

COMP3911 Secure Computing

13: Buffer Overruns

Nick Efford

<https://comp3911.info>

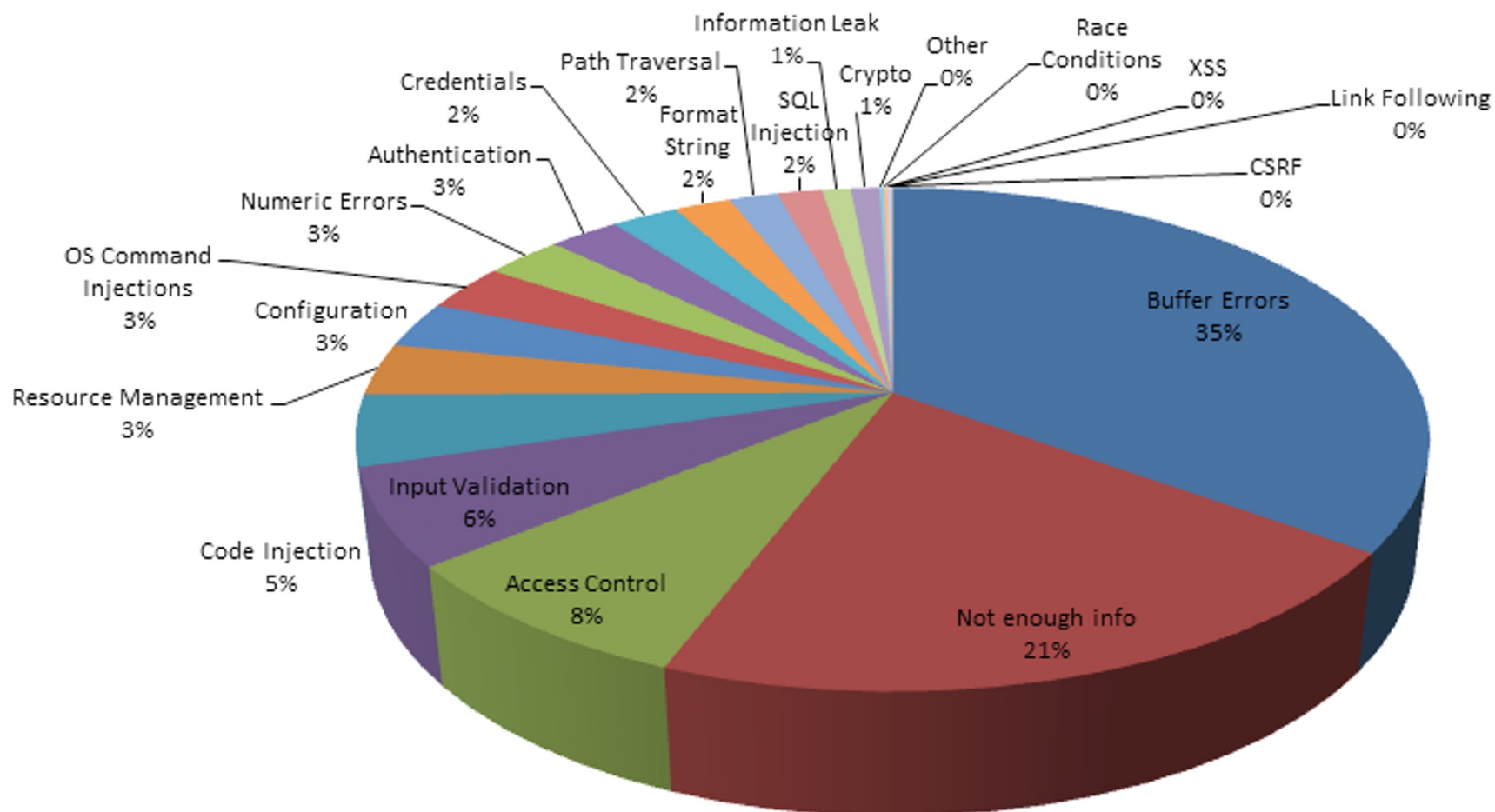
Objectives



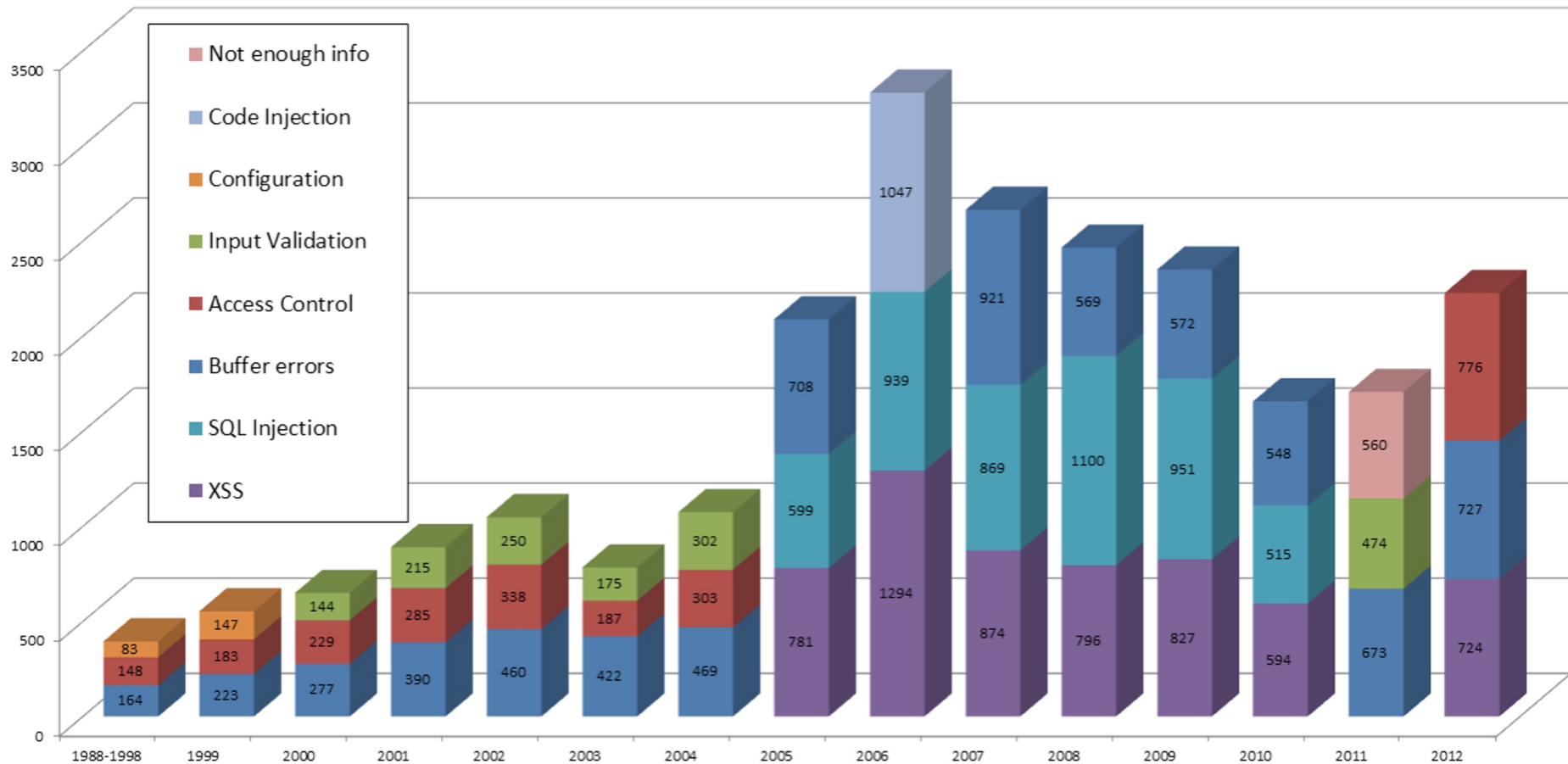
UNIVERSITY OF LEEDS

- To consider the prevalence of low-level buffer errors in comparison with other types of vulnerability
- To explore how **buffer overruns** can occur in C code and what the consequences of this can be
- To discuss defensive measures that make it much harder to exploit buffer overruns

Critical Vulnerabilities (1988–2012)



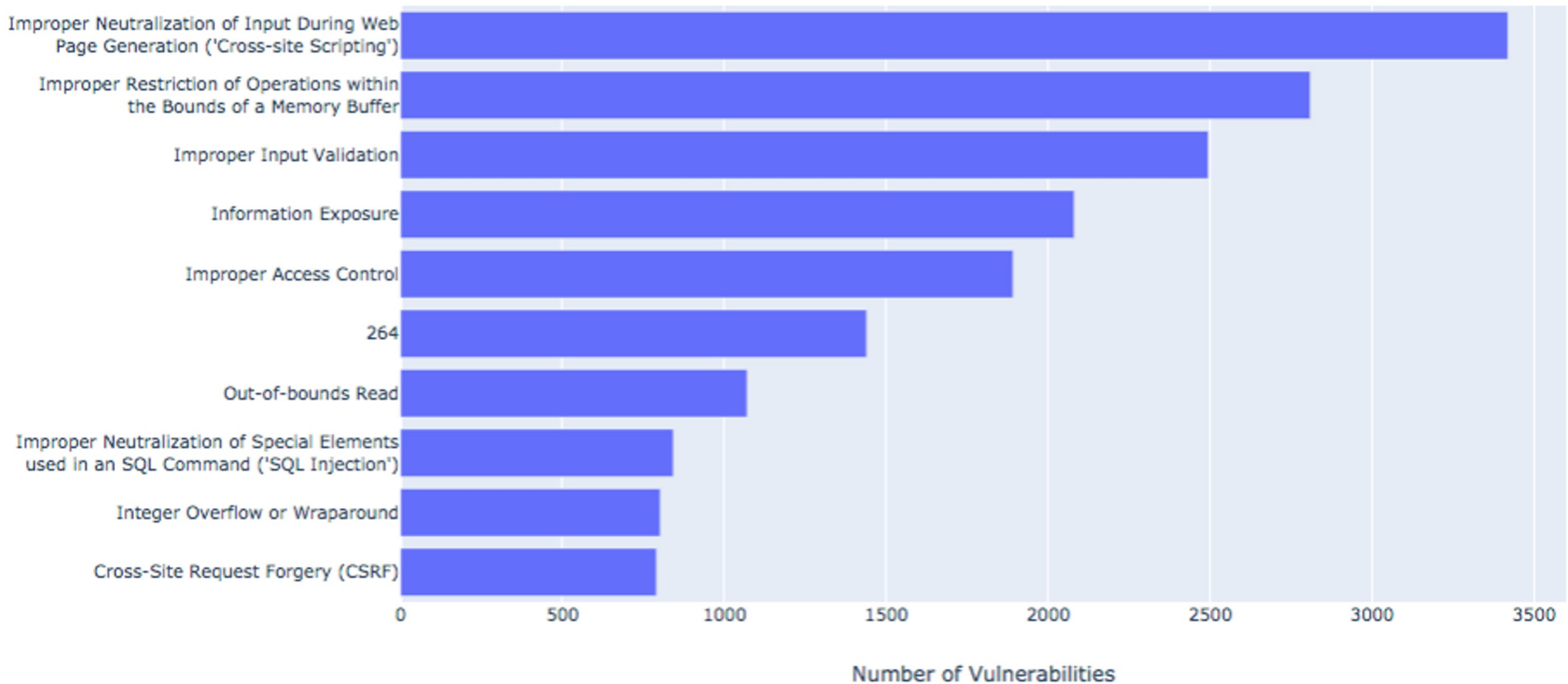
Top 3 Vulnerabilities (to 2012)



So low-level 'buffer errors' have been a significant problem...

More Recently...

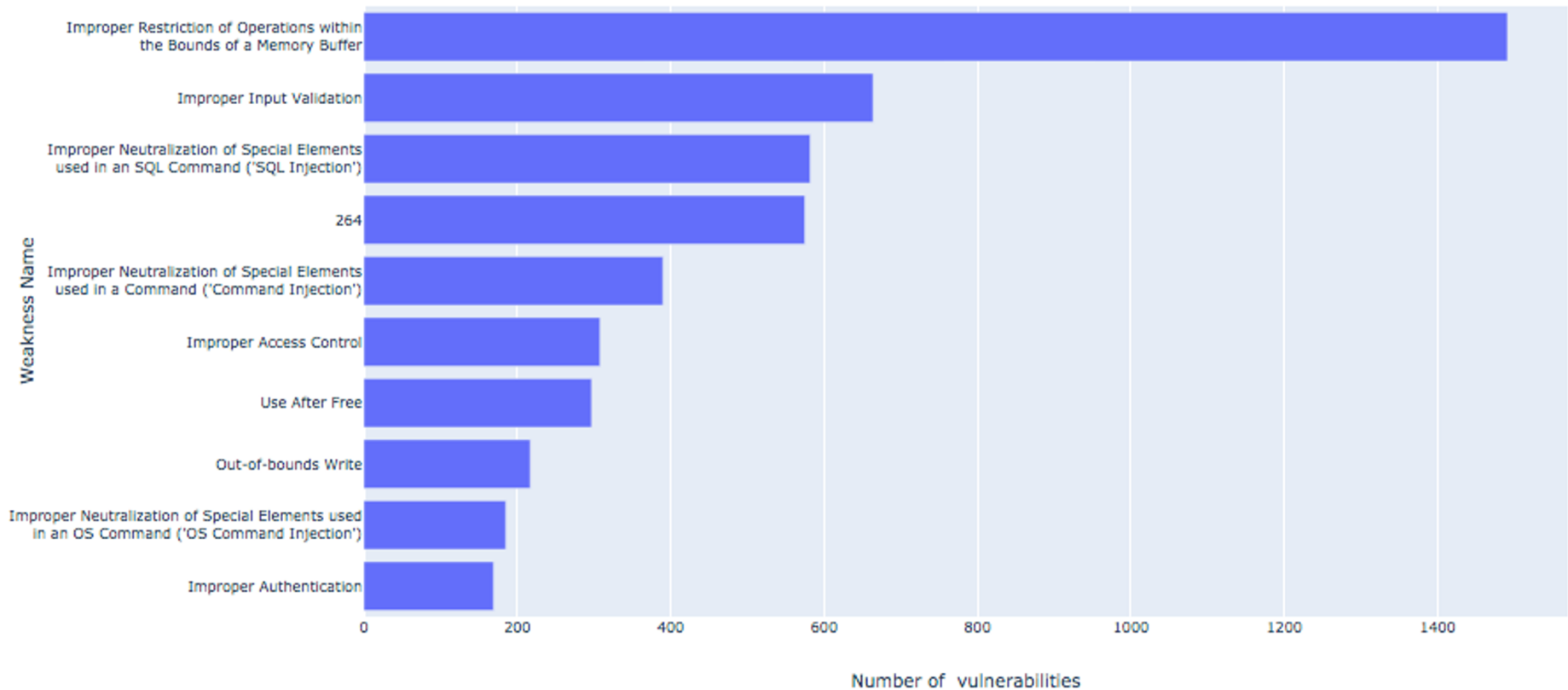
Top 10 weaknesses (CWE)



(from ENISA report [State of Vulnerabilities 2018/19](#))

More Recently...

Top 10 weaknesses with CVSS score ≥ 7



24 Deadly Sins...



SIN 5

BUFFER OVERRUNS

Howard, LeBlanc & Viega, [*24 Deadly Sins of Software Security*](#)

What is a Buffer Overrun?

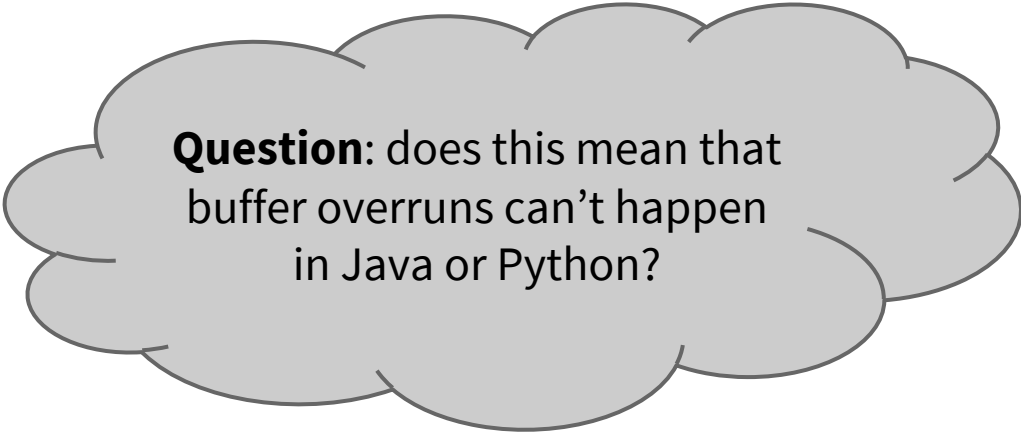
- Program allocates a contiguous chunk of memory of fixed size to store data (a **buffer**)
- Amount of data copied to buffer exceeds its capacity
- Other memory is overwritten
- ... with a variety of harmful consequences (potentially)



See [Exercise 16](#)

Where Do They Happen?

- Typically occur in programs written in C or C++
 - Because bounds-checked memory access is **not done by default** in either language
- Can be serious because critical programs are often written in these languages, for performance reasons



Question: does this mean that
buffer overruns can't happen
in Java or Python?

Simple Example

```
char buf[32];
```

```
strcpy(buf, input);
```

strcpy copies
input into buf
without checking
its size!

... so if input is longer
than 31 chars **and can be
affected by attacker**, we
may have a problem

Vulnerability Note VU#677427

D-Link routers HNAP service contains stack-based buffer overflow

Original Release date: 07 Nov 2016 | Last revised: 10 Nov 2016



Overview

D-Link DIR routers contain a stack-based buffer overflow in the HNAP Login action.

Description

CWE-121: Stack-based Buffer Overflow - CVE-2016-6563

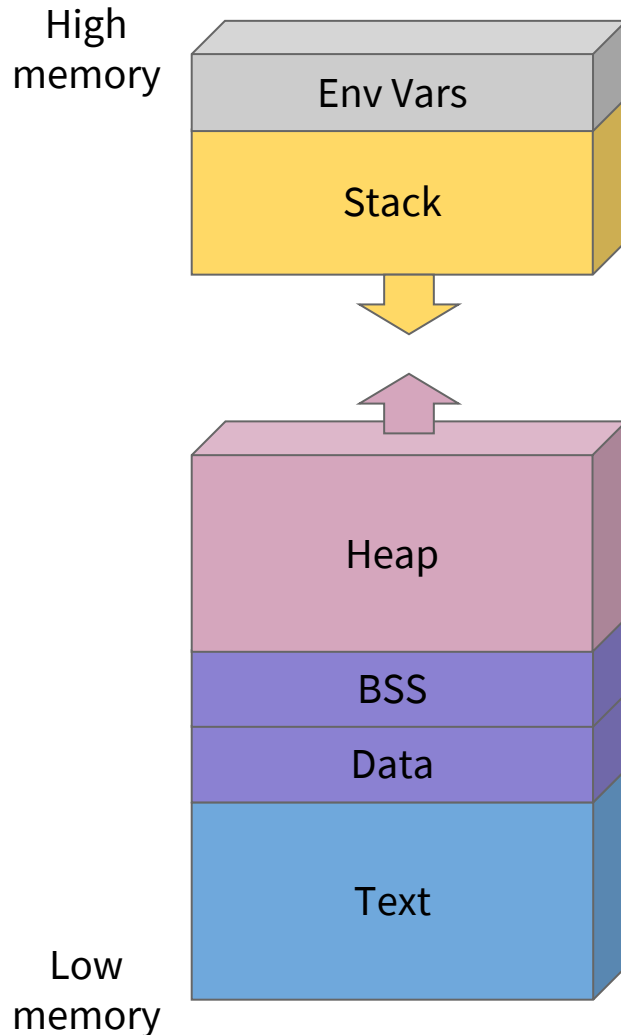
Processing malformed SOAP messages when performing the HNAP Login action causes a buffer overflow in the stack. The vulnerable XML fields within the SOAP body are: Action, Username, LoginPassword, and Captcha.

“A remote, unauthenticated attacker may be able to execute arbitrary code with root privileges.”

Program Memory Layout

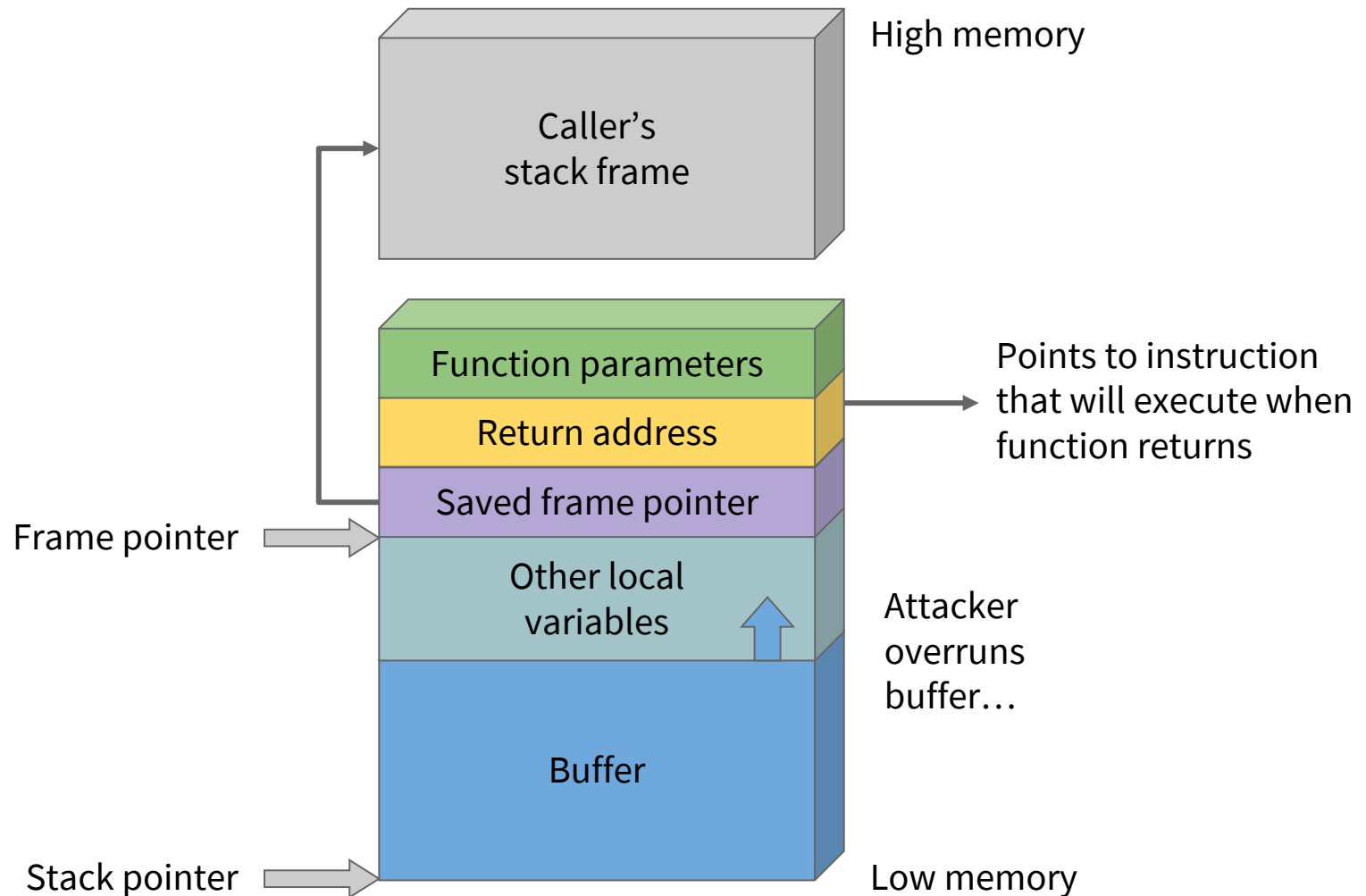


UNIVERSITY OF LEEDS

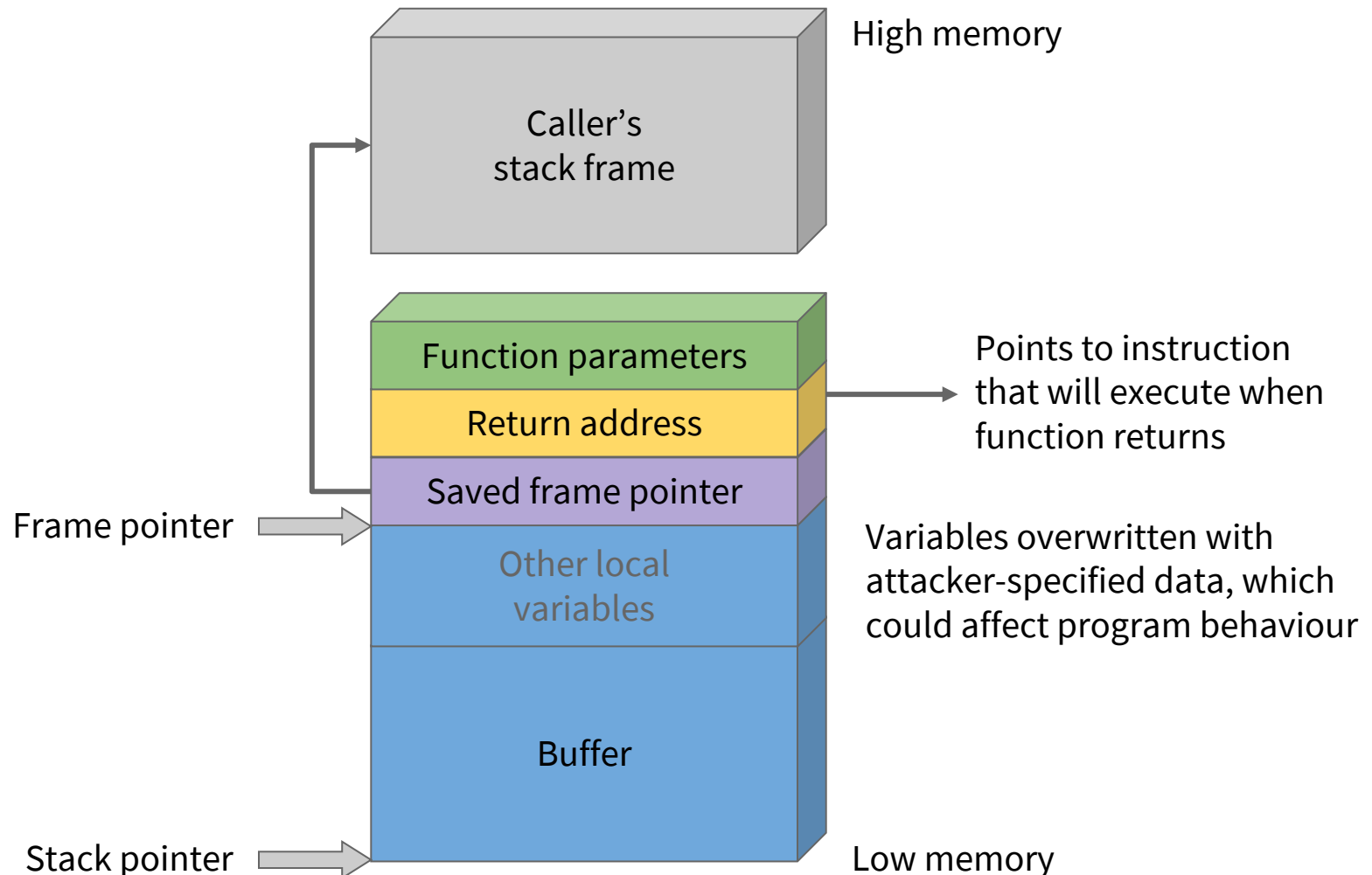


- Text segment holds program instructions (read-only)
- Data & BSS segments provide storage for static / global data
- Stack & heap change size as program executes
- Stack holds information about context of function calls in a **stack frame**
 - Function parameters
 - Local variables
 - Saved register information
 - Return address for call

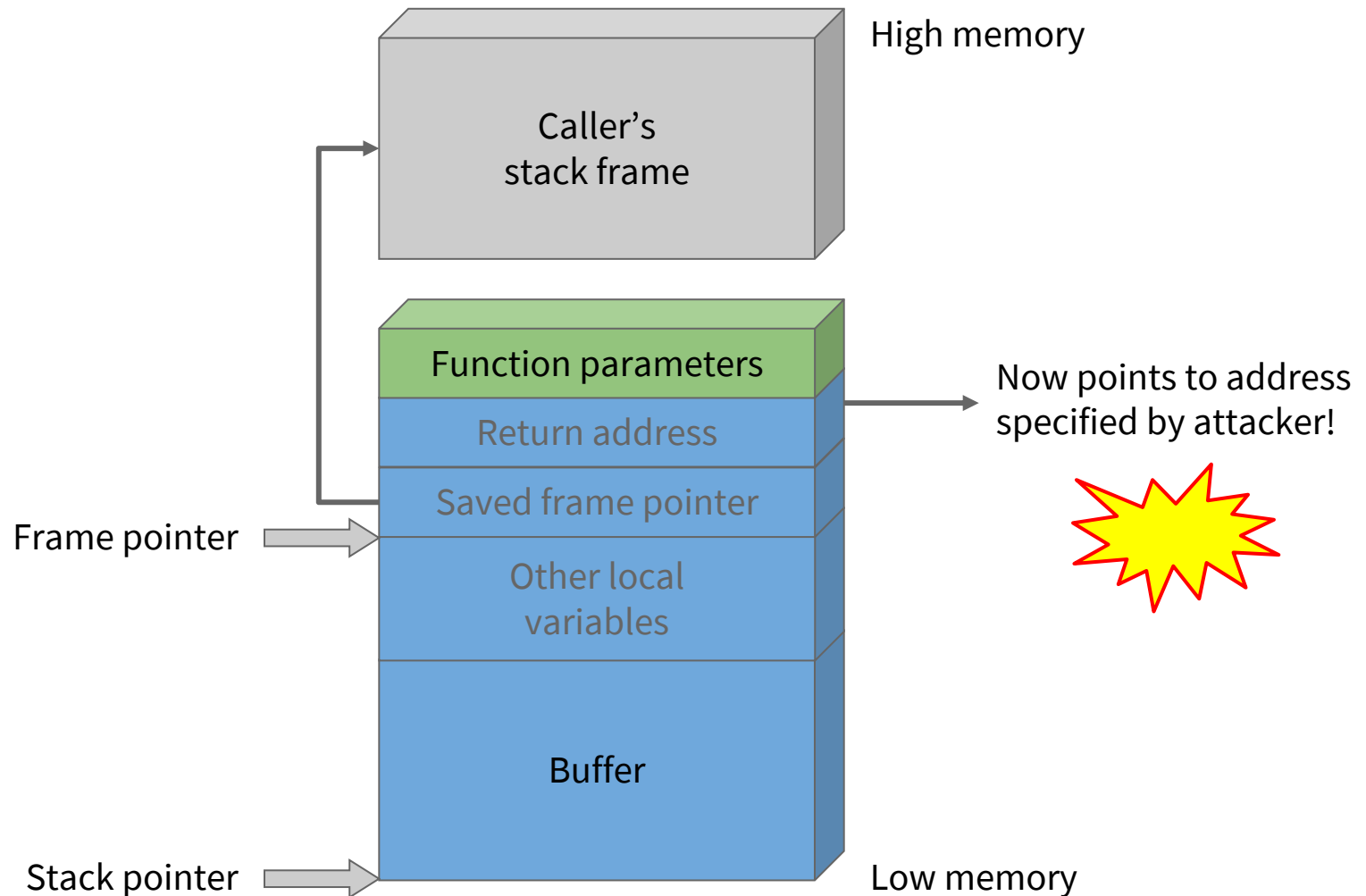
Stack Smashing



Stack Smashing



Stack Smashing



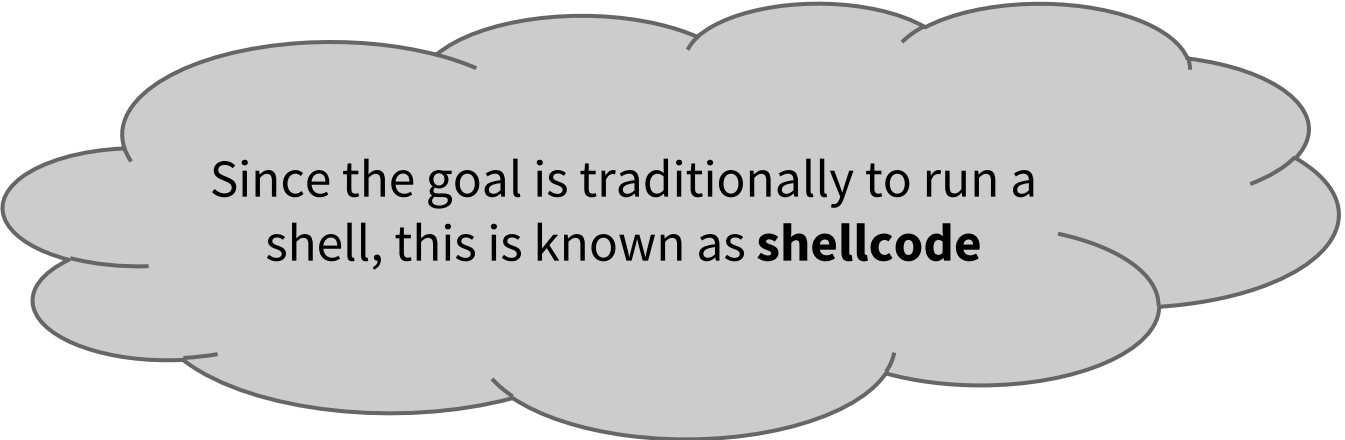
Worst-Case Scenario

- Program is a server, running with high privileges
- Attacker can overrun a buffer with input they supply
- ... and can make return address point back into this input
- ... which contains machine code!

Crafting an Exploit

Typical approach:

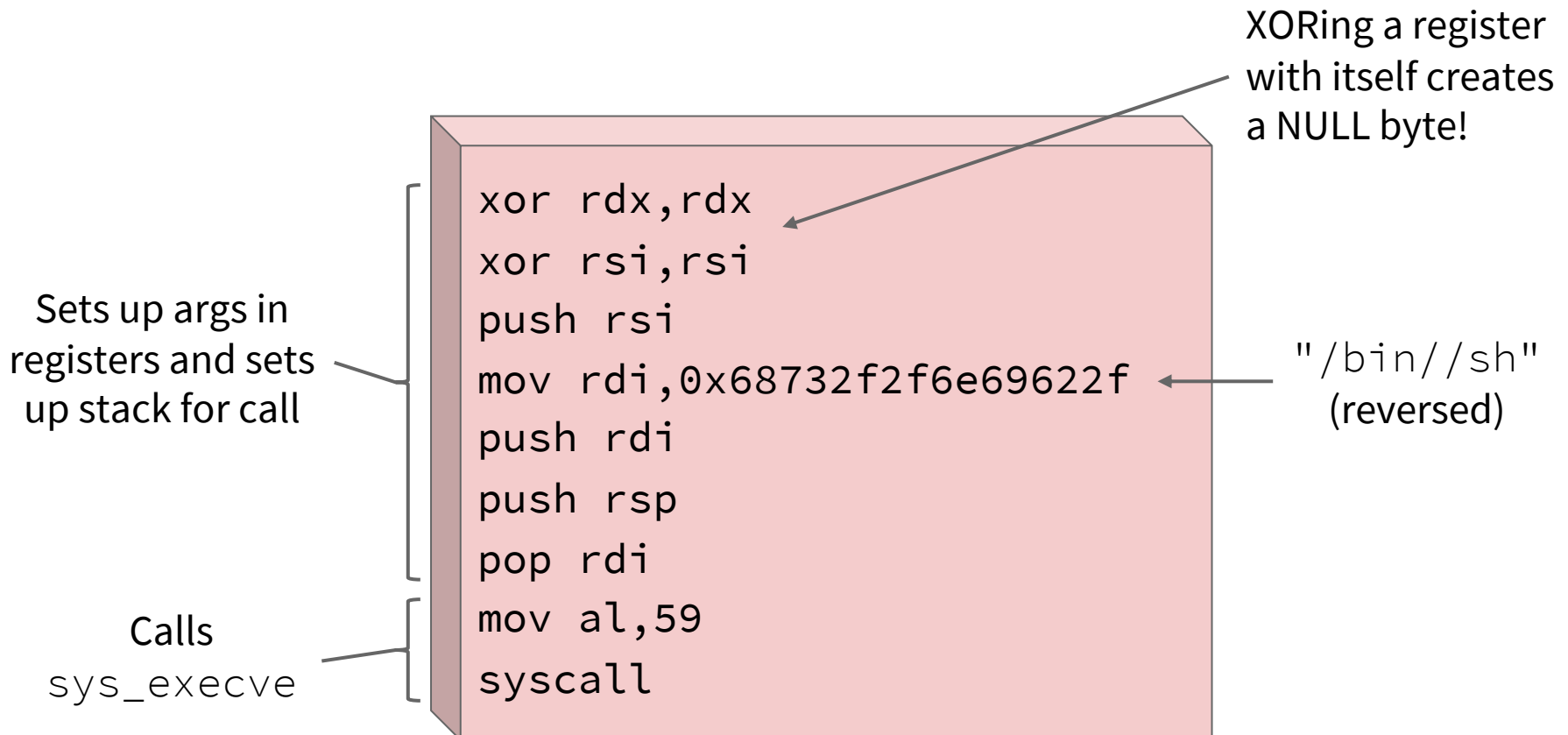
1. Write exploit in C (e.g., to make a system call that is useful to the attacker) and compile to object code
2. Disassemble and hand-edit the assembly language – e.g., to remove explicit NULLs that would stop `strcpy()` from copying the code



Since the goal is traditionally to run a shell, this is known as **shellcode**

Shellcode Example

Using `execve()` to run `/bin/sh`

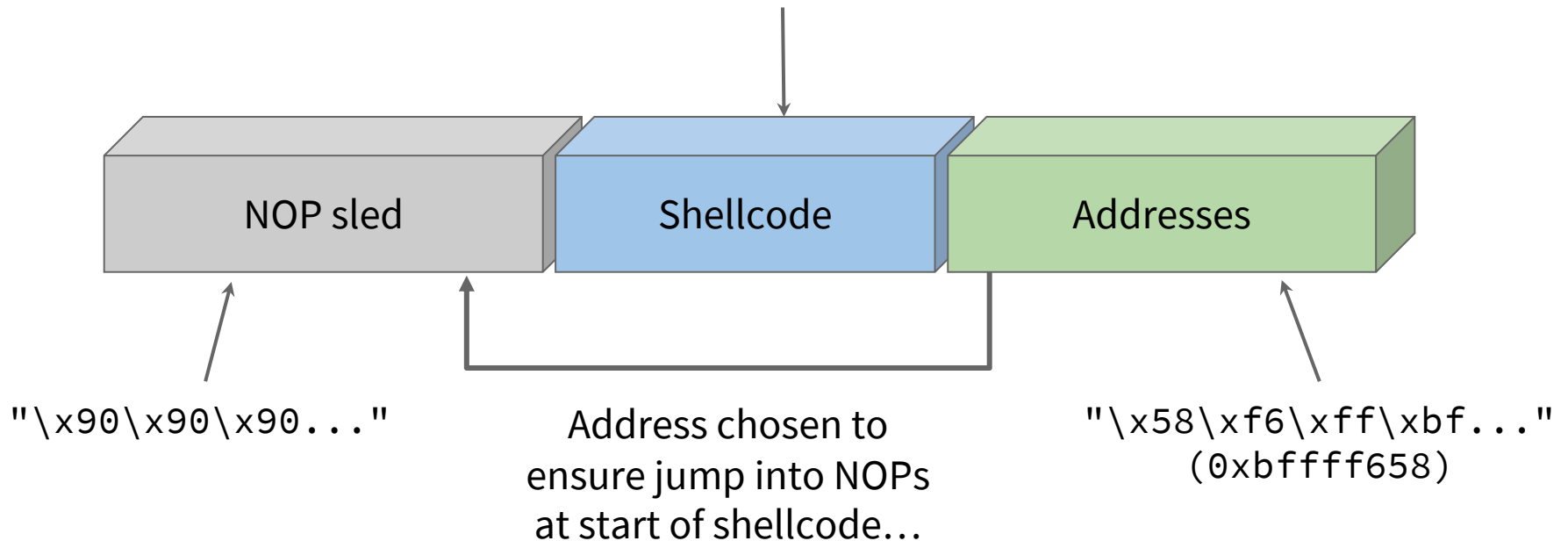


Crafting an Exploit

- In a blind attack, hacker doesn't know
 - How big the buffer is
 - How much space is occupied by other local variables between the buffer and the return address
- So hacker doesn't know exactly where to put a return address that will force shellcode execution
- ... or exactly what that return address should be!

Crafting an Exploit

```
"\x50\x48\x31\xd2\x48\x31\xf6\x48  
\xbb\x2f\x62\x69\x6e\x2f\x2f\x73  
\x68\x53\x54\x5f\xb0\x3b\x0f\x05"
```



Questions

- Are very small stack frames susceptible to attack?
 - Yes, if attack can inject shellcode elsewhere (e.g., in environment variables area)
- How big must an overrun be in order to execute shellcode?
 - As little as one byte! (**off-by-one error**)

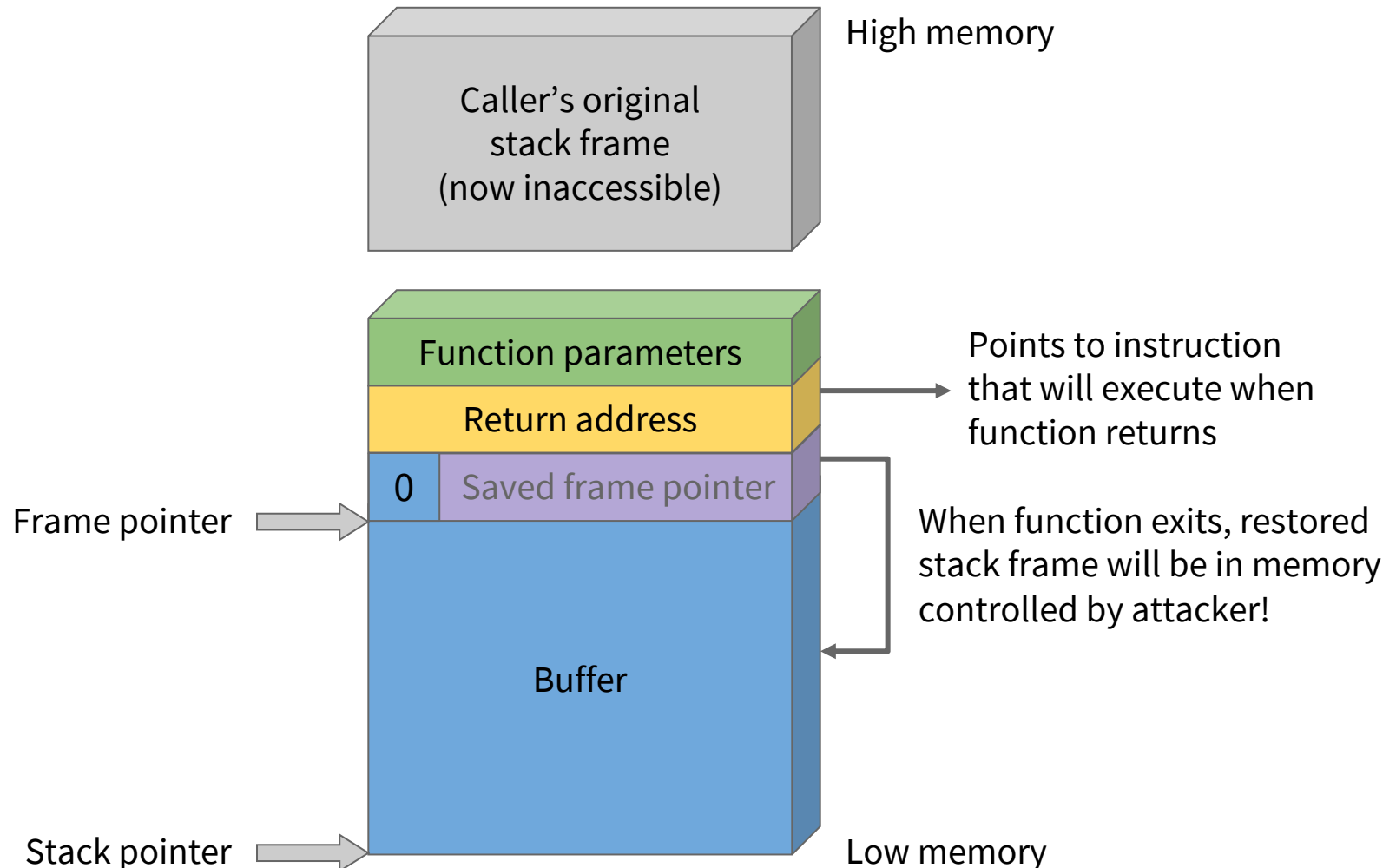
```
char buf[256];
```

```
strncpy(buf, input, sizeof(buf));  
buf[sizeof(buf)] = '\0';
```



should be a -1 here!

Off-By-One Error



Heap Overruns

- Can overwrite application variables, as with stack
- Can also overwrite metadata used to manage storage
 - Heap is a linked list of allocated chunks
 - Careful attacker may be able to overwrite the pointers linking these chunks
 - ... which may allow them to write to a memory location of their choice
 - ... and even trigger execution of shellcode

Defences



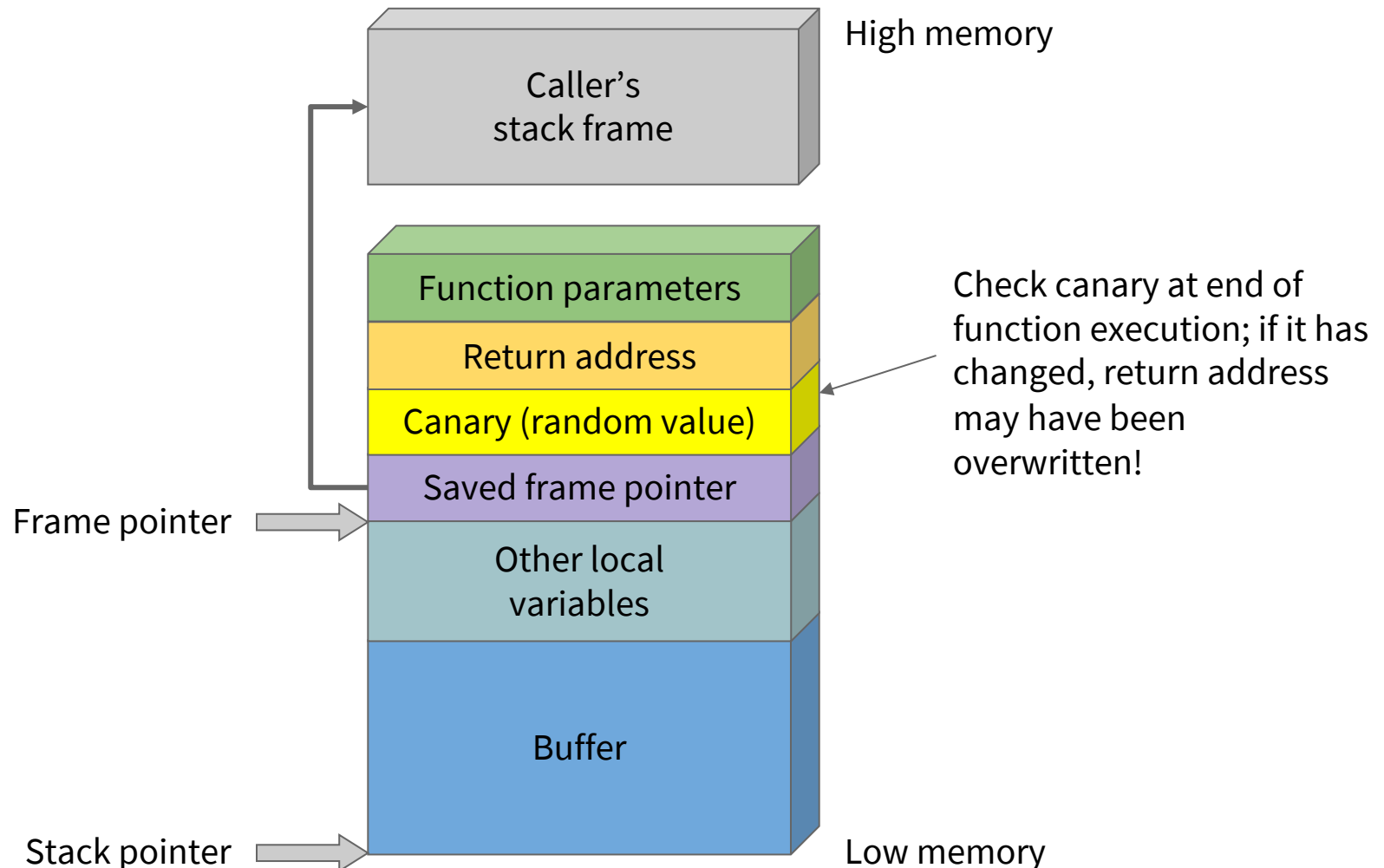
UNIVERSITY OF LEEDS

- Correct programming! (bounds checking)
- Compile protection into applications
- Transparent protection
 - Support from OS kernel / hardware
 - Intercept dangerous library calls

Protection Via Recompilation

- Compile with safer version of C library
- Use compiler support for bounds checking
 - GCC doesn't do this by default, [but can be patched](#)
 - /RTCs option in Visual Studio
- Use compiler stack protection support
 - -fstack-protector options in GCC
 - /GS option in Visual Studio (on by default)

Stack Protection Using a Canary



Transparent Protection

- Non-executable stack & heap
 - NX bit on 64-bit CPUs; if bit 63 of a page table entry is set to 1, code cannot be executed from that page
 - ... or can emulate in software (e.g., Windows DEP)
 - Doesn't prevent 'return to libc' attacks
- Address Space Layout Randomisation (ASLR)
 - Positions of stack, heap, libraries change randomly on every execution of program
 - ... making it harder to run shellcode reproducibly

Summary

We have

- Reviewed the vulnerability landscape and highlighted the significance of low-level buffer errors
- Explored how the classic ‘stack smashing’ attack works
- Discussed variations such as ‘off-by-one’ and heap attacks
- Investigation various protections against buffer overruns, involving recompilation or hardware / OS
- Noted that correct programming is the best defence!

Follow-Up / Further Reading

- [Exercise 16](#)
- Chapter 5 of [24 Deadly Sins of Software Security](#)
- Microsoft's [‘Checked C’ initiative](#)
- Recent vulnerability reports – e.g., the BugTraq archives at www.securityfocus.com
 - How many buffer overruns?
 - How many in the heap vs the stack?