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?

Stack as it is a local variable

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

The return address from %rip is pushed to stack with every call instruction

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?

Because s.a[6] is outside the size of the array. It accessed other parts of the stack.

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.

```c
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`?

No. We only get the starting address of *dest.

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

Stop reading when user enter EOF or \n

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

No. Input can be any size and there is no checking

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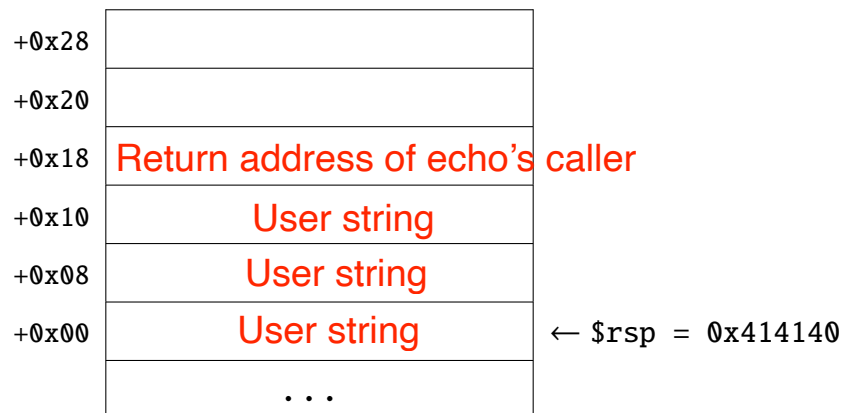
Problem 7. How much space is allocated on the stack in `echo()`?

<p style="color:red; text-align:center">24 bytes</p>

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?

<p style="color:red">23 bytes. That means 23 characters plus '\0' at the end. char buf[23]</p>
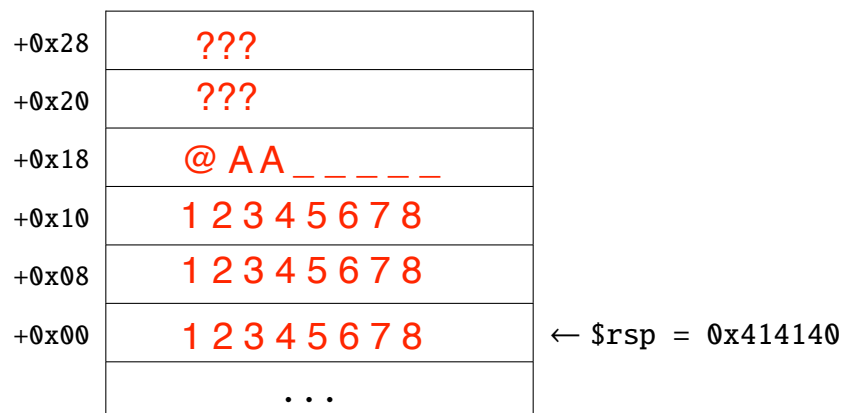
Problem 9. In the following stack diagram, label the type of each region of memory or leave it blank if it is unknown. Before the call to `gets()`, `%rsp` has the value `0x414140`.

+0x28	
+0x20	
+0x18	Return address of echo's caller
+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	@ A A _ _ _ _ _
+0x10	1 2 3 4 5 6 7 8
+0x08	1 2 3 4 5 6 7 8
+0x00	1 2 3 4 5 6 7 8 ← $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!

0x4006db (mov)   0x4006de (puts)   0x4006e3 (add)

0x4006e7 (ret)

Problem 12. When does this flow differ from the original execution path?
The original return address changed to 0x414140
When control is going to return to echo's caller, it will instead be
transfered to the start of user input
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?

Input string (first five characters)
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);
```

It specifies the size of the buffer and
user cannot input string longer than that

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

Jump to an unknown address and cause crash

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?

Randomise the stack address for each execution

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.

Problem 19. What would happen if we used our earlier exploit string with this revised assembly?   Program would terminate if it tries to go beyond the canary check

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

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?

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?

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

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.

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