CS 88: Security and Privacy

06: Software Security – Defenses

02-08-2024
Announcements

• Clicker mappings on edstem.
  • please use the google sheet link to update your clicker choices
• Midterm dates:
• Speak to me about accommodations now!
Reading Quiz
Last Class

• Stack Buffer Overflow
• Integer Overflow Vulnerabilities
Today

• Format String Attacks
• Return Oriented Programming
• S/w Defenses
Return-Oriented Programming
Return Oriented Programming: Code Reuse

- Can’t inject code onto the stack (non-executable stack)
  - How about assembly instructions that already exist in our code?
  - What if we string together a few instructions at a time?
- A short sequence of instructions that we construct are called gadget
  - A gadget usually ends in a `ret` instruction.
    - Once we execute `ret`:
      - the address of the next gadget off the stack is popped
      - and control flow jumps to that address.

Source: RPISEC: Return Oriented Programming
Attacks on Non-executable pages

Return into libc: set up the stack and “return” to exec()

- Overwrite stuff above saved return address with a “fake call stack”, overwrite saved return address to point to the beginning of exec() function
- Especially easy on x86 since arguments are passed on the stack
Return Oriented Programming

• Idea: chain together “return-to-libc” idea many times
• ROP compiler
• Tools democratize things for attackers:
  • Find a set of short code fragments (gadgets) that when called in sequence execute the desired function
  • Inject into memory a sequence of saved "return addresses" that will invoke them Sample gadget: add one to EAX, then return
  • Find enough gadgets scattered around existing code that they’re Turing-complete Compile your malicious payload to a sequence of these gadgets
• Yesterday's Ph.D. thesis or academic paper is today's Intelligence Agency tool and tomorrow's Script Kiddie download
Attack: Return Oriented Programming (ROP)

Control hijacking **without injecting code**:

- **Stack**
  - local buf
  - saved ebp
  - return addr
  - args

- **libc.so**
  - exec()
  - printf()
  - "/bin/sh"
Return Oriented Programming: Code Reuse

- Can’t inject code onto the stack (non-executable stack)
  - How about assembly instructions that already exist in our code?
  - What if we string together a few instructions at a time?
- A short sequence of instructions that we construct are called gadgets

Source: RPISEC: Return Oriented Programming
Return Oriented Programming: Code Reuse

- We can get each sequence to end in a “ret” instruction
  - i.e.:
    - pop the value at the top of the stack
    - store this value in eip
    - decrement the stack pointer 4 bytes below.
    - now eip executes whatever instruction is present at this memory address
- at the next call to ret,
  - we again pop the top value of the stack
  - store this value in eip,
  - and so on…

Source: RPISEC: Return Oriented Programming
Return Oriented Programming: chain gadgets to form a ROP chain

Objective: set the `execve` shellcode register state

rax: 0x3b  
rdi: “/bin/sh”  
rsi: 0  
rdx: 0  
syscall
Return Oriented Programming: chain gadgets to form a ROP chain

Final State:
- rax: 0x3b
- rdi: "/bin/sh"
- rsi: 0
- rdx: 0
- syscall

This value is popped off the stack and eip starts executing these instructions when the "ret" call is made.

Overwritten saved eip value

Final State:
- rax: 0x3b
- rdi: "/bin/sh"
- rsi: 0
- rdx: 0
- syscall
Return Oriented Programming: chain gadgets to form a ROP chain

<table>
<thead>
<tr>
<th>rax: ?</th>
<th>rdx: ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsi: ?</td>
<td>rdx: ?</td>
</tr>
</tbody>
</table>

\[
eip = 0x401d70
\]

- `0x400590` - overwritten saved eip value
- `0x489864 -> "/bin/sh"`
- `0x0`
- `0x455e55`

**Final State:**
- `rax: 0x3b`
- `rdi: "/bin/sh"`
- `rsi: 0`
- `rdx: 0`
- `syscall`
Return Oriented Programming: chain gadgets to form a ROP chain

- `rax:?`
- `rdi: 0x489864` -> "/bin/sh"
- `rsi:?`
- `rdx:?`

**Final State:**
- `rax: 0x3b`
- `rdi: "/bin/sh"`
- `rsi: 0`
- `rdx: 0`
- `syscall`

**Overwritten Saved EIP Value:**
- `eip = 0x401d78`
- `pop rdi`
- `pop rax`
- `ret`
- `pop rsi`
- `pop rdx`
- `ret`

**System Call:**
- `syscall`
Return Oriented Programming: chain gadgets to form a ROP chain

**eax:** 0x3b

**rdi:** 0x489864
- “/bin/sh”

**rsi:** ?

**rdx:** ?

**eip** = 0x401d80

- `pop rdi`
- `pop rax`
- `ret`

- `ret` statement treats the next value on the stack as the mem. addr of the next instruction

- `pop rsi`
- `pop rdx`
- `ret`

**Final State:**
- eax: 0x3b
- rdi: “/bin/sh”
- rsi: 0
- rdx: 0
- syscall

- overwritten saved eip value

- esp
Return Oriented Programming: chain gadgets to form a ROP chain

rax: 0x3b

rdi: 0x489864
  -> “/bin/sh”

rsi: ?

rdx: ?

syscall 0x400590
0x401d70

pop rdi
pop rax
ret

pop rsi
pop rdx
ret

0x400590
0x3b

0x489864 -> “/bin/sh”

 esp

Final State:
rax: 0x3b
rdi: “/bin/sh”
si: 0
rdx: 0
syscall

syscall

0x455e55
Return Oriented Programming: chain gadgets to form a ROP chain

rax: 0x3b

di: 0x489864
   -> "/bin/sh"

rsi: 0x0

rdx: ?

Final State:
rax: 0x3b
rdi: "/bin/sh"
rsi: 0
rdx: 0
syscall

eip = 0x400598
pop rsi
pop rdx
ret

syscall

0x3b
0x0
0x0
0x0
0x455e55
0x400590
0x401d70
0x489864 -> "/bin/sh"
Return Oriented Programming: chain gadgets to form a ROP chain

rax: 0x3b
rdi: 0x489864
-> "/bin/sh"
rsi: 0x0
rdx: 0x0

pop rdi
pop rax
ret

eip = 0x4005A0
pop rsi
pop rdx
ret

syscall

Final State:
rax: 0x3b
rdi: "/bin/sh"
rsi: 0
rdx: 0
syscall

ret statement treats the next value on the stack as the mem. addr of the next instruction

esp
Return Oriented Programming: chain gadgets to form a ROP chain

rax: 0x3b

rdi: 0x489864
  -> "/bin/sh"

rsi: 0x0

rdx: 0x0

Final State:
rax: 0x3b
rdi: "/bin/sh"
rsi: 0
rdx: 0
syscall

eip = 0x455e55
syscall

0x401d70
0x489864 -> "/bin/sh"
0x3b
0x400590
0x0
0x455e55
0x0
Exactly the state we set out to achieve! We have successfully launched our shell without injecting any code!

**Objective:** set the execve shellcode register state

- rax: 0x3b
- rdi: "/bin/sh"
- rsi: 0
- rdx: 0

```c
syscall
```

```
eip = 0x455e55
```
What happened?

Programmer:
   This program crashes if the input is too big

Hacker:
   Let’s change some local variables!
   Actually, let’s call some functions...
   Well as long as we’re already here...let’s call some of *our* specially cherry picked instructions (err.. functions).
Buffer Overflow: Cures

Idea: prevent execution of untrusted code
  • Make stack and other data areas non-executable
    • Note: messes up useful functionality (e.g., Flash, JavaScript)
  • Digitally sign all code
  • Ensure that all control transfers are into a trusted, approved code image
Validating input

• Determine acceptable input, check for match --- don’t just check against list of “non-matches”
• Limit maximum length
• Watch out for special characters, escape chars.
• Check bounds on integer values
• Check for negative inputs
• Check for large inputs that might cause overflow!
Validating input

• Filenames
• Command-line arguments
• Even argv[0]...
• Commands
  • E.g., URLs, http variables., SQL
  • E.g., cross site scripting, (next lecture)
Memory attacks

The problem: mixing data with control flow in memory

Your program manipulates data
Data manipulates your program
Memory Attacks: Causes

“Classic” memory exploit involves code injection

- malicious code @ predictable location in memory -> masquerading as data
- trick vulnerable program into passing control
“Classic” memory exploit involves code injection

Idea: prevent execution of untrusted code

Developer approaches:
- Use of safer functions like `strlcpy()`, `strlcat()` etc.
- safer dynamic link libraries that check the length of the data before copying.

Hardware approaches: Non-Executable Stack

OS approaches: ASLR (Address Space Layout Randomization)

Compiler approaches: Stack-Guard Pro-Police
Data Execution Prevention: a.k.a Mark memory as non-executable

Each page of memory has separate access permissions:
• R -> Can Read, W -> Can Write, X -> Can Execute

Mark all writeable memory locations as non-executable

**NX-bit** on AMD64, **XD-bit** on Intel x86 (2005), **XN-bit** on ARM

• Now you can’t write code to the stack or heap
• No noticeable performance impact
Address Space Layout Randomization

Onload: Randomly relocate the base address of everything in memory

- libraries (DLLs, shared libs), application code, stack heap

⇒ attacker does not know location

Example: PAX implementation
Address Space Layout Randomization

32-bit PaX ASLR (x86)

**Stack:**

```
  1 0 1 0 R R R R R R R R R R R R R R R R R R 0 0 0 0
  fixed                           random (24 bits)                                zero
```

**Mapped area:**

```
  0 1 0 0 R R R R R R R R R R R R R R R R 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  fixed                           random (16 bits)                                zero
```

**Executable code, static variables, and heap:**

```
  0 0 0 0 R R R R R R R R R R R R R R R R 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  fixed                           random (16 bits)                                zero
```
Launch buffer overflow? Difficult to guess the stack address!

Difficult to guess %ebp address and address of the malicious code
Compiler Defenses: Stack Canary
Method 1: StackGuard

- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return.
StackGuard

Overflow canary? Segfault!

Random canary:
• Random string **chosen at program startup**
• To corrupt, attacker must learn/guess current random string

Terminator canary:
• `{0, newline, linefeed, EOF}`
• String functions will not copy beyond terminator
• Attacker cannot use string functions to corrupt the stack

Minimal performance effects: 8% for Apache Program must be recompiled
Canary check in gcc:

Dump of assembler code for function foo:

```
0x0000120d <+0>:   endbr32
0x00001211 <+4>:   push %ebp
0x00001212 <+5>:   mov %esp,%ebp
0x00001214 <+7>:   push %ebx
0x00001215 <+8>:   sub $0x24,%esp
0x00001218 <+11>:  call 0x12b4 <__x86.get_pc_thunk.ax>
0x0000121d <+16>:  add $0x2db3,%eax
0x00001222 <+21>:  mov 0x8(%ebp),%edx
0x00001225 <+24>:  mov %edx,-0x1c(%ebp)
0x00001228 <+27>:  mov %gs:0x14,%ecx
0x0000122f <+34>:  mov %ecx,-0xc(%ebp)
0x00001232 <+37>:  xor %ecx,%ecx
0x00001234 <+39>:  sub $0x8,%esp
0x00001237 <+42>:  pushl -0x1c(%ebp)
0x0000123a <+45>:  lea -0x18(%ebp),%edx
0x0000123d <+48>:  push %edx
0x0000123e <+49>:  mov %eax,%ebx
0x00001240 <+51>:  call 0x10a0 <strcpy@plt>
0x00001245 <+56>:  add $0x10,%esp
0x00001248 <+59>:  nop
0x00001249 <+60>:  mov -0xc(%ebp),%eax
0x0000124c <+63>:  xor %gs:0x14,%eax
0x00001253 <+68>:  je 0x125a <foo+77>
0x00001255 <+72>:  call 0x1340 <__stack_chk_fail_local>
0x0000125a <+77>:  mov -0x4(%ebp),%ebx
0x0000125d <+80>:  leave
0x0000125e <+81>:  ret
End of assembler dump.
```
Defeating StackGuard

Random canary:
- Random string **chosen at program startup**
- To corrupt, attacker must learn/guess current random string

Terminator canary:
- `{0, newline, linefeed, EOF}`
- String functions will not copy beyond terminator
- Attacker cannot use string functions to corrupt the stack
Defeating StackGuard

Random canary:
- Random string **chosen at program startup**
- To corrupt, attacker must learn/guess current random string

Terminator canary:
- `{0, newline, linefeed, EOF}`
- String functions will not copy beyond terminator
- Attacker cannot use string functions to corrupt the stack
StackGuard Variations

- Rearrange stack layout to prevent ptr overflow.

Stack Growth

- String Growth
  - local non-buffer variables
  - local string buffers
  - CANARY

Stack Growth

- copy of pointer args
- ret addr
- args

Protects pointer args and local pointers from a buffer overflow, but no arrays.
StackGuard Variations

char buffer

saved ebp

saved eip/return address

args

exception handler pointer

Buffer Growth

esp = 0xfffffcd4e

ebx = 0xfffffcd6c

ebp = 0xfffffcd80

Stack Growth

exception handler code
....
call %ebx
....
PointGaurd

• Insight:
  • pointers in memory corrupted via overflow
  • pointers in registers are not overflowable

• Solution:
  • Store pointers encrypted in memory
  • To dereference a pointer: decrypt it as you load it unto a register
Normal Pointer Dereference

1. Fetch Pointer Value
2. Access data referenced by pointer
Normal Pointer Dereference under attack
PointerGuard Pointer Dereference

1. Fetch Pointer Value
2. Access data referenced by pointer

CPU

Memory

0x7239

Encrypted Pointer

0x1234

0x1234

Data
PointerGuard Pointer Dereference Under Attack
Formal Verification
Approaches for Ensuring Memory Safety

• How do we reason about the code to get some confidence that the result will be memory safe?

• Approach using formal mathematical logic, induction to verify that your code is memory safe.

**GOAL:** You shouldn't have to know what the code inside the function is, the details of how it works is secondary, the pre- and post-conditions should be sufficient.

**General correctness proof strategy for memory safety:**
• Identify each point of memory access
• Write down precondition that it requires
• Propagate that requirement up to the beginning of the function
Setting up pre-and post-conditions

Going through our code base, one function at a time we are specifying the contract or API for each function. Also known as contract-based coding.

Pre-conditions
• When a function is invoked, and before it starts executing, the properties of the input variables, that need to be true for the function execution to be memory safe.
• caller's responsibility to setup

Post-conditions
• the function assumes the caller has setup the pre-conditions correctly, then the post-conditions are what should hold after the function finishes executing.
• post-conditions are guarantees the function provides about the return value or the results of the computation
Setting up pre-and post-conditions

```c
int deref(int *p){
    return *p;
}
```

Pre-conditions
- When a function is invoked, and before it starts executing, the properties of the input variables, that need to be true for the function execution to be memory safe.
- **caller's responsibility to setup**

Post-conditions
- the function **assumes** the caller has setup the pre-conditions correctly, then the post-conditions are what should hold after the function finishes executing.
- post-conditions **are guarantees** the function provides about the return value or the results of the computation
Setting up pre-and post-conditions

/* requires: p! = NULL and p as a valid pointer */

int deref(int *p){
    return *p;
}

Pre-conditions
• When a function is invoked, and before it starts executing, the properties of the input variables, that need to be true for the function execution to be memory safe.
• caller's responsibility to setup

Post-conditions
• the function assumes the caller has setup the pre-conditions correctly, then the post-conditions are what should hold after the function finishes executing.
• post-conditions are guarantees the function provides about the return value or the results of the computation
Setting up pre- and post-conditions

/* ALTERNATE IMPLEMENTATION
requires: p as a valid pointer */

int deref(int *p) {
    if (p != NULL) {
        return *p;
    }
}  

Pre-conditions
• When a function is invoked, and before it starts executing, the properties of the input variables, that need to be true for the function execution to be memory safe.
• caller's responsibility to setup

Post-conditions
• the function assumes the caller has setup the pre-conditions correctly, then the post-conditions are what should hold after the function finishes executing.
• post-conditions are guarantees the function provides about the return value or the results of the computation
Setting up pre-and post-conditions

/* requires:
   Ensures: */

```c
void *mymalloc (unsigned int n){
    void *p = malloc(n);
    if (!p){
        perror("malloc");
        exit(1);
    }
    return p;
}
```

**Pre-conditions**
- When a function is invoked, and before it starts executing, the properties of the input variables, that need to be true for the function execution to be memory safe.
- **caller's responsibility to setup**

**Post-conditions**
- the function *assumes* the caller has setup the pre-conditions correctly, then the post-conditions are what should hold after the function finishes executing.
- post-conditions *are guarantees* the function provides about the return value or the results of the computation.
Setting up pre-and post-conditions

/*
    Ensures: return value != NULL
    (and a valid pointer)
*/
void *mymalloc (unsigned int n){
    void *p = malloc(n);
    if (!p){
        //code checks if malloc returns with valid pointer
        perror("malloc");
        exit(1);
    }
    return p;
}

Pre-conditions
• When a function is invoked, and before it starts executing, the properties of the input variables, that need to be true for the function execution to be memory safe.
• caller's responsibility to setup

Post-conditions
• the function assumes the caller has setup the pre-conditions correctly, then the post-conditions are what should hold after the function finishes executing.
• post-conditions are guarantees the function provides about the return value or the results of the computation