# 15213 Lecture 9: Advanced aka Security

## 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;
}
```

1. Where is the struct allocated in the system?
2. What critical value is pushed onto the stack for every function call?
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?

## 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;
}
```

1. In C, do we know how much space has been allocated to `*dest`?
2. Given that `getchar()` reads one character from stdin, when does `gets()` stop reading in input?
3. Do the terminating conditions for `gets()` have any relation to the input buffer?
4. What is another C routine that accepts buffer(s) and not their size(s)?

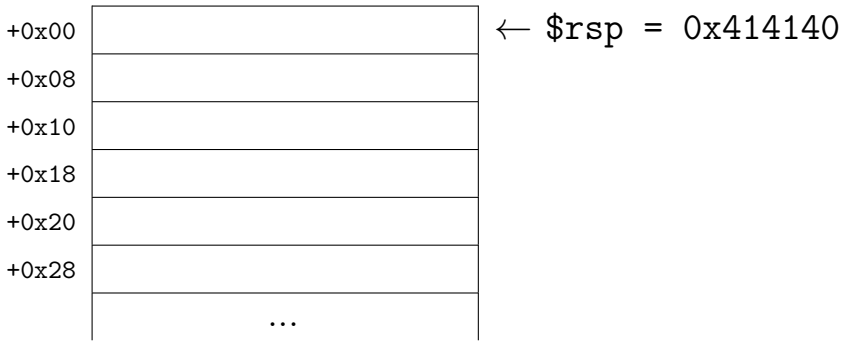
## 3 Overwriting Stack

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

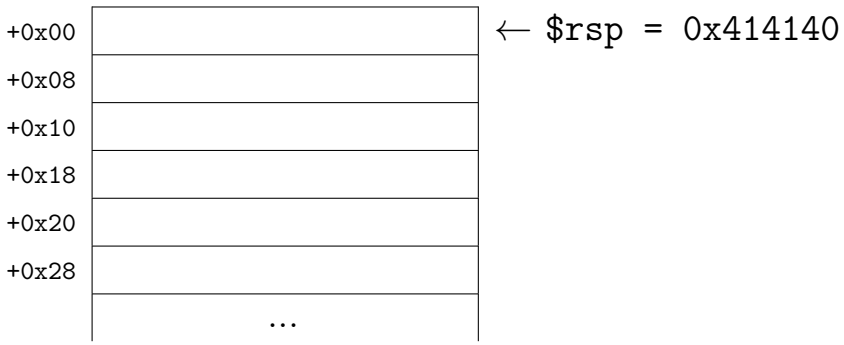
```
00000000004006cf <echo>:
 4006cf: 48 83 ec 18           sub    $0x18,%rsp
 4006d3: 48 89 e7              mov    %rsp,%rdi
 4006d6: e8 a5 ff ff ff        callq  400680 <gets>
 4006db: 48 89 e7              mov    %rsp,%rdi
 4006de: e8 3d fe ff ff        callq  400520 <puts>
 4006e3: 48 83 c4 18  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

1. Given that `echo()` allocates a char buffer on the stack, what is the largest that the buffer may be legitamately sized by the compiler?

2. 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`.




3. While the `echo()` function is running, you type in the following string:
`1234567812345678123456780AA\0`
Fill in the following stack diagram with the new values.



## 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.

1. Based on the previous stack diagram and input, draw the flow of execution starting with the return from `gets()` in `echo()`.
   (You can use the ASCII values (`1=0x31,@=0x40,A=0x41`).)

2. When does this flow differ from the naive execution path?

The exploit string "returned" execution to the stack. Currently, execution would attempt to use the ASCII string as instructions (the first of which would be `xor %esi,(%rdx)`).

1. Suppose you wanted to make the `0xdecafbad` the return value, write the assembly to do so.

2. The bytes for that instruction are: `0xb8 0xad 0xfb 0xca 0xde`. Where would those bytes be placed in our input string to execute them?

Processes include a copy of the C standard library. That contains functionality to open a shell and thereby run arbitrary code. So most exploits are trying to invoke the appropriate function. In this case, we took the first step in running arbitrary code.

1. (Advanced) We made a simplifying assumption regarding the exploit's address. What is it?

## 5 Defense

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

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

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

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

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

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

The register `%fs` is special and always holds a pointer.

```
  40072f: sub    $0x18,%rsp
* 400733: mov    %fs:0x28,%rax
* 40073c: mov    %rax,0x8(%rsp)
* 400741: xor    %eax,%eax
  400743: mov    %rsp,%rdi
  400746: callq  4006e0 <gets>
  40074b: mov    %rsp,%rdi
  40074e: callq  400570 <puts@plt>
* 400753: mov    0x8(%rsp),%rax
* 400758: xor    %fs:0x28,%rax
* 400761: jz     400768 <echo+0x39>
* 400763: callq  400580 <__stack_chk_fail>
  400768: add    $0x18,%rsp
  40076c: retq
```

1. Write pseudo-code for the canary instructions.

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

3. (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 C standard library must be executable in the process.

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

1. How did we initially transfer control to our code in the exploit before?
2. In the following assembly, the middle section shows the machine code for the corresponding assembly sequence. What byte value corresponds to a return instruction?
3. `movq %rax, %rdi` is encoded as `48 89 c7`. At what address does that byte sequence exist in the following assembly?

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

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

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