Information Security CS3002 (Sections BDS-7A/B) Lecture 16

Instructor: Dr. Syed Mohammad Irteza

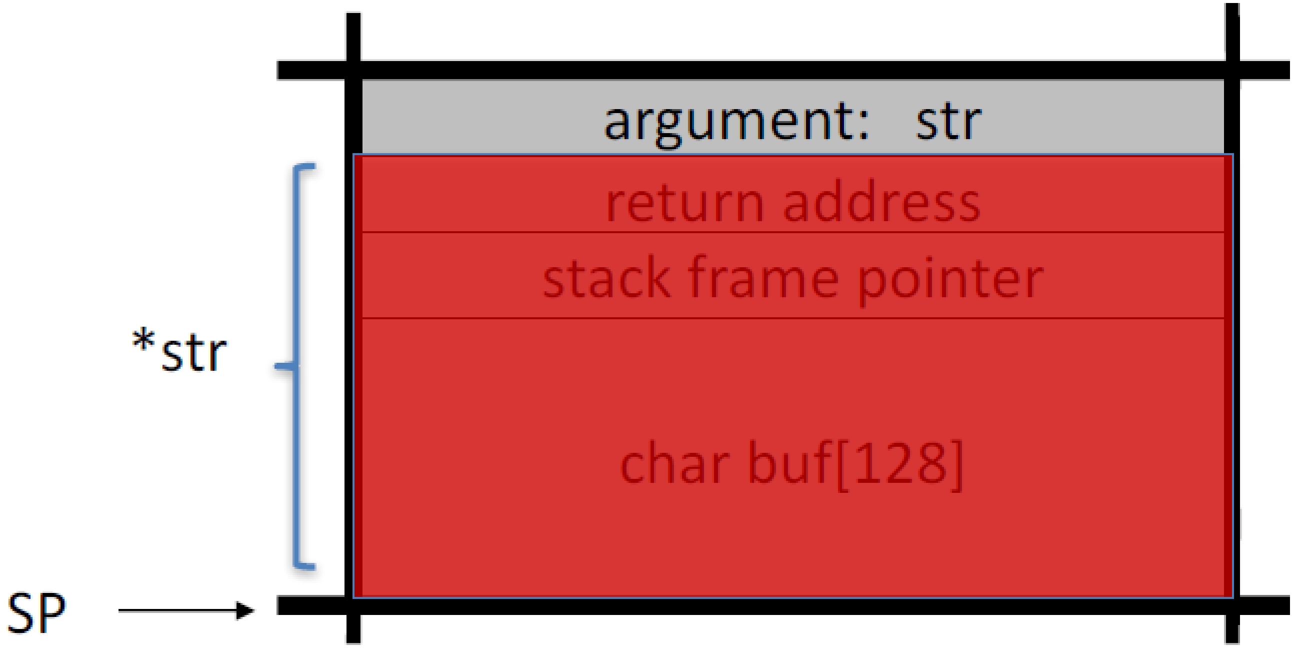
Assistant Professor, Department of Computer Science

14 October, 2024

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

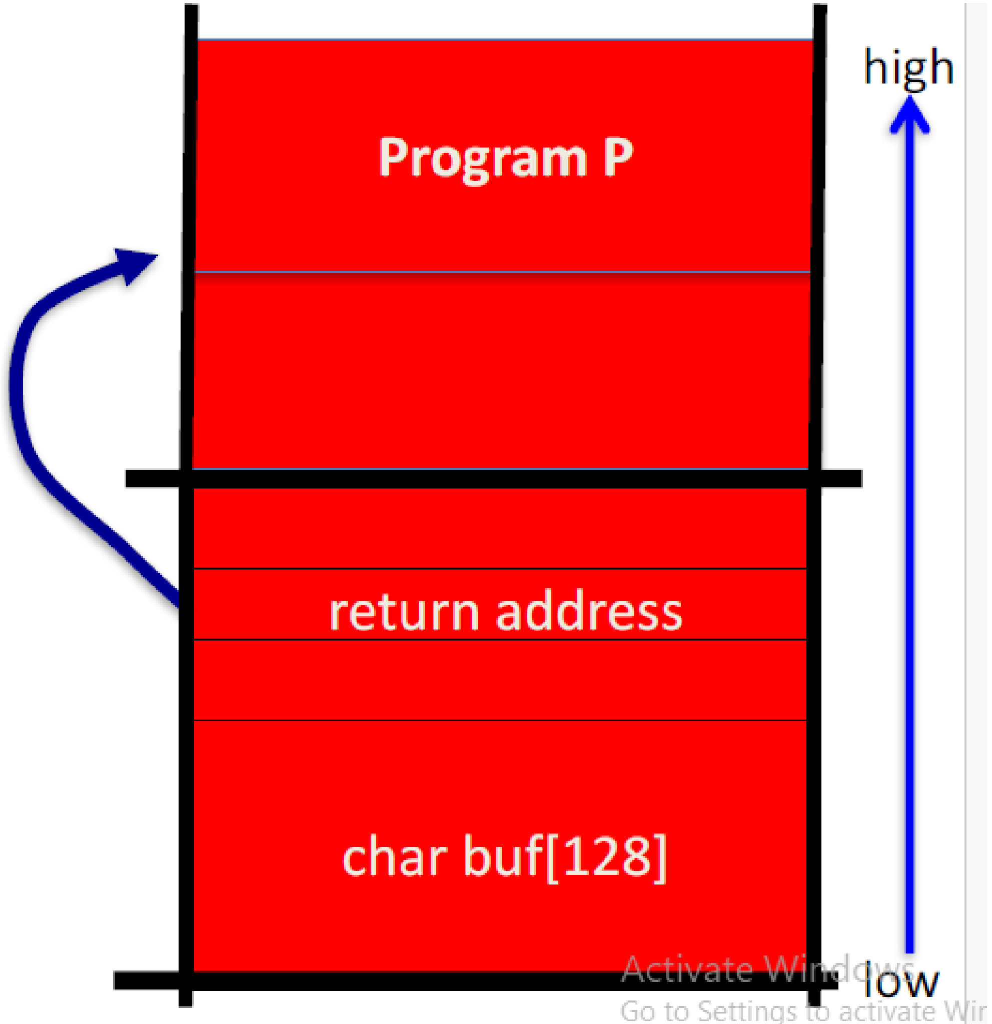
Suppose `*str` is such that after `strcpy` the stack looks like:

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*.



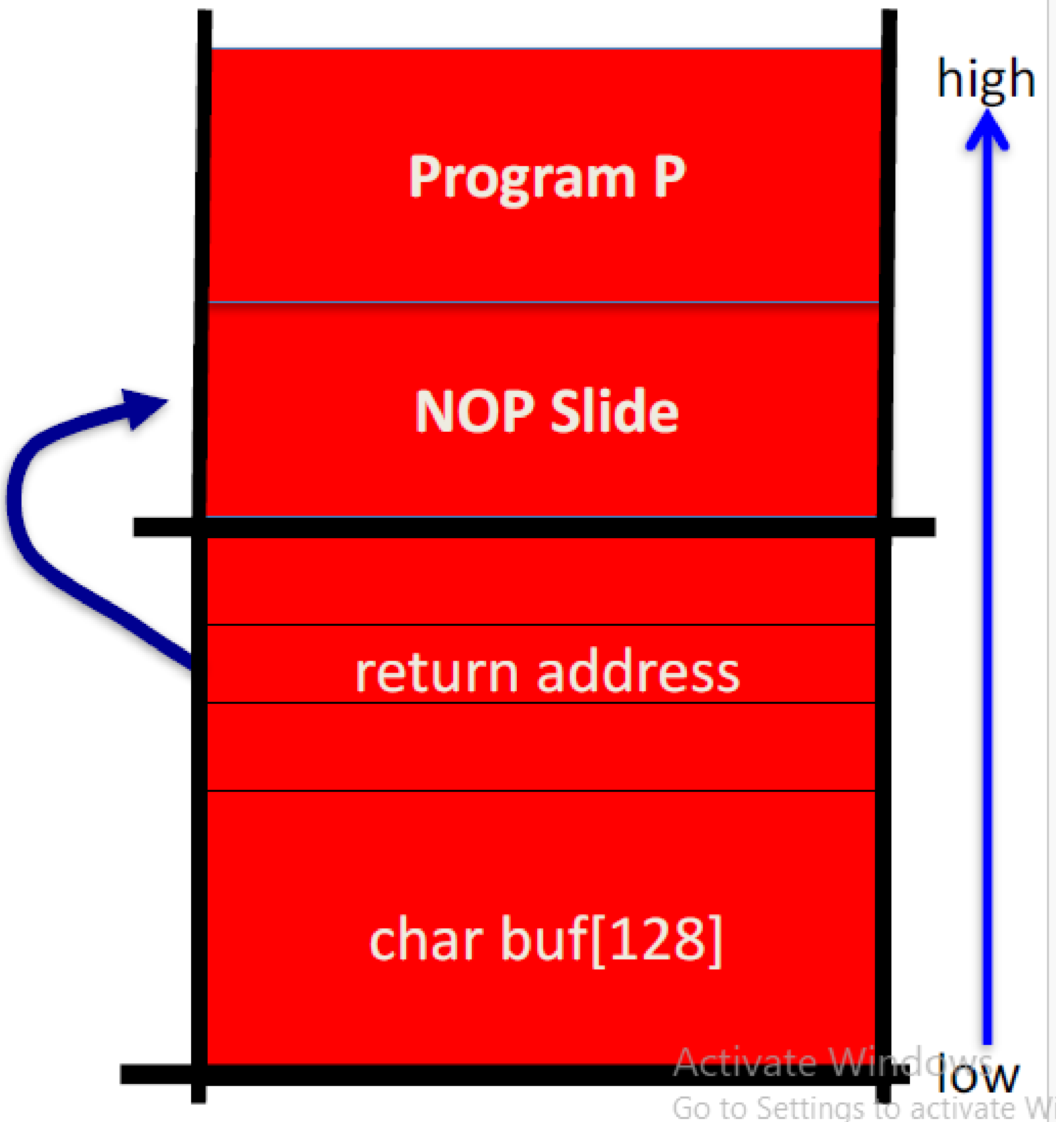
The NOP slide (NOP sled)

Problem: how does attacker determine the 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
```



NOP slide (NOP sled)

How does a NOP sled work? - Stack Overflow

- Some attacks consist of making the program jump to a specific address and continue running from there.
  - *The injected code has to be loaded previously somehow in that exact location*
- Stack randomization and other runtime differences may make the address where the program will jump impossible to predict
  - *So the attacker places a NOP sled in a big range of memory*
- If the program jumps to anywhere into the sled, it will run all the remaining NOPs, doing nothing, and then will run the payload code, just next to the sled.
- ***The reason the attacker uses the NOP sled is to make the target address bigger***: the code can jump anywhere in the sled, instead of exactly at the beginning of the injected code

Details and examples

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

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

overflow when parsing PE headers … kernel vulnerability

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.

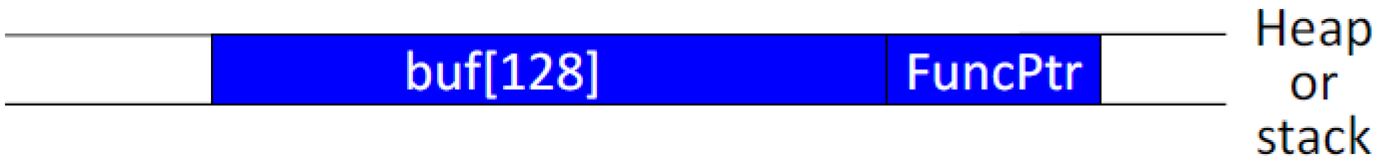
- Windows C run time (CRT):
  - `strcpy_s (*dest, DestSize, *src)`: ensures proper termination

Buffer overflow opportunities

- Exception handlers: (Windows SEH attacks … [more](#))
  - Overwrite the address of an exception handler in stack frame.

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

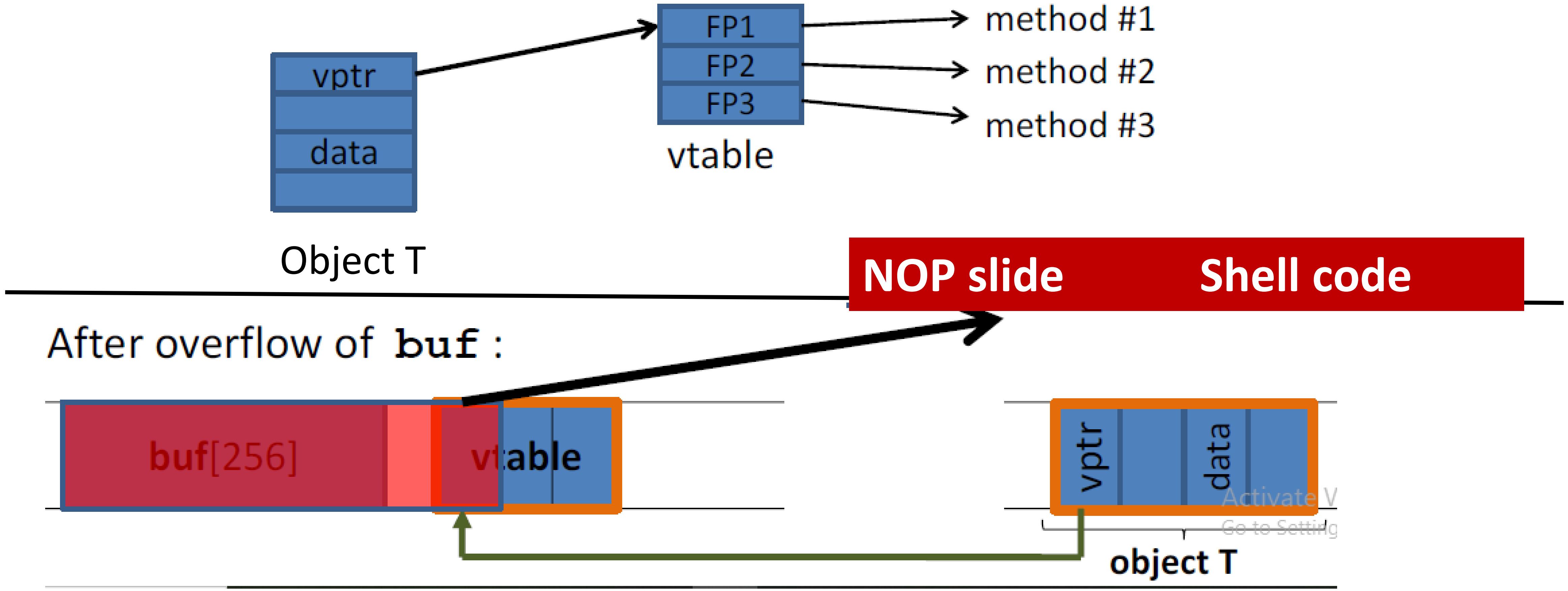
  - Overflow



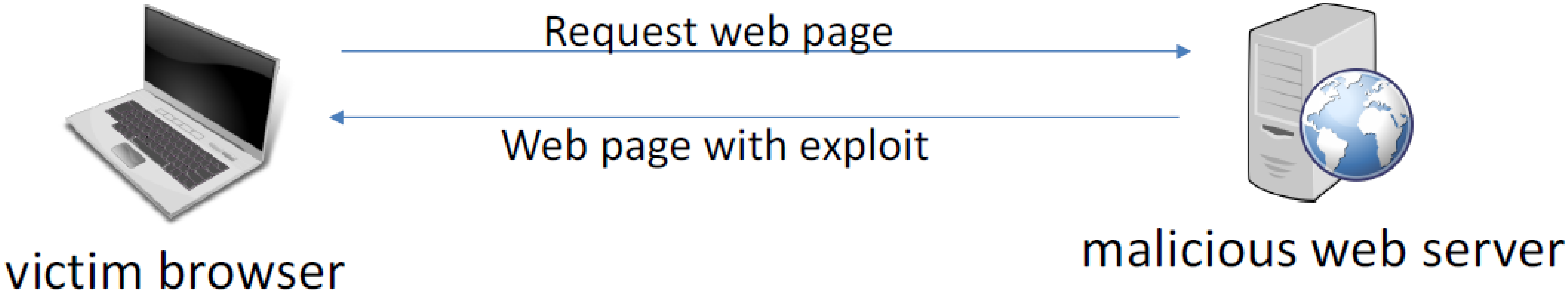
- Longjmp buffers: longjmp(pos) (e.g. Perl 5.003)
  - *longjmp is intended for handling unexpected error conditions where the function cannot return meaningfully. This is similar to exception handling in other programming languages.*
  - Overflowing buf next to pos overrides value of pos.

Heap exploits: corrupting virtual tables

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



An example: exploiting the browser heap



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

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

Finding overflows by fuzzing

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

More Hijacking Opportunities

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

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          ⇒ c = 0
s = 0xff80 + 0x80                    ⇒ s = 0
m = 0xffffff80 + 0x80                ⇒ 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
}
```

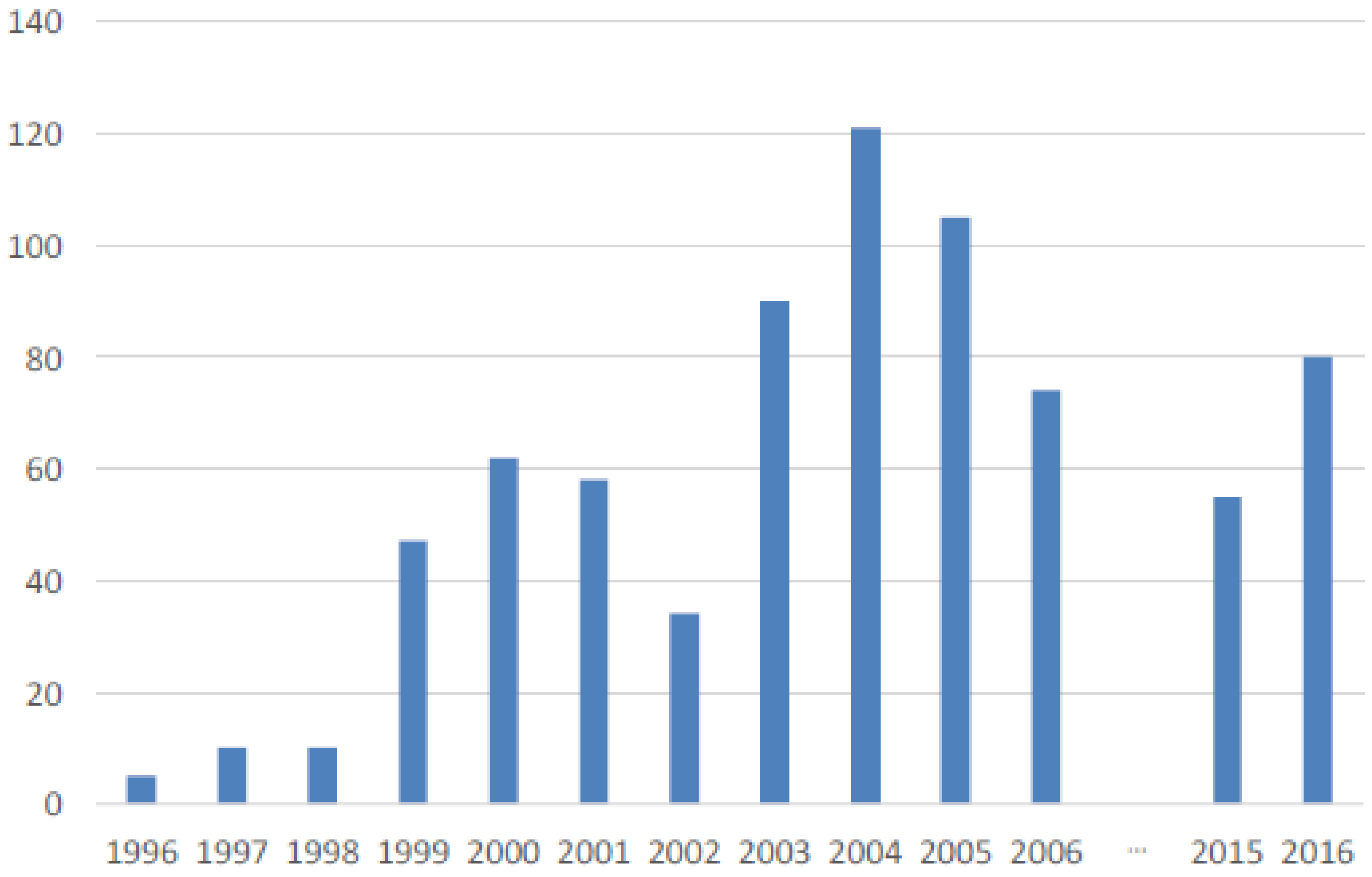
What if **len1 = 0x80, len2 = 0xffffff80** ?

→ len1+len2 = 0

Second memcpy() will overflow heap !!

Integer overflow exploit stats (source: NVD/CVE)

NVD = National Vulnerability Database; CVE = Common Vulnerabilities and Exposures



Preventing Integer Overflow Attacks

- Prefer using *unsigned integer types* whenever possible.
- Review and test your code by *writing out all casts explicitly* to identify where implicit casts might cause integer overflows.
- Turn on *any options available in your compilers* that can help identify certain types of integer overflows.
- Adopt *secure coding practices such as bounds checking*, input validation, and using safer functions.
- Perform a *bounds check on every value that is user-modifiable* before using it in an arithmetic operation.

Format String Attacks

Format string attack

[Format string attack | OWASP Foundation]

To understand the attack, it's necessary to understand the components that constitute it.

- The **Format Function** is an ANSI C conversion function, like `printf`, `fprintf`, which converts a primitive variable of the programming language into a human-readable string representation.
- The **Format String** is the argument of the Format Function and is an ASCII Z string which contains text and format parameters, like:

```
printf (“The magic number is: %d\n”, 1911);
```

- The **Format String Parameter**, like **%x %s** defines the type of conversion of the format function.

- The attack could be executed when the application doesn't *properly validate the submitted input.*
  - In this case, if a Format String parameter, like %x, is inserted into the posted data, the string is parsed by the Format Function, and the conversion specified in the parameters is executed.
  - However, the Format Function is expecting more arguments as input, and if these arguments are not supplied, the function could read or write the stack.

- In this way, it is possible to define a *well-crafted input that could change the behavior of the format function*, permitting the attacker to cause denial of service or to execute arbitrary commands.

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”)`

Preventing Format String Vulnerabilities

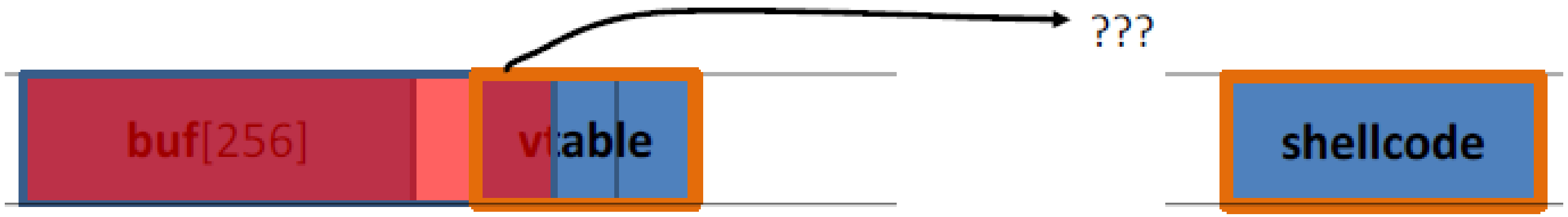
- Always specify a *format string as part of program, not as an input*. Most format string vulnerabilities are solved by specifying **"%s"** as format string and not using the data string as format string
- If possible, make the *format string a constant*. Extract all the variable parts as other arguments to the call. Difficult to do with some internationalization libraries
- If the above two practices are not possible, *use defenses such as Format_Guard* . Rare at design time. Perhaps a way to keep using a legacy application and keep costs down. Increase trust that a third-party application will be safe

Extra Slides

A reliable exploit?

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

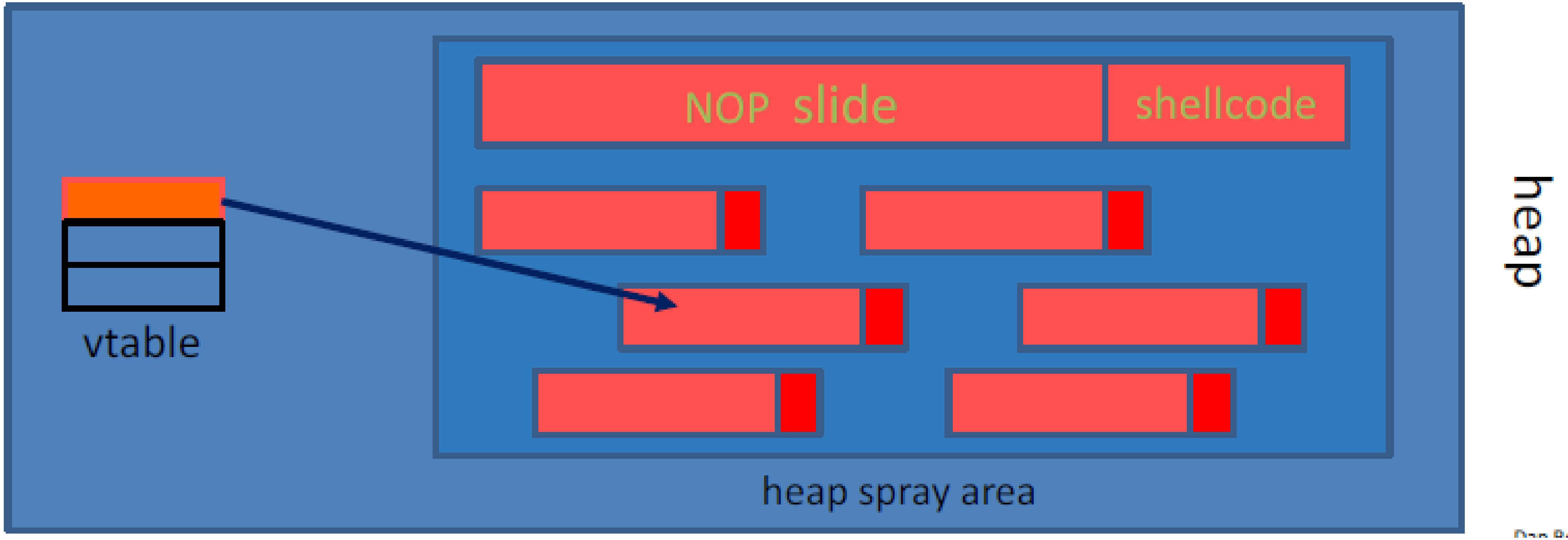
Problem: attacker does not know where browser places shellcode on the heap



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



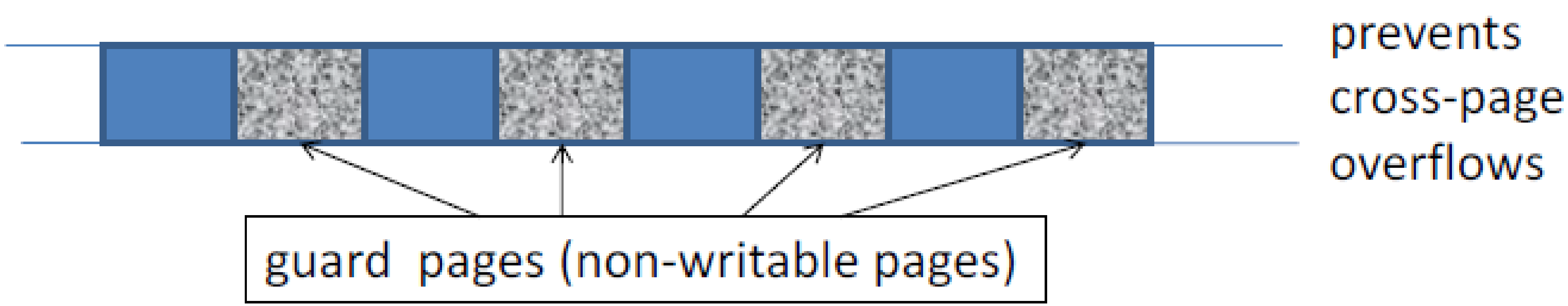
Javascript heap spraying

```
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:



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

Use after free exploits

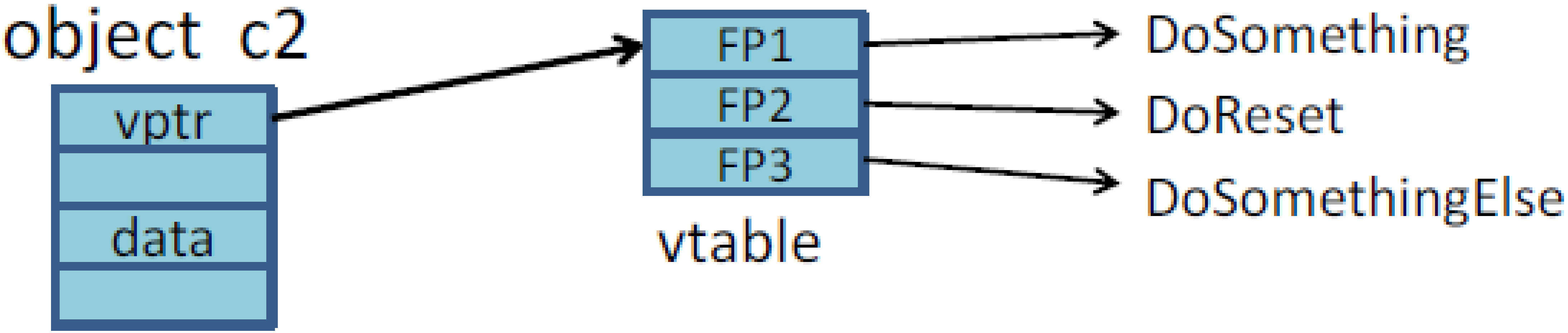
IE11 Example: CVE-2014-0282 (simplified)

```
<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()

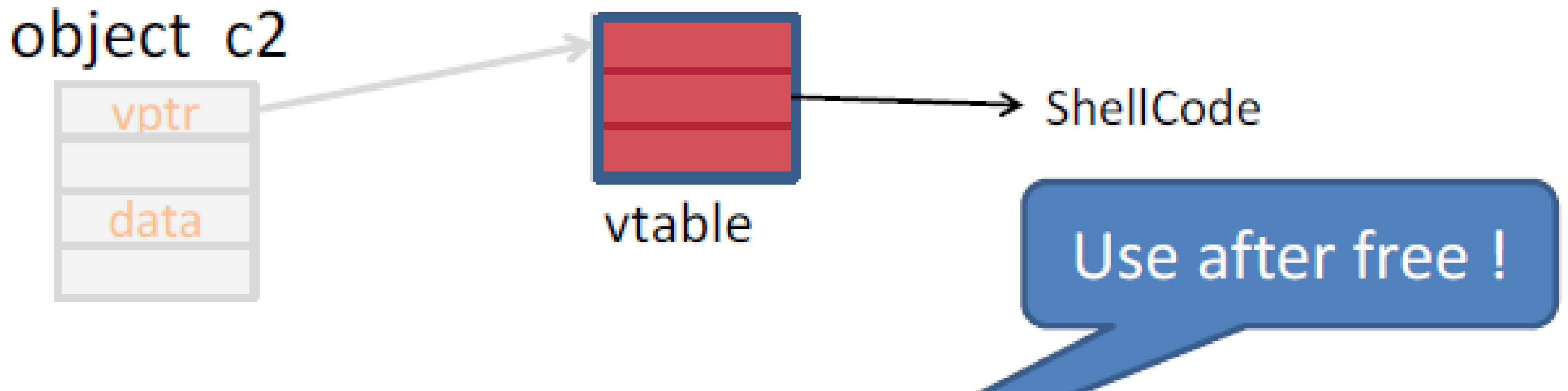
What just happened?

`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



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 !!

Acknowledgments

- Dan Boneh, Stanford University