f) () = &foo;
    gets(buf);
    f();
}

- Assume no chance of overflowing the return address
- Can overflow the buffer so that the function pointer is modified to be the address of bar
  - Then the function call will call bar instead of foo
Other Ways of Hijacking Function Pointers

- Use heap overflows to hijack a function pointer on the heap
- Hijacking global function pointers
- Function pointers in Global Offset Table (GOT)
  - Used for dynamically linked functions
Global Data Overflow

- Can attack buffer located in global data
  - may be located above program code
  - if has function pointer and vulnerable buffer
  - or adjacent process management tables
  - aim to overwrite function pointer later called
/* global static data - targeted for attack */
struct chunk {
    char inp[64];        /* input buffer */
    void (*process)(char *); /* ptr to function */
} chunk;

void showlen(char *buf)
{
    int len;
    len = strlen(buf);
    printf("buffer6 read %d chars\n", len);
}

int main(int argc, char *argv[])
{
    setbuf(stdin, NULL);
    chunk.process = showlen;
    printf("Enter value: ");
    gets(chunk.inp);
    chunk.process(chunk.inp);
    printf("buffer6 done\n");
}
Detection: Guard Pages

- Can be thought of as extension of StackGuard
- Place guard pages between critical regions of memory
  - flagged in MMU as illegal addresses
  - any access aborts process
- Can even place between stack frames and heap buffers
  - at the cost of performance/memory overhead
Runtime Mitigation: DEP (Data Execution Prevention)

- Computer architectures follow a Von-Neumann architecture
  - Storing code as data
  - This allows an attacker to inject code into stack or heap, which is supposed to store only data
- A Harvard architecture is better for security
  - Divide the virtual address space into a data region and a code region
  - The code region is readable (R) and executable (X)
  - The data region is readable (R) and writable (W)
  - No region is both writable and executable
    - An attacker can inject code into the stack, but cannot execute it
Runtime Mitigation: DEP (Data Execution Prevention)

- DEP prevents code-injection attacks
  - AKA Nx-bit (non executable bit), $W \oplus X$

- DEP is now supported by most OSes and ISAs
Runtime Mitigation: DEP

- Issue: some legit apps support executable code in the data region
  - e.g., a JIT (Just-In-Time) compiler
  - Runtime code generation
  - need special provisions
Defeating DEP: Code Reuse Attacks

- Idea: reuse code in the program itself
  - No need to inject code
- Return-to-libc: replace return address with the address of a dangerous library function
  - Attacker constructs suitable parameters on stack above return address
    - On x64, need more work of setting up parameter-passing registers
  - Function returns and library function executes
    - E.g. `execve("/bin/sh")`
  - Can even chain two library calls
void foo () {...}
int main() {
    char buf [16];
    void (*f) () = &foo;
    gets(buf);
    f();
}

- Attack 1: changing f’s value to a libc system function and put the arguments on the stack
- Attack 2: chain two calls of libc functions
Code Injection vs Code Reuse

- The difference is subtle, but significant
  - In **code injection**, we wrote the address of `execve` into buffer on the stack and modified return address to start executing at buffer
    - i.e., we are executing in the stack memory region
  - In **code reuse**, we can modify the return address to point to `execve` directly, so we continue to execute code
    - Reusing available code to do what the adversary wants
Code Injection vs Code Reuse

- **Code Injection**
  - `execve("/bin/sh")`
  - `ret`
  - Stack frame for main

- **Code Reuse**
  - `buffer`
  - `ret`
  - Stack frame for main
  - `execve("/bin/sh")`
  - Existing code
In many attacks, a code reuse attack is used as a first step to disable DEP
- Goal is to allow execution of stack memory
- There’s a system call for that
  ```c
  int mprotect(void *addr, size_t len, int prot);
  ```
- Sets protection for region of memory starting at address
- Invoke this system call to allow execution on stack and then start executing from the injected code
Code Reuse: ROP

- Return-Oriented Programming (ROP)
  - [Shacham et al], 2008
  - Arbitrary behavior without code injection
  - Combine snippets of existing code (gadgets)
  - A set of Turing-complete gadgets and a way of chaining these gadgets
  - People have shown that in small programs (e.g., 16KB), they can find a turing-complete set of gadgets
Use gadgets to perform general programming:
- arithmetics;
- arbitrary control flow: jumps; loops; ...

ROP's control stack (just data)
Return-Oriented Programming

*The following slides are by Dr. Shacham
any sufficiently large program codebase

arbitrary attacker computation and behavior, *without* code injection
Machine Instructions

- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip
ROP Execution

- Stack pointer (%esp) determines which instruction sequence to fetch & execute
- Processor doesn’t automatically increment %esp; — but the “ret” at end of each instruction sequence does
ROPA Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1: pop %eax
    ret

G2: pop %ebx
    ret

G3: movl %eax, (%ebx)
    ret

%eax = 
%ebx =
```

```
Code

Stack

Return Address

buf

Registers

Memory

0x8048000 =
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control.
  - Termed "return-oriented attacks" (ROP)

Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```plaintext
G1:

G2:  pop %ebx
    ret

G3:  movl %eax, (%ebx)
    ret

Registers
%eax = 5
%ebx =

Stack

Memory

Code Stack

G1
5
G2
0x8048000
G3
...

Return Address

buf

Buffers

Jump G2

Return Address

G1:

G3:

5

0x8048000 =
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control.
  - Termed "return-oriented attacks" (ROP)

Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: pop %eax ret</td>
<td>G1</td>
<td>%eax = 5</td>
<td>0x8048000 =</td>
</tr>
<tr>
<td>G2: pop %ebx ret</td>
<td>5</td>
<td>%ebx =</td>
<td></td>
</tr>
<tr>
<td>G3: movl %eax, (%ebx) ret</td>
<td>G2</td>
<td></td>
<td>buf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x80480000</td>
</tr>
</tbody>
</table>
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks" (ROP)

Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:      pop %eax
        ret

G2:      movl %eax, (%ebx)
        ret

G3:      movl %eax, (%ebx)
        ret
```

```
%eax = 5
%ebx = 0x8048000
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control.

  - Termed “return-oriented attacks” (ROP)

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:  pop %eax
    ret

G2:  pop %ebx
    ret

G3:  movl %eax, (%ebx)
    ret
```

```
%eax = 5
%ebx = 0x8048000
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control.
  - Termed “return-oriented attacks” (ROP)

Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
# Code
G1:  pop %eax
    ret

G2:  pop %ebx
    ret

G3:   

# Stack
G1
5
G2
0x8048000
G3...

# Registers
%eax = 5
%ebx = 0x8048000

# Memory
0x8048000 =
```

Return Address
buf
RDP Example

• Use ESP as program counter
  – E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1: pop %eax
    ret

G2: pop %ebx
    ret

G3:  

%eax = 5
%ebx = 0x8048000
0x8048000 = 5
```

- Termed "return-oriented attacks" (ROP)

- Example

```
%eax = %ebx = 0x8048000
```

```
G1: pop %eax
    ret

G2: pop %ebx
    ret

G3:  

%eax = 5
%ebx = 0x8048000
0x8048000 = 5
```
Building ROP Functionality

- **No-ops**
  - C library
  - ret

- **Instruction Pointer**
  - no-op instruction does nothing but advance `%eip`

- **Return-oriented equivalent**:
  - point to return instruction
  - advances `%esp`

- **Useful in nop sled**
Building ROP Functionality

Immediate constants

- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use
Building ROP Functionality

Control flow

- Ordinary programming:
  - (Conditionally) set %eip to new value
- Return-oriented equivalent:
  - (Conditionally) set %esp to new value
Return-oriented Programming

- What can we do with return-oriented programming?
  - Anything any other program can do
  - How do we know?
Return-oriented Programming

- What can we do with return-oriented programming?
  - Anything any other program can do
  - How do we know? Turing completeness

- A language is Turing complete if it has (loosely)
  - Conditional branching
  - Can change memory arbitrarily

- Both are possible in ROP
Protection against ROP

- ROP works by changing the control flow of the program

- Control-flow integrity (CFI)
  - Take a vulnerable program and a pre-determined a control-flow graph
  - Insert checks into the program so that it stops working if an illegal control flow transfer happens during runtime
    - Via compiler changes or binary rewriting
  - More on this later
Runtime Mitigation: Randomization

- Exploits requires knowing code/data addresses
  - E.g., the start address of a buffer
  - E.g., the address of a library function
- Idea: introduce artificial diversity (randomization)
  - Make addresses unpredictable for attackers
- Many ways of doing randomization
  - Randomize location of the stack, location of key data structures on the heap, and location of library functions
  - Randomly pad stack frames
  - At compile time, randomize code generation for defending against ROP
Implementation of Randomization

- Can be performed
  - At compile time
  - At link time
  - Or at runtime (e.g., via dynamic binary rewriting)
Linux Address-Space Layout Randomization (ASLR)

- For a position-independent executable (PIE), randomize
  - The base address of the executable
- All libraries are PIE
  - So their base addresses are randomized
- Main executables may not be PIE
  - May not be protected by PIE
- A form of coarse-grained randomization
  - Only the base address is randomized
  - Relative distances between memory objects are not changed
Ways of Defeating ASLR

- Perform an exhaustive search, if the random space is small
  - E.g., Linux provides 16-bit of randomness
    - It can be defeated by an exhaustive search in about 200s

- ASLR often defeated by memory disclosure
  - E.g., if the attacker can read the value of a pointer to the stack
    - Then he can use it to discover where the stack is
Summary

Defenses
- Stack Guard
- DEP
- ASLR

Advanced attacks
- Memory disclosure (e.g., via buffer overread)
- Code reuse
  - Return to libc
  - ROP