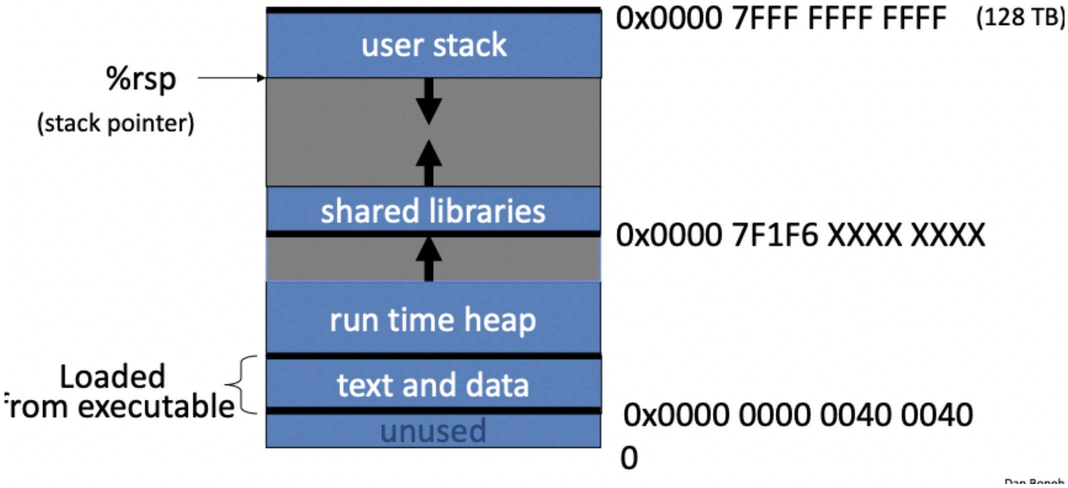
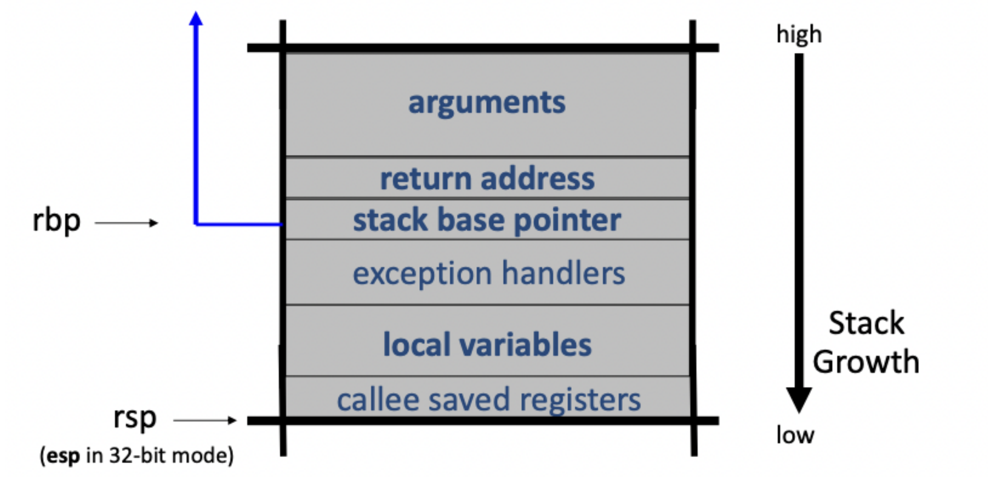
CAS CS 357

InClass Note 21

1. Control hijacking attacks
    a. Attacker’s goal: take over target machine (e.g. web server)
    b. Execute arbitrary code on target by hijacking application control flow
    c. Examples:
        - Buffer overflow and integer overflow attacks
        - Format string vulnerabilities
        - Use after free
2. First example: buffer overflow
    a. Extremely common bug in C/C++ programs
    b. First major exploit: 1988 Internet Worm, Fingerd
3. What is needed
    a. Understanding C functions, the stack, and the heap
    b. Know how system calls are made
    c. The exec() system call
    d. Attacker needs to know which CPU and OS used on the target machine:
        - Our examples are for x86 running Linux or Windows
        - Details vary slightly between CPUs and OSs
4. Linux process memory layout

a. 

5. Stack Frame

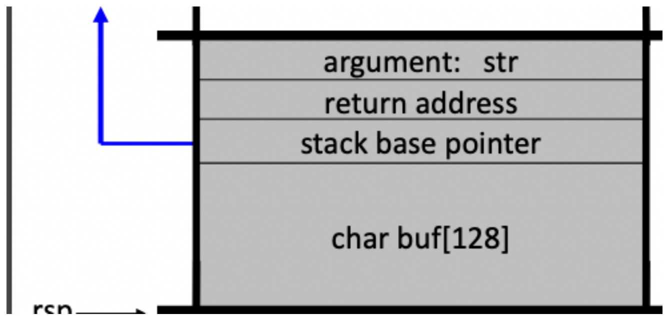
a. 

6. What are buffer overflows?

a. Suppose a web server contains a function:

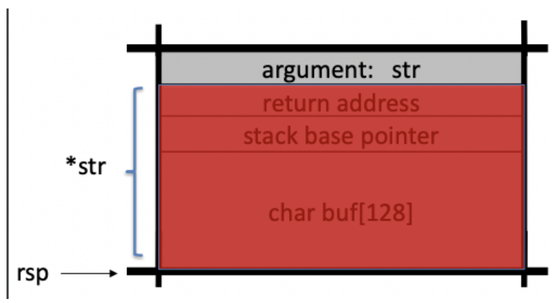
b.
```
void func(char *str) {
    char buf[128];

    strcpy(buf, str);
    do-something(buf);
}
```

c. After func() is called stack looks like:

d. 

e. What if *url is 144 bytes long?

f. After strcpy:

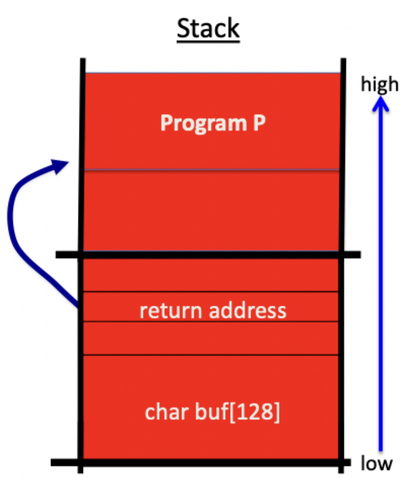
g. 

h. Poisoned return address!

i. Problem: no bounds checking in strcpy()

7. Basic stack exploit

a. Suppose *url is such that after strcpy stack looks like:

b. 

c. Program P: exec("/bin/sh")

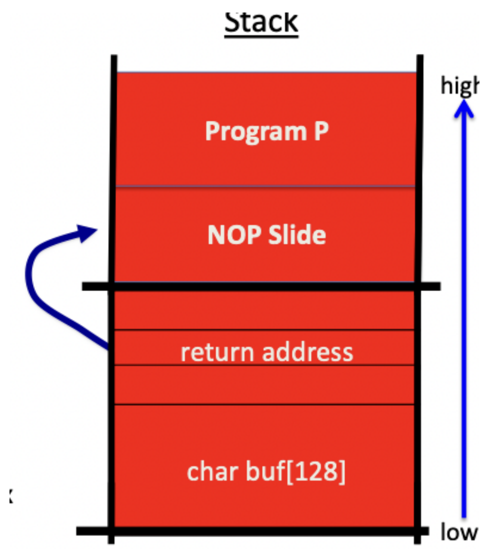
8. The NOP slide

    a. Problem: how does attacker determine ret-address?

    b. Solution: NOP slide

        - Guess approximate stack state when func() is called

        - Insert many NOPs before program P: nop, xor eax, eax, inc ax

    c. 

9. Details and examples

    a. Some complications:

        - Program P should not contain the '\0' character

        - Overflow should not crash program before func() exits

    b. Famous remote stack smashing overflows:

        - Overflow in Windows animated cursors (ANI)

        - Buffer overflow in Symantec virus detection

10. Many unsafe libc functions

    a. Strcpy (char *dest, const char *src)

    b. Strcat (char * dest, const char *src)

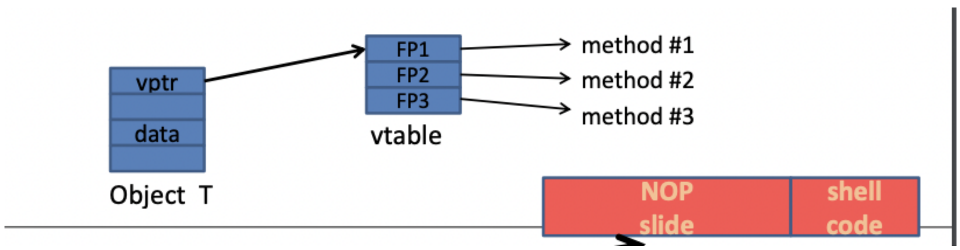
c. Gets (char *s)

d. Scanf (const char *format,...) and many more

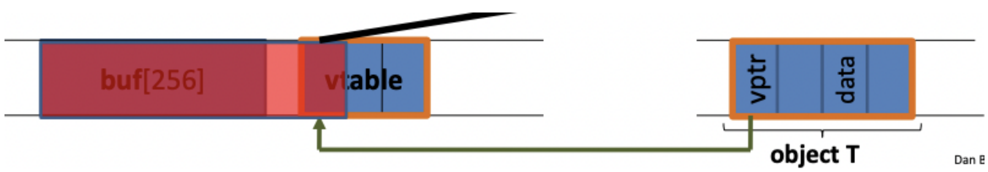
e. “Safe” libc versions strncpy(), strncat() are misleading

11. Heap exploits: corrupting virtual tables

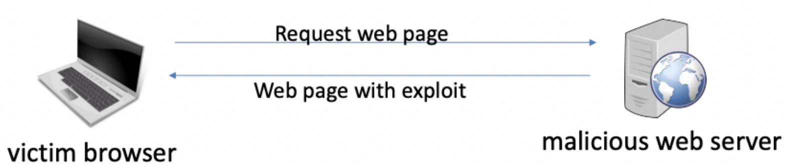
a. Compiler generated function pointers (e.g. C++ code)

b. 

c. After overflow of buf:

d. 

12. Exploiting the browser heap

a. 

b. Attacker’s goal is to infect browsers visiting the web site

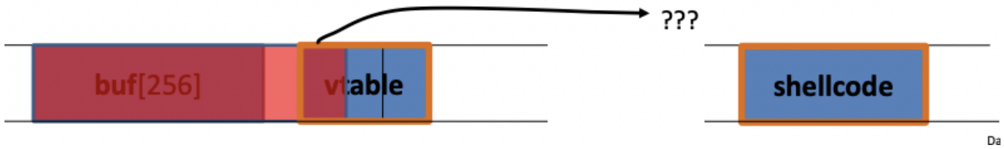
c. How: send javascript to browser that exploits a heap overflow

13. A reliable exploit?

a.

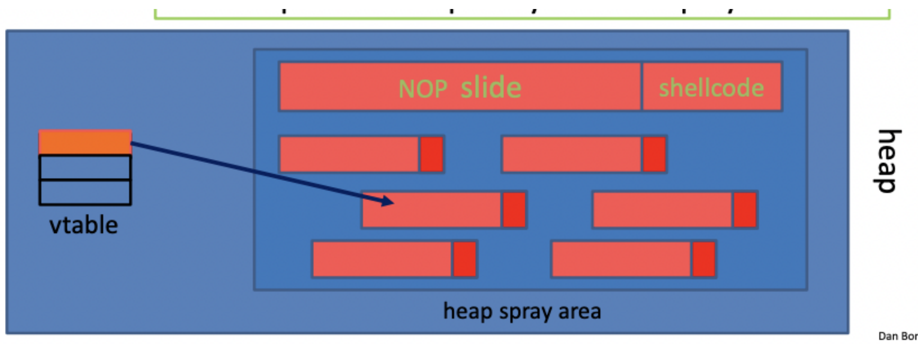
```
<SCRIPT language="text/javascript">
shellcode = unescape("%u4343%u4343%...");   // allocate in heap
overflow-string = unescape(“%u2332%u4276%...”);
cause-overflow(overflow-string );      // overflow  buf[ ]
</SCRIPT>
```

b. Problem: attacker does not know where browser places shellcode on the heap

c. 

14. Heap spraying

a. Idea:

- Use javascript to spray heap with shellcode (and NOP slides)
- Then point vtable ptr anywhere in spray area
- 

15. Javascript Heap spraying

a.

```
var  nop = unescape("%u9090%u9090")
while (nop.length < 0x100000)  nop += nop;

var shellcode = unescape("%u4343%u4343%...");

var x = new Array ()
for (i=0;  i<1000;  i++) {
        x[i] = nop + shellcode;
}
```

b. Pointing function-ptr almost anywhere in heap will cause shellcode to execute

16. More Hijacking opportunities

a. Integer overflows

b. Double free

c. Use after free

d. Format string vulnerabilities

17. Integer Overflows

a. Problem: what happens when int exceeds max value?

b. int m → 32 bits, short s → 16 bits, char c → 8 bits

c.

c = 0x80 + 0x80 = 128 + 128	⇒	c = 0
s = 0xff80 + 0x80	⇒	s = 0
m = 0xffffff80 + 0x80	⇒	m = 0

d. Example:

e.

```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
      char temp[256];
      if  (len1 + len2 > 256)   {return -1}               // length check
      memcpy(temp,  buf1,  len1);                         // cat buffers
      memcpy(temp+len1,  buf2,  len2);
      do-something(temp);                                 // do stuff
}
```

f.

What if **len1 = 0x80,** **len2 = 0xffffff80** ?

⇒ len1+len2 = 0

Second memcpy() will overflow heap !!

g. Example: better length check

h.

```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
      char temp[256];
      // length check
      if  (len1 > 256)  ||  (len2 > 256)  ||  (len1+ len2 > 256)
            return -1;
      memcpy(temp,  buf1,  len1);                         // cat buffers
      memcpy(temp+len1,  buf2,  len2);
      do-something(temp);                                 // do stuff
}
```

18. Format string bugs

   a. Format string problem

   b.
```
int func(char *user)  {
  fprintf(stderr, user);
}
```

   c. Problem: what if *user = “%s%s%s%s%s%s%s”??

      - Most likely program will crash: DoS
      - If not, program will print memory contents.
      - Full exploit using user = “%n”

   d. Correct form: fprintf(stdout, “%s”, user);

19. Vulnerable functions

   a. Any function using a format string

   b. Printing:

      - Printf, frpinrt, sprintf
      - vprintf,vfprintf,vsprintf

   c. logging:

      - Syslog, err, warn

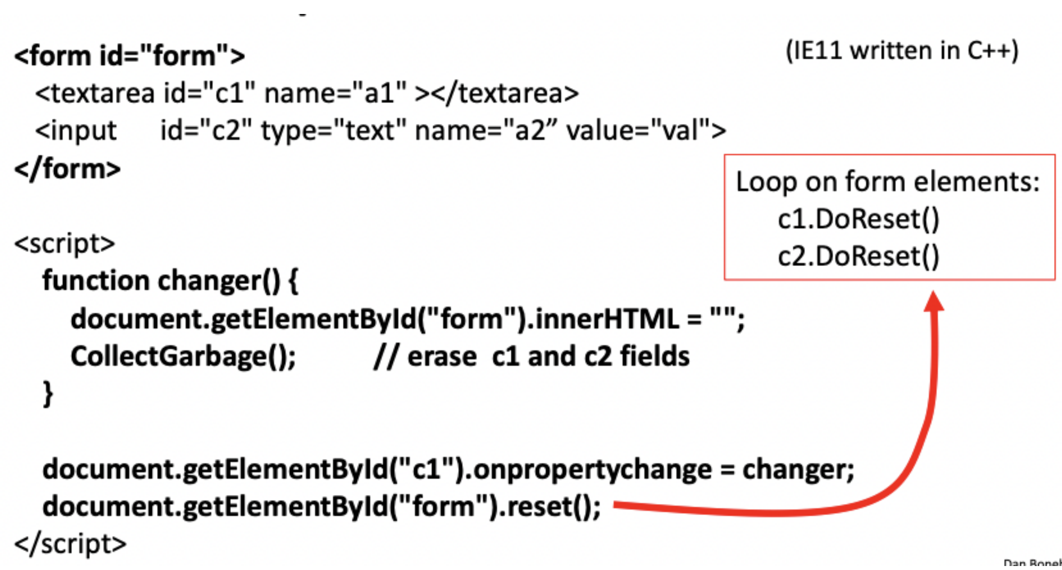
20. Exploit

   a. Dumping arbitrary memory:

   b. Writing to arbitrary memory

21. Use after free exploits

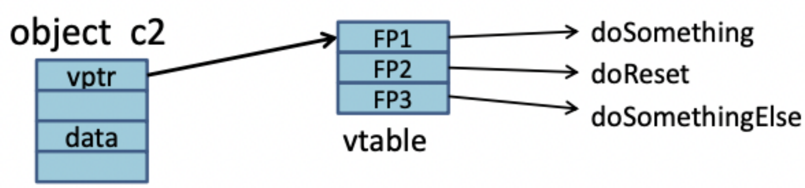
   a. 1E11 Example: CVE-2014-0282



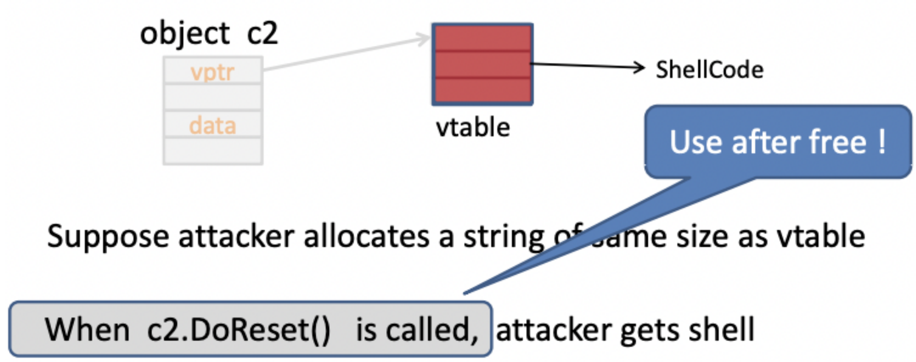
b.

22. What just happened?

a. c1.doReset() causes changer() to be called and free object c2



b.



c.