Lecture 10: Buffer Overflows (cont'd)
Review: Buffer Overflow Attack

- Idea: overwrite return address with address of instruction you want to execute next
  - If a string: use padding to fill up space between array and saved rip
  - Stack smashing: use padding to write program and jump there

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq $0x18, %rsp
    movq %rsp, %rdi
    call gets
    call puts
    addq $0x18, %rsp
    ret
```
Review: Buffer Overflow Examples
Defense #1: Avoid Overflow Vulnerabilities

For example, use library routines that limit string lengths

- fgets instead of gets
- strncpy instead of strcpy
- Don’t use scanf with %s conversion specification (use fgets to read the string or use %ns where n is a suitable integer)

Or use a high-level language

/* Echo Line */
void echo()
{
    char buf[4]; /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
Buffer Overflow Vulnerabilities
Defense #2: Compiler checks

• Idea
  • Place special value ("canary") on stack just beyond buffer
  • Check for corruption before exiting function

• GCC Implementation
  • \texttt{-fstack-protector}
  • Now the default (disabled earlier)
Stack Canaries

**Stack Frame for call_echo**

```
00 00 00 00
00 40 06 f6
33 32 31 30
39 38 37 36
35 34 33 32
31 30 39 38
37 36 35 34
33 32 31 30
```

- **saved %rip**
- **canary**
- **buf ← %rsp**

**authenticate:**
```
pushq %rbx
subq $16, %rsp
movq %rdi, %rbx
movq %fs:40, %rax
movq %rax, 8(%rsp)
xorl %eax, %eax
movq %rsp, %rdi
call gets
movq %rsp, %rsi
movq %rbx, %rdi
call strcmp
testl %eax, %eax
sete %al
movq 8(%rsp), %rdx
xorq %fs:40, %rdx
je .L2
call __stack_chk_fail
```

**.L2:**
```
movzbl %al, %eax
addq $16, %rsp
popq %rbx
ret
```
Exercise 1: Stack Canaries

Which of the following would make a good stack canary?

1. A secret, constant value
2. A fixed sequence of common terminators (\0, EOF, etc.)
3. A random number chosen each time the program is run
Review: Stack Smashing

- Idea: fill the buffer with bytes that will be interpreted as code
- Overwrite the return address with address of the beginning of the buffer

```
/* Echo Line */
void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}
```

echo:
    subq $18, %rsp
    movq %rsp, %rdi
    call gets
    call puts
    addq $18, %rsp
    ret

Stack Frame for call_echo

| saved %rip | 7f ff ff ff |
| 40 ea 40 33 |
| 30 39 38 37 |
| 36 35 34 33 |
| 32 31 30 39 |
| 38 37 36 35 |
| 34 33 32 31 |
| 30 29 28 27 |
| 26 25 24 23 |

%rsp -> buf
Defense #3: Memory Tagging

- \( W \oplus X \)

Memory Tagging diagram:

- Stack: 0x7FFF (NX)
- Heap: 0x0000 (NX)
- Data: 0x0000 (NX)
- Code: 0x0000 (Nw)
Code Reuse Attacks

• Key idea: execute instructions that already exist

• Defeats memory tagging defenses

• Examples:
  1. return to a function or line in the current program
  2. return to a library function (e.g., return-into-libc)
  3. return to some other instruction (return-oriented programming)
Handling Arguments

what function expects when it is called…

overflow with argument
## Return-into-libc

<table>
<thead>
<tr>
<th>Sr.No.</th>
<th>Function &amp; Description</th>
</tr>
</thead>
</table>
| 1      | **double atof(const char *str)**
|        | Converts the string pointed to, by the argument *str* to a floating-point number (type double). |
| 2      | **int atoi(const char *str)**
|        | Converts the string pointed to, by the argument *str* to an integer (type int). |
| 3      | **long int atol(const char *str)**
|        | Converts the string pointed to, by the argument *str* to a long integer (type long int). |
| 8      | **void free(void *ptr)**
|        | Deallocates the memory previously allocated by a call to calloc, malloc, or realloc. |
| 9      | **void *malloc(size_t size)**
|        | Allocates the requested memory and returns a pointer to it. |
| 10     | **void *realloc(void *ptr, size_t size)**
|        | Attempts to resize the memory block pointed to by ptr that was previously allocated with a call to malloc or calloc. |
| 15     | **int system(const char *string)**
|        | The command specified by string is passed to the host environment to be executed by the command processor. |
| 16     | **void *bsearch(const void *key, const void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))**
|        | Performs a binary search. |
| 17     | **void *qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *))**
|        | Sorts an array. |
| 18     | **int abs(int x)**
|        | Returns the absolute value of *x*. |
| 22     | **int rand(void)**
|        | Returns a pseudo-random number in the range of 0 to RAND_MAX. |
| 23     | **void srand(unsigned int seed)**
|        | This function seeds the random number generator used by the function rand. |
ASCII Armoring

- Make sure all system library addresses contain a null byte (0x00).
- Can be done by placing this code in the first 0x01010101 bytes of memory
Properties of x86 Assembly

- lots of instructions
- variable length instructions
- not word aligned
- dense instruction set
Gadgets

```c
void setval(unsigned *p) {
    *p = 3347663060u;
}
```

<setval>:
```
4004d9: c7 07 d4 48 89 c7 movl $0xc78948d4,(%rdi)
4004df: c3 ret
```

gadget address: 0x4004dc
encodes: movq %rax, %rdi
executes: %rdi <- %rax
Example Gadgets

Load Constant

Load from memory

0xbad01010

5a c3

popq %rdx
ret

0xbad00002

48 89 C0 C3

movq (%rax), %rax
ret

58 c3

popq %rax
ret
Return-oriented Programming is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.
Return-oriented Programming

Final ret in each gadget sets pc (%rip) to beginning of next gadget code
Return-Oriented Shellcode
Exercise 2: ROP

- What are the values in the registers when the function at address 0x401a82 gets called?

<table>
<thead>
<tr>
<th>%rip</th>
<th>%rdi</th>
<th>%rsi</th>
<th>%rax</th>
<th>%rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
0x401a82
0x401a82
0x402a3
0x401bb0
0x7fffffffdea50
0x7fffffffdea58
0x7fffffffdea50
0x7fffffffdea38
0x7fffffffdea30
0x7fffffffdea28
0x7fffffffdea20
```
Exercise 2: ROP

- What are the values in the registers when the function at address 0x401a82 gets called?

<table>
<thead>
<tr>
<th>%rip</th>
<th>%rdi</th>
<th>%rsi</th>
<th>%rax</th>
<th>%rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>401a82</td>
<td>2f</td>
<td></td>
<td>2f</td>
<td>..ea58</td>
</tr>
</tbody>
</table>

```
7fffffff97e9f
7fffffff97e98
7fffffff97e90
7fffffff97e88
7fffffff97e80
7fffffff97e78
7fffffff97e70
7fffffff97e68
7fffffff97e60
7fffffff97e58
7fffffff97e50
7fffffff97e48
7fffffff97e40
7fffffff97e38
7fffffff97e30
7fffffff97e28
7fffffff97e20
0x59b997d0
0x401a82
0x4002a3
0x401b0
0x7fffffff97e50
0x401a5b
0x59b997ff
0x4019e5
```

![Diagram showing the execution flow of the function with register values and memory addresses.](image-url)
Address Space Layout Randomization
Other defenses

Gadget Elimination

Control Flow Integrity

Code

Code

(c)
The state of the world

Defenses:

- high-level languages
- Stack Canaries
- Memory tagging
- ASLR
- continuing research and development…

But all they aren't perfect!
Exercise 3: Feedback

1. Rate how well you think this recorded lecture worked
   1. Better than an in-person class
   2. About as well as an in-person class
   3. Less well than an in-person class, but you still learned something
   4. Total waste of time, you didn't learn anything

2. How much time did you spend on this video lecture (including time spent on exercises)?

3. Do you have any questions that you would like me to address in this week's problem session?

4. Do you have any other comments or feedback?