Lecture 9: Machine Programming: Advanced aka Security **15-213/15-513/14-513**
Fall 2022

1 Getting Started

Let's look back at the example in Lecture 1:

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

Problem 1. Where is the struct allocated in the system?

The struct is allocated on the stack, as a local variable.

Problem 2. What critical value is pushed onto the stack for every function call?

Each `call` instruction pushes the (correct) return address onto the stack.

Problem 3. The `s.a[i]` line is writing to memory. In lecture 1, `fun(6)` crashed the program. Why did writing to this location cause the process to crash?

Writing to `s.a[6]` overwrote the return address for `fun` with a nonsense value, causing the `ret` instruction to transfer control to a memory location that was either unmapped, non-executable, or contained invalid machine instructions.

2 Gets

There are many routines in C that take in user input. We will briefly explore one of the simplest, `gets()`, and how this routine has security vulnerabilities.

```
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    // EOF indicates there is no further input.
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

Problem 4. In C, do we know how much space has been allocated to `*dest`?

We don't!

Problem 5. Given that `getchar()` reads one character from stdin, when does `gets()` stop reading in input?

When it encounters a newline or the end of the user input stream.

Problem 6. Do the terminating conditions for `gets()` have any relation to the input buffer?

No, they purely depend on how many characters the user inputs.

3 Overwriting Stack

In the following section, we will be investigating how the following routine, intended for echoing user input back to the screen (using `puts` to print the string), may exhibit unintended behavior.

```
00000000004006cf <echo>:
 4006cf:       48 83 ec 18            sub    $0x18,%rsp
 4006d3:       48 89 e7               mov    %rsp,%rdi
 4006d6:       e8 a5 ff ff ff         call   400680 <gets>
 4006db:       48 89 e7               mov    %rsp,%rdi
 4006de:       e8 3d fe ff ff         call   400520 <puts>
 4006e3:       48 83 c4 18            add    $0x18,%rsp
 4006e7:       c3                     ret
```

Problem 7. How much space is allocated on the stack in `echo()`?

`0x18` = 24 bytes are allocated.

Problem 8. Assume `echo()` allocates a char buffer on the stack. What is the largest that this buffer could have been, in the C source code?

The buffer was, at most, 24 bytes long. It could have been smaller, because the compiler is obliged to keep the stack pointer *aligned* and this involves rounding up the amount of space requested. The user may or may not enter a string that is shorter than 24 bytes.

Problem 9. In the following stack diagram, label the type of each region of memory or leave it blank if it is unknown. Before the call to `gets()`, `%rsp` has the value `0x414140`.

`+0x28`	??	
`+0x20`	??	
`+0x18`	return address	
`+0x10`	user string	
`+0x08`	user string	
`+0x00`	user string	← `$rsp = 0x414140`
	...	

Problem 10. While the `echo()` function is running, you type in the following string:

`123456781234567812345678@AA`

Fill in the following stack diagram with the new values. (Use the characters from the string, except use _ to indicate a byte that is unchanged and `0` to indicate a byte with numeric value zero.)

`+0x28`	??	
`+0x20`	??	
`+0x18`	`@AA0____`	
`+0x10`	`12345678`	
`+0x08`	`12345678`	
`+0x00`	`12345678`	← `$rsp = 0x414140`
	...	

4 Exploit

Note, while the compiled code must follow the Linux x64 ABI, that is not a requirement of the system and exploits can ignore these conventions to achieve the desired behavior. We will see how the simple stack overwrite from the previous section can do more than just crash the program.

Problem 11. Based on the previous stack diagram and input, and assuming that any bytes of the return address that were *not* overwritten by `gets` were zero to begin with, write down the flow of execution starting with the return from `gets` in `echo`. Note: ASCII encodes `@` as `0x40` and `A` as `0x41`. Watch out for endianness!

Starting from `echo()`'s call to `gets()`, at `0x4006d6`:

```
0x4006d6 -> 0x4006db (mov) -> 0x4006de (puts) -> 0x4006e3 (add) ->
0x4006e7 (ret) -> [USER INPUT] 0x414140 (???)
```

Problem 12. When does this flow differ from the original execution path?

When control was going to be returned to `echo()`'s caller, control was instead transferred to user input on the stack.

The exploit string "returned" execution to the stack. Currently, execution would attempt to use the ASCII string as instructions.

Problem 13. Suppose you wanted to make `0xdecafbad` the return value. Write the assembly to do so.

```
mov $decafbad, %eax
```

Problem 14. The bytes for the instruction(s) you need are: `0xb8 0xad 0xfb 0xca 0xde`. Where would those bytes be placed in our input string to execute them?

They should replace the first five characters of the input string.

Processes include a copy of the C standard library, which contains the functionality to open a shell and thereby run arbitrary code. Most exploits, then, aim to invoke the appropriate function. In our example, we took the first step in running arbitrary code.

5 Defense

System designers have come up with many countermeasures to reduce program vulnerabilities. We will review several of these approaches here.

Problem 15. `fgets()` has the following type signature; why is it safer than `gets()`?

```
    char *fgets(char *s, int size, FILE *stream);
```

The `size` parameter limits the number of bytes that are read from the given stream. As long as the buffer pointed to by the `s` parameter is at least `size` bytes large, `fgets` will not overwrite any other data.

Problem 16. Previously, the last characters of the exploit string were carefully chosen. What would happen if the stack were somewhere else in memory?

Execution would jump to an unknown section of memory, almost certainly causing a crash as in problem 3.

Problem 17. The OS decides where to place the stack in memory. How could we minimize the chance that the exploit guessed the right address?

Choose the starting address of the stack at random when each new program is started.

So far, the program has trusted that its stack is uncorrupted. The compiler can put a "canary" value on the stack to detect if the stack is modified. The following assembly includes the canary instructions, denoted by *.

The register `fs` is special. It always holds a pointer, and it is accessed differently than other registers. The assembly operand `%fs:0x28` means the same thing as `0x28(%fs)` would if `fs` were a normal register (but it isn't, and you can't write it that way).

The function `__stack_chk_fail` terminates the program.

```
  40072f:     sub    $0x18,%rsp
* 400733:     mov    %fs:0x28,%rax
* 40073c:     mov    %rax,0x8(%rsp)
* 400741:     xor    %eax,%eax
  400743:     mov    %rsp,%rdi
  400746:     call   4006e0 <gets>
  40074b:     mov    %rsp,%rdi
  40074e:     call   400570 <puts@plt>
* 400753:     mov    0x8(%rsp),%rax
* 400758:     xor    %fs:0x28,%rax
* 400761:     jz     400768 <echo+0x39>
* 400763:     call   400580 <__stack_chk_fail>
```

```
400768:     add    $0x18,%rsp
40076c:     ret
```

Problem 18. Write pseudo-code for the canary instructions.

```
// ... echo starts up...
rax = *(fs + 0x28);
*(rsp + 8) = rax;
rax = 0;
// ...echo's code...
rax = *(rsp + 8)
if (rax != *(fs + 0x28)
    __stack_chk_fail(); // does not return
// ... echo returns ...
```

Problem 19. What would happen if we used our earlier exploit string with this revised assembly?

We would overwrite `*(rsp + 8)`, causing `__stack_chk_fail` to be called, and the program would terminate before it could jump to our exploit.

Problem 20. (Optional) What costs (memory, time, instructions, etc.) are associated with this revised assembly?

Each function that uses a "canary" is larger (in the example, `echo` grew by 37 bytes) and slightly slower. The compiler has extra work to do, to decide which functions need "canaries" and emit the extra instructions for them.

6 ROP

64-bit x86 adds a no-execute bit that is commonly applied to the stack. However, the stack is still writable by user input, and the process must still allow the C standard library to be executable.

The new restrictions have led to the increased prevalence of return-oriented programming (ROP) attacks, in which the attacker searches the executable for preexisting "gadgets" with parts of the desired assembly sequence.

Problem 21. How did we initially transfer control to our code in the exploit before?

We overwrote the return address for `echo`'s caller before executing `ret`.

Problem 22. In the following assembly, the middle section shows the machine code for the assembly sequence to its right.

```
<setval>:
  4004d0:  c7 07 d4 48 89 c7    movl $0xc78948d4,(%rdi)
  4004d6:  c3                   ret
```

What byte value corresponds to a return instruction?

`0xC3`

Problem 23. `mov %rax, %rdi` is encoded as `48 89 c7`. At what address does that byte sequence exist in the following assembly?

Address `0x4004d3` (*inside* a different instruction).

Problem 24. If the value on the top of the stack was the address in the previous question, and a return instruction was executed, describe the steps of the subsequent execution.

```
mov %rax, %rdi
ret
; ...instructions starting at the second next stack address
; prior to running this code block...
```

Problem 25. Why is it important that each gadget ends with a `ret` instruction?

If there is no return instruction at the end of the gadget, execution will never jump to the next gadget in the chain.