Week 10 - Software Security



THE UNIVERSITY OF SYDNEY

Dr. Suranga Seneviratne

School of Computer Science, The University of Sydney

Agenda

- What are the software security problems?
- Background
- Buffer overflow attacks
- Defences against software vulnerabilities

Recommended Reading

- Hacking: The Art of Exploitation – Second Edition by Jon Erickson:
  - **Chapter 2** – Programming
  - **Chapter 3** – Exploitation

- https://www.oreilly.com/library/view/hacking-the-art/9781593271442/

Software Security

- So far, we have mostly discussed security at the design level
  - Security goals
  - Cryptography
  - Security mechanisms (authentication, access control, etc.)
  - Network security (protocols and firewalls) software is free of errors at the implementation level

Terminology



Flaw is a system property at the design level that may lead to breach of security policy.



Vulnerability is a system property at the implementation level that may lead to breach of security policy.



Exploit is a method (e.g. in software) to breach a security policy due to a flaw or vulnerability.



Bug is a property in a program's code that causes it to produce incorrect or unexpected results



Note: Every vulnerability is a bug; but not every bug is a vulnerability.

1. What are the software security problems?



Problems in Software Security

- The security of software is influenced by a number of factors.
  - Language safety
    - Memory safety
    - Type safety
- API safety
- Dependencies
- Available tools to identify bugs, in particular vulnerabilities
- …

Programming Language Safety



Language Safety: What protection from undefined or undesired behaviour does the language and its ecosystem of development tools and compilers provides you?

- The earlier and error is raised, the safer the software will be
- Three main types:
  - Memory safety
  - Type safety
  - Thread safety

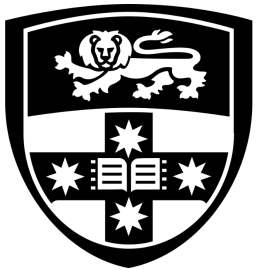
Feature	Unsafe Languages	Safe Languages
Memory safety	C, C++	Rust > Java, Python
Type safety	C, C++	Haskell, Java
Thread safety	C, C++	Rust, Java

Why bother about C/C++



Why C/C++ matter?

- Why don’t we just all use safe languages? Many reasons.
  - Legacy code
  - Speed - Compiling C/C++ results in code that’s highly efficient on the given machine
  - Control over number of CPU cycles used → Side-channel attacks!
  - Resources - Higher-level languages require a costly execution environment
- Even if you don’t intend to program C/C++, it’s imperative that you understand the trade-offs made between control the programming language gives you and the protection it awards you.



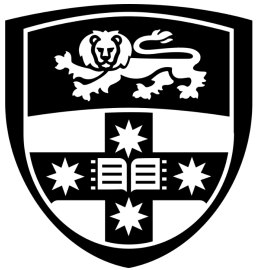
THE UNIVERSITY OF
SYDNEY

Memory Safety in Programming Languages

- **What is memory safety?**
- A program is memory safe if **memory access errors** never occur. Specifically:
  - **Buffer overflows** - Writing data beyond allocated memory boundaries, corrupting adjacent memory.
  - **Null pointer dereference** - Attempting to access memory through a pointer that points to nothing (NULL/0).
  - **Dangling pointer/use after free errors** - Accessing memory through a pointer after that memory has been deallocated.
  - **Double free errors** - Calling free/delete on the same memory address twice, corrupting the memory allocator.
- A language is memory safe if all programs executed in the language are free of memory access errors.

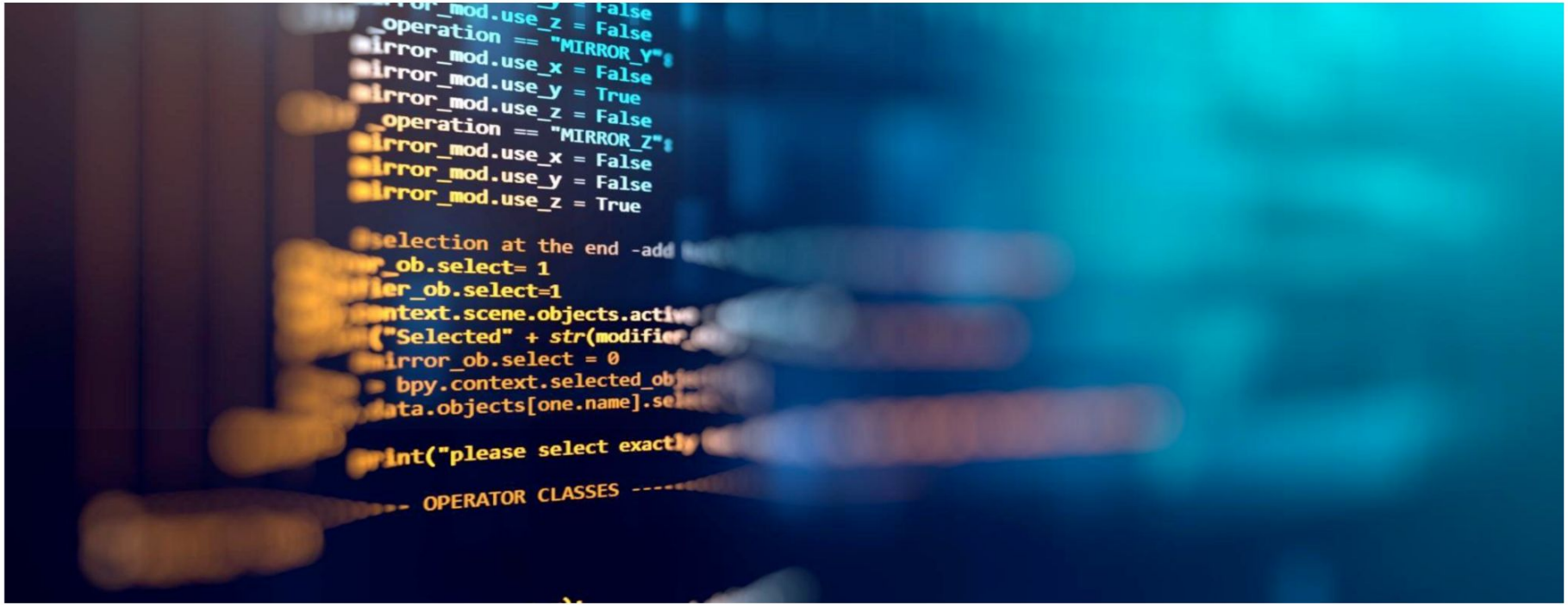
Memory Unsafe Languages

- **What makes a language unsafe?**
- Taking the example of C, C++
  - Support arbitrary pointer operations
  - Do not have bounds checking on arrays and memory allocation
  - Program can overwrite bounds without errors being raised
- Here, we focus on memory access errors in C and common pitfalls to look out for
  - Widely used in critical applications (embedded, IoT)
  - Pitfalls can allow for full system compromise
  - Defences may not exist on older/weaker systems



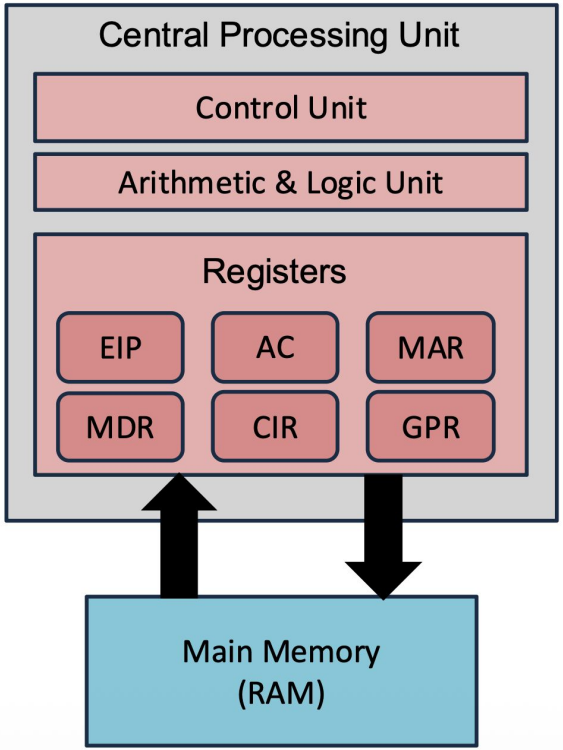
THE UNIVERSITY OF
SYDNEY

2. Background



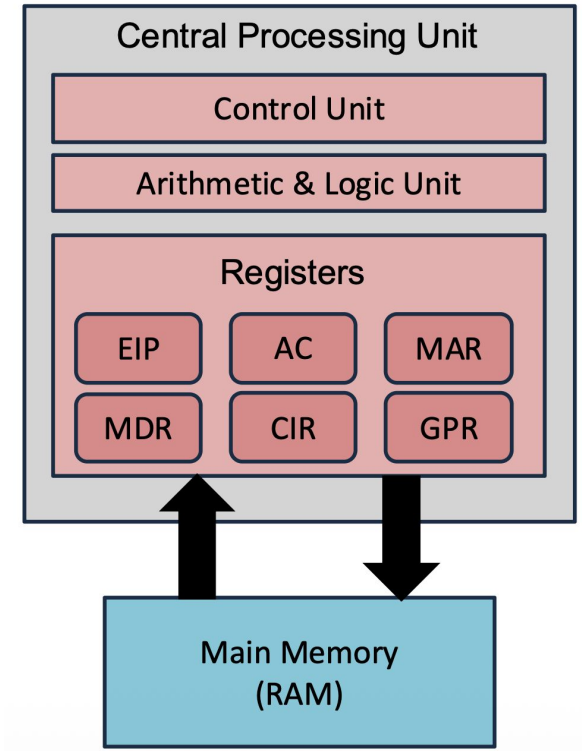
Central Processing Unit (CPU)

- **CPU registers**
  - Small and very fast
  - Located within CPU
  - Temporarily hold data for instruction execution
- **Main memory**
  - Relatively larger and slower
  - Outside CPU but directly addressable
  - Holds data and instructions of a running program

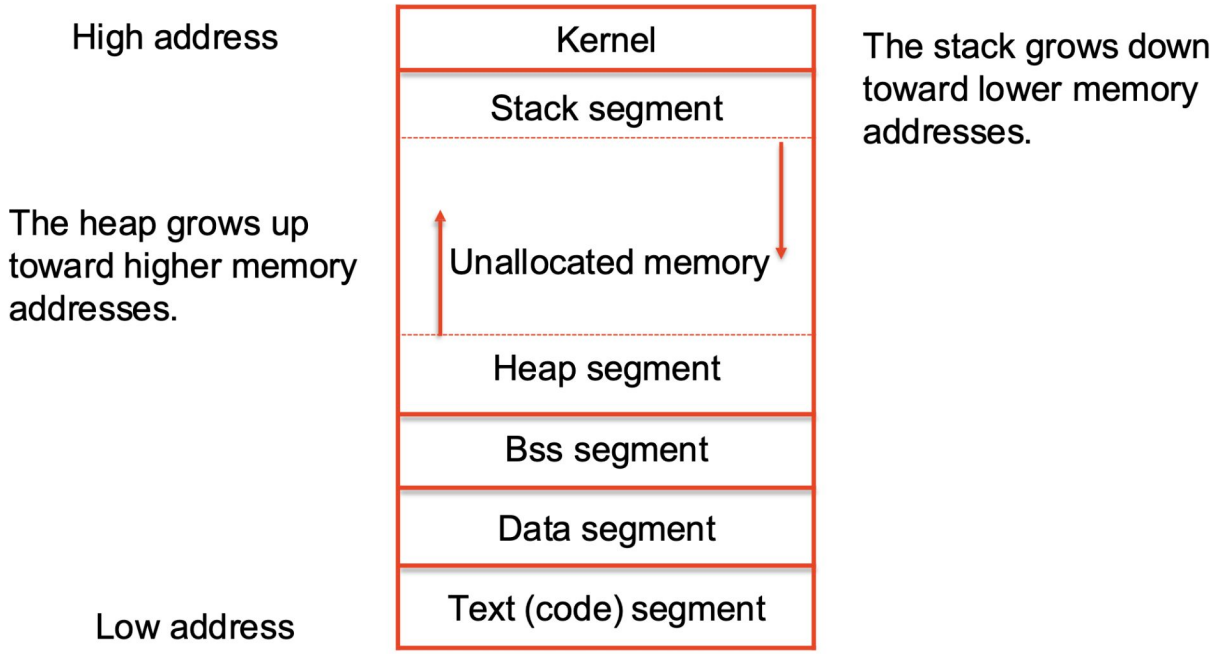


Instruction Cycle

- **Fetch**
  - CPU retrieves instruction from memory using the address in PC.
  - PC is incremented to point to the next instruction in memory.
- **Decode**
  - The CPU decodes binary instruction to understand what operation it specifies (e.g., add, move, jump).
- **Execute:**
  - The decoded instruction is executed by the ALU or other execution units.

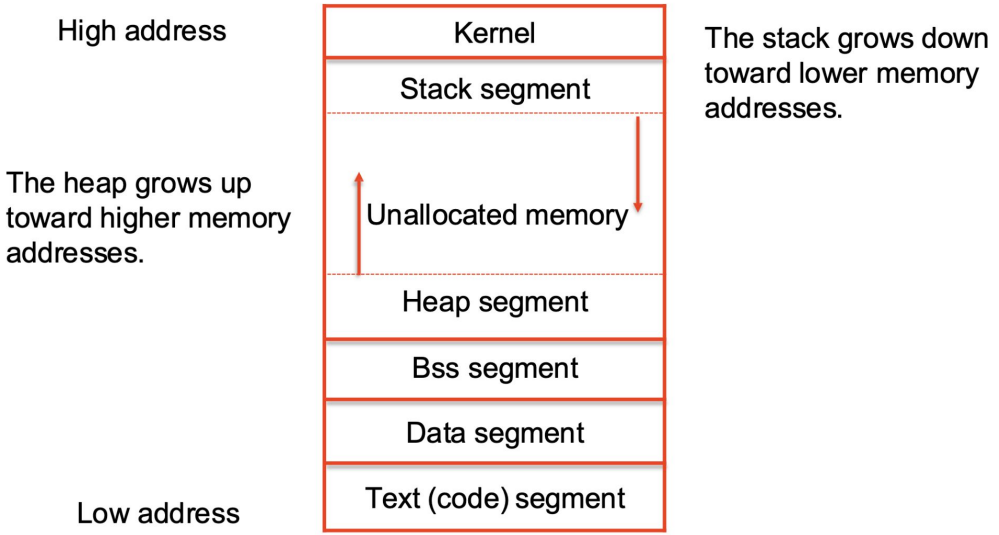


Memory Segments



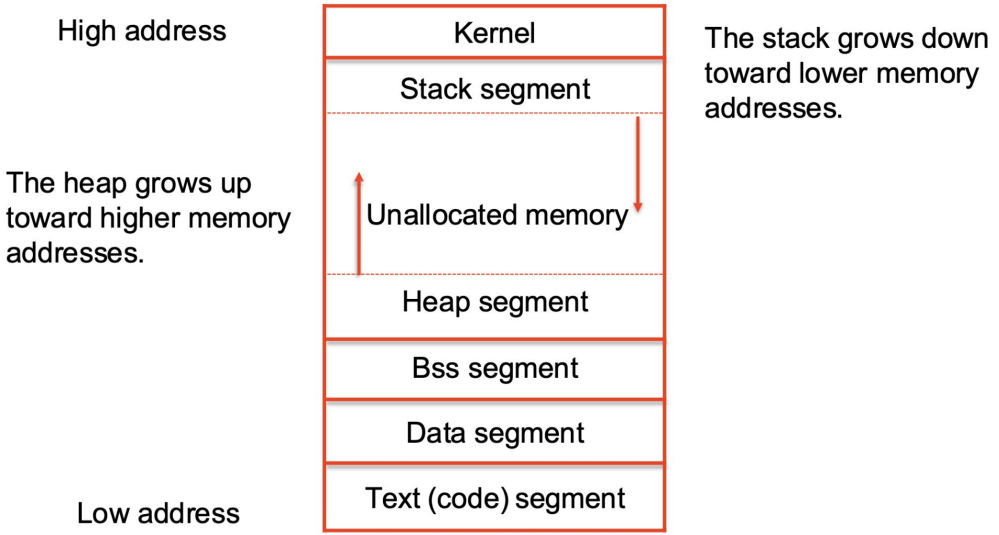
Memory Segments

- Mixing of data and program code - CPU fetches code from memory, where it is stored alongside data
- If attackers can get CPU to read code from wrong location, they might be able to control the program



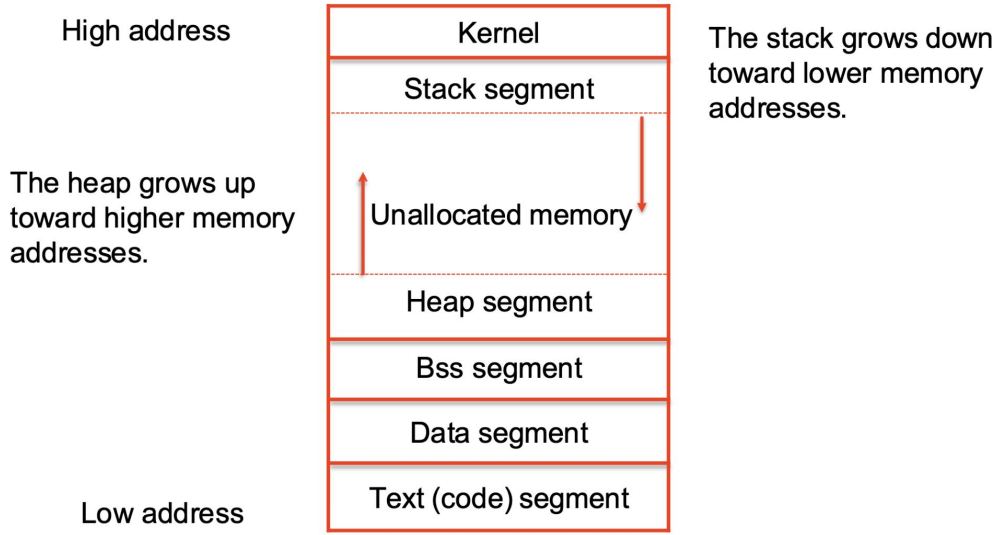
Memory Segments in C

- There are three main regions in C memory model.
  - **Global/static memory** - stores variables that are allocated for the lifetime of program. e.g.
    - `const int x = 5;`
- **Heap** - used for dynamic memory allocation
  - Allocation and freeing controlled by program
  - `s = (char *) malloc(10);`



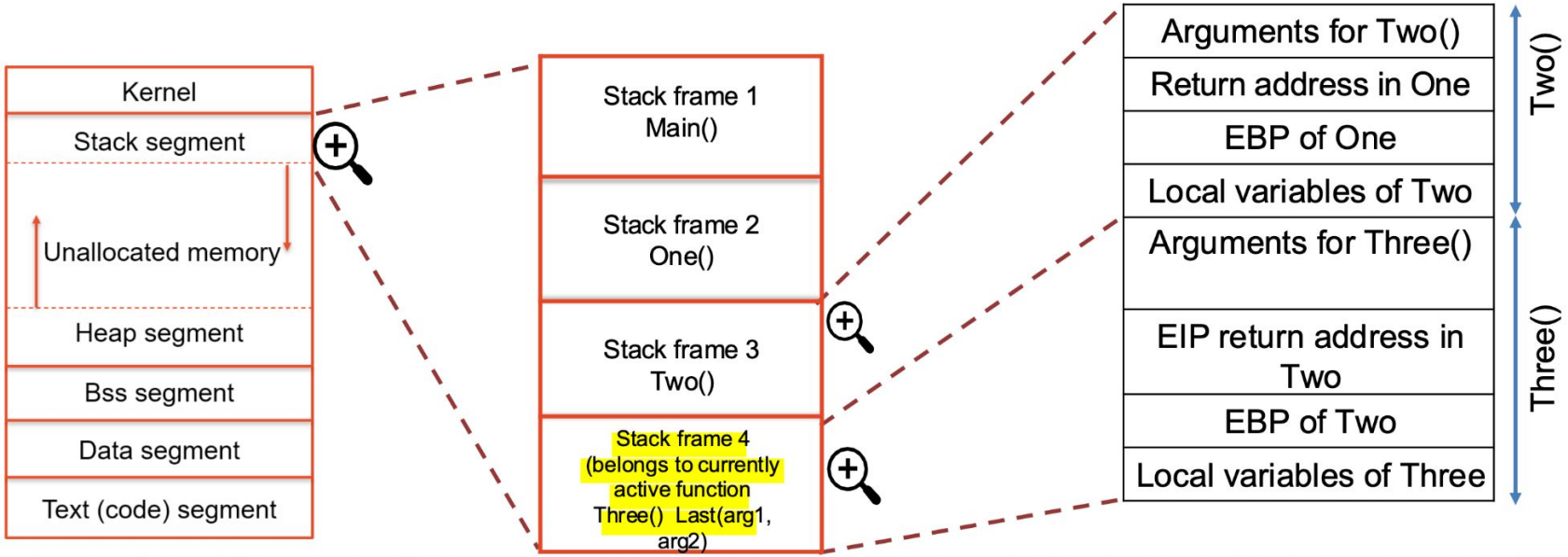
Memory Segments in C

- **Stack** - used when function calls are made
  - Typically grows from high to low memory addresses (convention)
  - Managed by compiler
  - **Stack pointer** points to address of last item put onto stack
  - Consists of **stack frames**
  - One stack frame holds data that the function needs to execute and return
    - void func(){ int x;}
    - x and any other functions called within func() will grow and shrink the stack



In this lecture, we will focus on issues with the stack.

Stack Frame


Kernel
Stack segment
Unallocated memory
Heap segment
Bss segment
Data segment
Text (code) segment
Stack frame 1
Main()
Stack frame 2
One()
Stack frame 3
Two()
Stack frame 4
(belongs to currently active function
Three() Last(arg1, arg2)
Arguments for Two()
Return address in One
EBP of One
Local variables of Two
Arguments for Three()
EIP return address in Two
EBP of Two
Local variables of Three
Two()
Three()

Function Calls

- A function call jumps to a different memory address to execute a separate instruction block.

- **Note:** Before starting the new function, need to save current function context to be able to return (give control back to calling function) E.g.:
  - Stack pointer
  - EIP (Return address)

Function Calls

- A function call jumps to a different memory address to execute a separate instruction block.

- **Fetch:**
  - Fetch the function_call instruction from the address pointed to by EIP.
  - Increment PC (EIP) to point to the next instruction (return address).

- **Decode:** The CPU decodes binary instruction.

- **Execute:**
  - Push return address (current EIP) and other information about current function onto the stack.
  - Update EIP to point to the function's starting address

Stack and Registers

- CPUs (generally) have dedicated registers where to store value to manage the stack.
  - Base pointer - points to base of stack (high addresses)
  - Stack pointer - points to top of stack (low addresses)
  - Instruction pointer - points to address of next instruction
- For an Intel architecture:

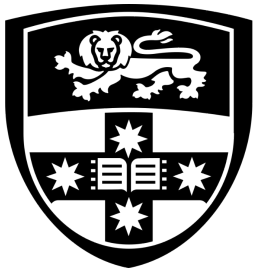
	x86 (32 bit)	x86_64 (64 bit)
Base pointer	EBP	RBP
Stack pointer	ESP	RSP
Instruction pointer	EIP	RIP

Stack-Based Buffer Overflow

- In languages with full memory control/poor memory safety, the following can happen.
  - A function creates a buffer of, say 10B, to hold user input.
  - Buffers are local variables
  - Let's say the function does not check the length of the user input (say, 20B)
  - Instead, it does strcpy(buf, input): copy input into buf, blindly
  - No bounds checking occurs
  - Buffer overflows and corrupts 10 bytes of above memory addresses
  - Then user input can also overwrite EIP - and set it to a value controlled by the user

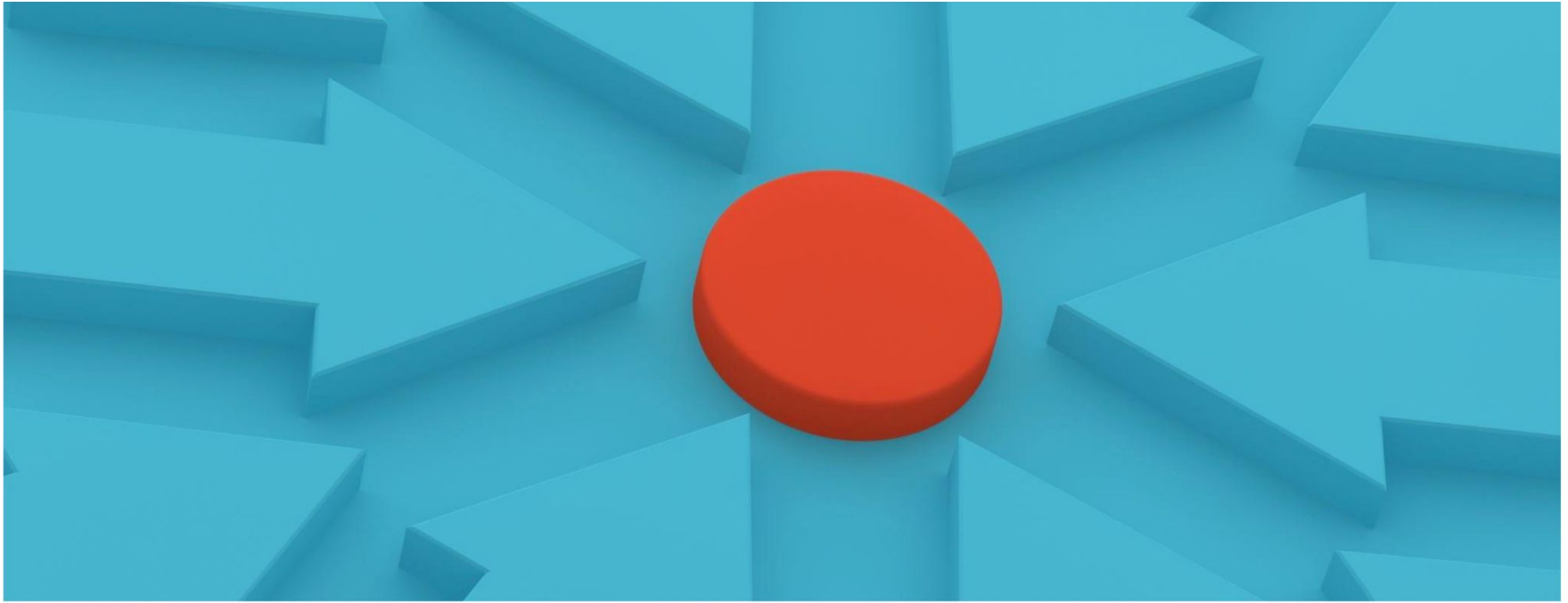
Vulnerable Code

```
void f(const char* input) {
    char buf[10];
    // No checking of length
    strcpy(buf, input);
    return;
}
```



THE UNIVERSITY OF
SYDNEY

3. Buffer Overflow Attack



Exploiting a Stack-Based Buffer Overflow

Three Steps

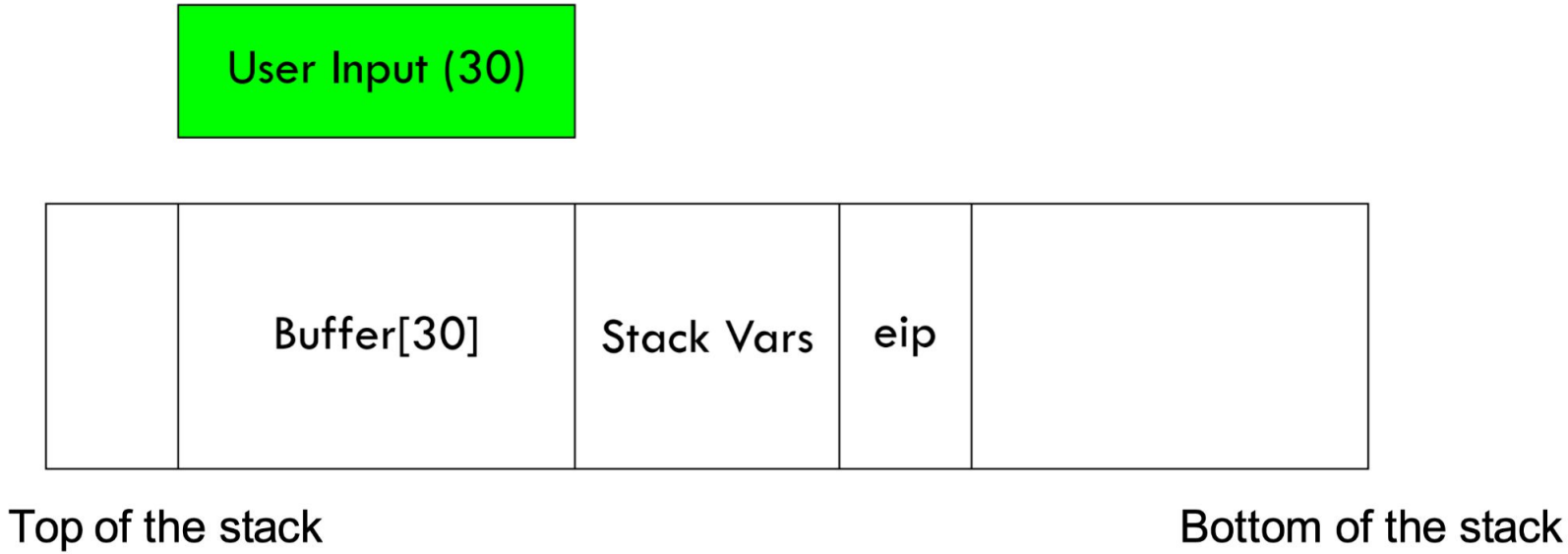
1. Insert malicious input - that overruns the allocated buffer space.

2. Jumping to and executing malicious code - overwriting the return address; point it to the address of malicious code (‘shellcode’)

3. Exploit system - shellcode can allow the attacker to run arbitrary code under the current user

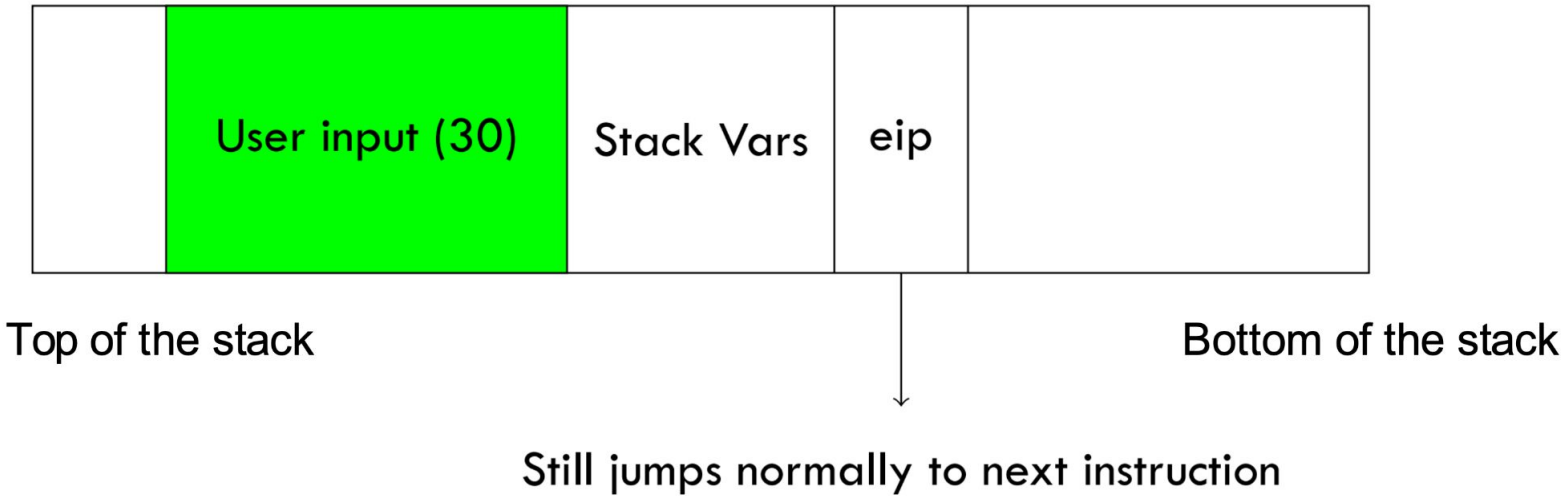
Typical Diagram of the Stack

- A typical scenario—intended usage of buffers in C



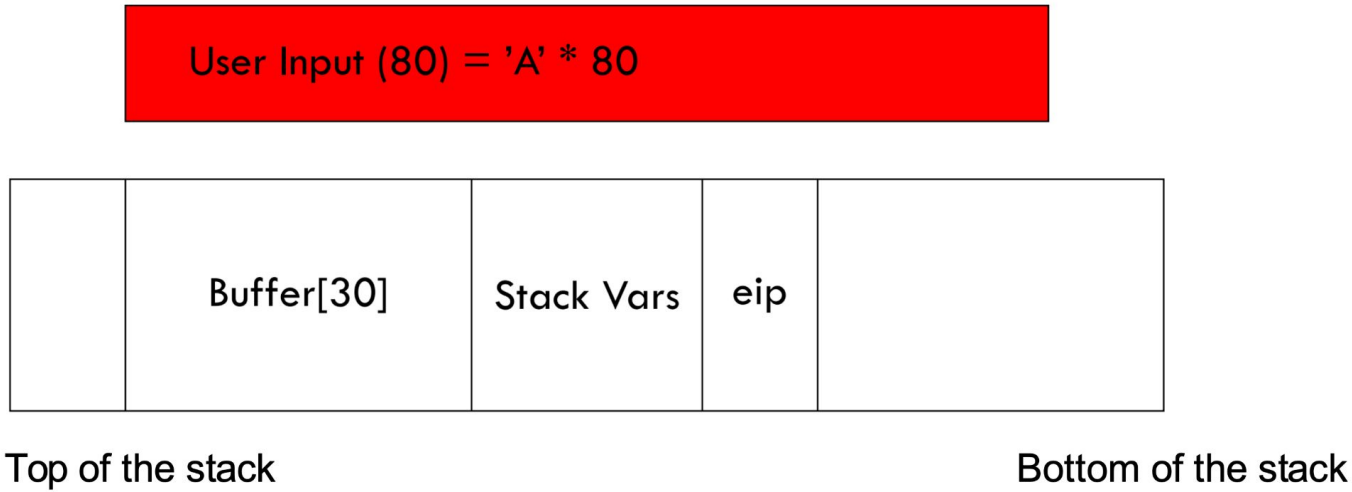
Typical Diagram of the Stack

- A typical scenario—intended usage of buffers in C



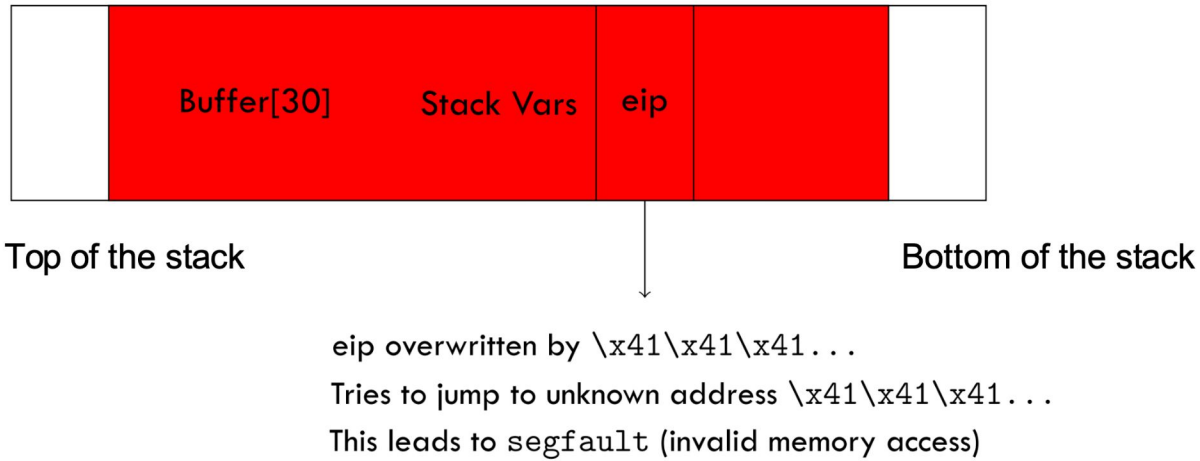
Typical Stack Overflow

- A function does not check memory bounds and user input overflows the buffer, resulting in segfault.



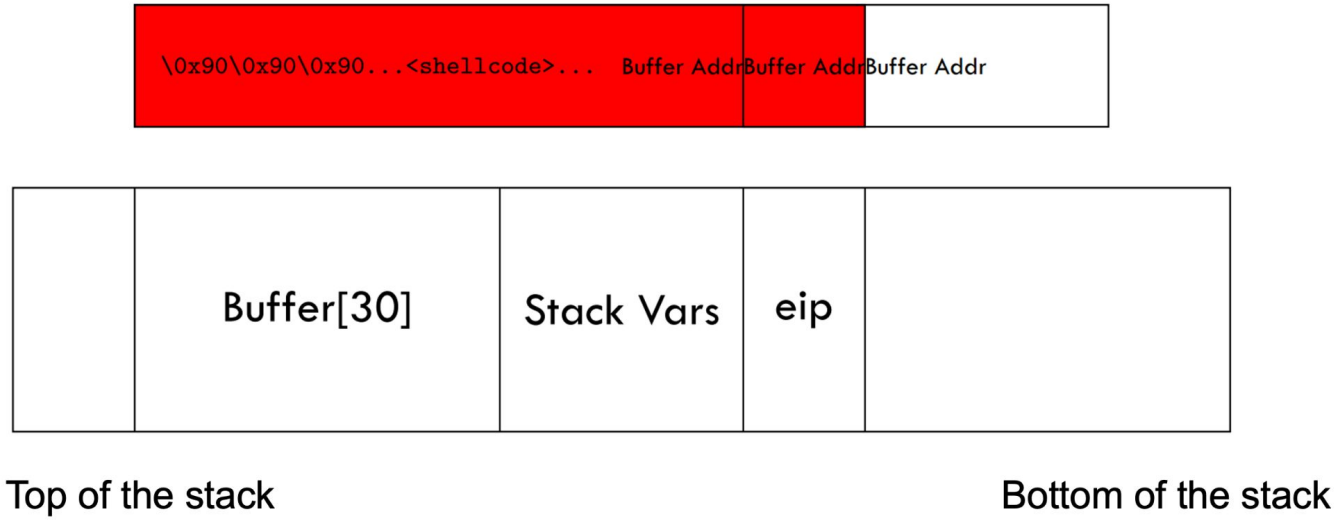
Typical Stack Overflow

- A function does not check memory bounds and user input overflows the buffer, resulting in segfault.



Exploiting a Stack Overflow

- **Idea:** find the byte offset to overwrite **EIP**, then overwrite it with start address of Buffer[30] ← attacker-controlled content!



Why the NOP and Extra Buffer Addresses

- The stack frame is created dynamically - and its layout depends on compiler settings.
- It is hard to predict absolute addresses of EIP and the start of Buffer.
- But their locations can be approximated by trial and error
- Adding many Buffer Addr after shellcode ensures we'll hit the EIP
  - `\0x90\0x90\0x90\0x90` is Intel instruction for 'Do Nothing'.
- Adding NOP before shellcode means that the return address, we put into EIP can be reasonably imprecise.
  - When NOP is encountered, it will move to next instruction, until shellcode is reached.
  - Can also use instructions that have similarly inconsequential effect (e.g., moving register to itself, or adding 0)

Exploiting a Stack Overflow

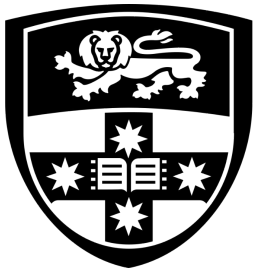
- **Idea:** find the byte offset to overwrite **EIP**, then overwrite it with start address of Buffer[30] ← attacker-controlled content!



1. eip points to start of buffer
2. \x90 no-op instruction (NOP), continues
3. executes shellcode - giving attacker control

Writing Shellcode

- Shellcode that are be input to the buffer should be written in machine language (binary) of target system.
- Must not contain null characters (`0x00`) - this is interpreted as end of string, and `strcpy()` stops on null character.
- You can write a code with assembly language of target system (readable by humans) first.
- Today, tools exist to generate shellcode (in binary) automatically (check out Metasploit); also takes care of getting around the null character issues.



THE UNIVERSITY OF
SYDNEY

Defeces Against Buffer Overflows

- **Alternative languages**
  - Memory-safe languages such as Java, .NET, Python, do not allow direct access to
  - memory addresses and can prevent this entire class of vulnerabilities.
- **Code Analysers**
  - Static code analysis
  - Dynamic code analysis
- **Compiler level**
  - StackGuard - Canaries
  - StackShield - Copy return address to safe area.
- **Operating system level**
  - Data Execution Prevention (DEP)
  - Address Space Layout Randomisation (ASLR)

Code Analysis

Static Code Analysis

- This type of tool tends to be used during implementation phase, e.g. release of a version will undergo a scan.

- Taint analysis
  - Finds variables that could have been ’tainted’ by user input
  - Identifies commonly vulnerable functions i.e. ’sinks’
  - Alert if tainted variables don’t go through sanitisation before reaching sink.
  - → Can be effective for vulnerabilities not caught by manual code review.

- **Limitations**:
  - Usually high number of false positives
  - Hard to detect vulnerabilities like broken authentication

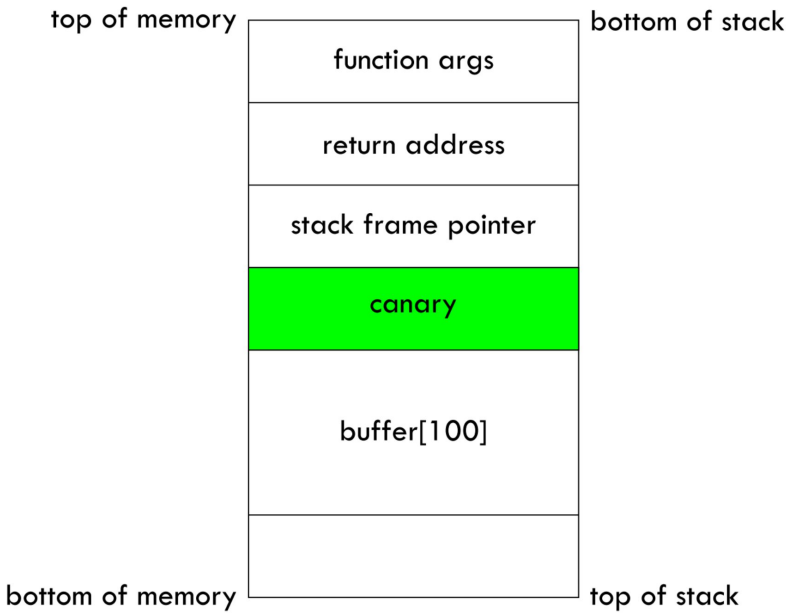
Compile Level

StackGuard - Canaries

- Mark buffer boundary with random value when function is called.
- Value is checked before function exits
- Idea: in order to overwrite return address, canary also has to be overwritten, enabling detection of buffer overflow.
- Trade-off, degrades performance.
- Note: not 100% effective - the code to handle a detected change of the canary is also in memory - attacker can try to reach it

Compile Level

StackGuard - Canaries



Compile Level

StackShield - Backup Return Address

- Copies return address to another area of memory when function is called.
- At the end of function, return address is checked.
- Even if return address is different, original return address is used.
- As with canaries, bypasses are possible. But canaries can detect corruption of other values as well, so they are considered a bit more useful.

Operating System Level

Address Space Layout Randomisation (ASLR)

- Basic Idea:
  - Exploiting buffer overflows requires knowledge of address of malicious code.
  - **Idea:** randomise addresses across memory to make it difficult to guess memory address of malicious code.
- Advanced techniques can bypass even this.

Operating System Level

Data Execution Prevention (DEP)

- Basic Idea:
  - Marks any code in the stack as non executable.
  - If code were to execute within these areas, OS level exceptions thrown.
  - However, can be defeated by Return Oriented Programming (ROP) e.g. return to libc attack
  - Instead of relying on injected shellcode, use code within the operating system to invoke shell.
  - Point return address to address of `'/bin/sh'` on OS

Using Safer APIs

- Examples of C/C++ API that have often led to exploits—and variants that encourage you to think about buffer lengths:

Dangerous	Defensive	Recommended Approach
gets()	fgets()	Don't use
strcat()	strncat()	Check length of buffer, null chars
strcpy()	strncpy()	Check length of buffer, null chars

- **Note**: The 'n' versions merely encourage defensive coding by requiring an explicit length parameter. They still can omit terminating null characters and cause problems.
- Some UNIX derivatives (e.g., OpenBSD) ofter 'l' versions (e.g. strlcpy()). These avoid the overrun but will silently omit data (another source of error).

Programming Errors: Null Termination

- A null-termination error occurs when a program using C-style strings either:
  - Fails to include a null terminator at the end of a string, or
  - Misinterprets the position of the null terminator.
- Because C doesn't store a string's length explicitly, functions like `strcpy`, `strlen`, or `printf("%s")` rely on scanning memory until they find a `0x00`.
- If the string isn't properly terminated, these functions keep reading or writing past the intended buffer, causing buffer overflows or information leaks.
- Even the 'n' versions may fail if the n given is less than the actual input, as the string in the buffer is no longer null terminated.

Programming Errors: Integer Overflows

- Can occur when storing or calculating a value that is too big to fit in a variable.
- Can also occur when comparing signed and unsigned integers.
- The size of the result is truncated to fit into the variable's maximum size.
- Why is it dangerous?
  - Like buffer overflows, silently ignored by compilers.
  - Hard to detect even after it has occurred, until fatal crash.
  - Found to be root cause of crash of Ariane 5 Rocket in 1996
  - Also found in software of Boeing 787

Programming Errors: Integer Overflows

```
a = 0xffffffff, b = 0x1
r = a + b
```

Instead of expected result:

```
r = 0x100000000 = 4,294,967,296
```

Actual result is calculated to ‘overwrap’ to back to 0:

```
r = (0xffffffff + 0x1) % 0x10000000
r = (0x100000000) % 0x100000000 = 0
```

Programming Errors: Format String Errors

- Format string controls output of print function, e.g.

```
printf("Number one = %d\n", 1)
```

- The function takes in a format parameter, %d, which passes a decimal value: 1
- There are several other format parameters such as:
  - `%n` - number of bytes as reference
  - `%x` - hexadecimal of the value
  - `%u` - unsigned int passed as value
  - `%s` - string passed as reference
- These format parameters are stored on the stack as printf is being executed.

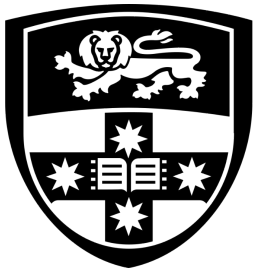
Programming Errors: Format String Errors

When user input is fed into this function, this can lead to mismatch in format parameters and values

- **Crash the program**
  - `printf("%s%s%s%s", user_input)`
  - Fetches string references from memory, whether they are there or not
  - If not there, will crash during runtime.
- **Arbitrary read from memory**
  - `printf("%08x %08x %08x")`
  - Reads 8 bytes of hex from stack, regardless of current function scope
- **Arbitrary write (integers) to memory**
  - `printf("12345%n", &val)`
  - Number of bytes written is stored in val.

Programming Errors: Use After Free and Double Free

- Memory control in C/C++ allows us to return memory to the OS for reuse
  - C: `free()`; C++: `delete()`
  - Immediate problem: which part of the program is responsible for doing this? Particularly, if there are threads?
- Problem: what if I reference the memory location via some other pointer?
- **Use-After-Free**: Accessing memory after it has been deallocated.
- **Dangling Pointer**: A pointer that still references a memory location that has been freed or is invalid.
- These would result in undefined behaviour

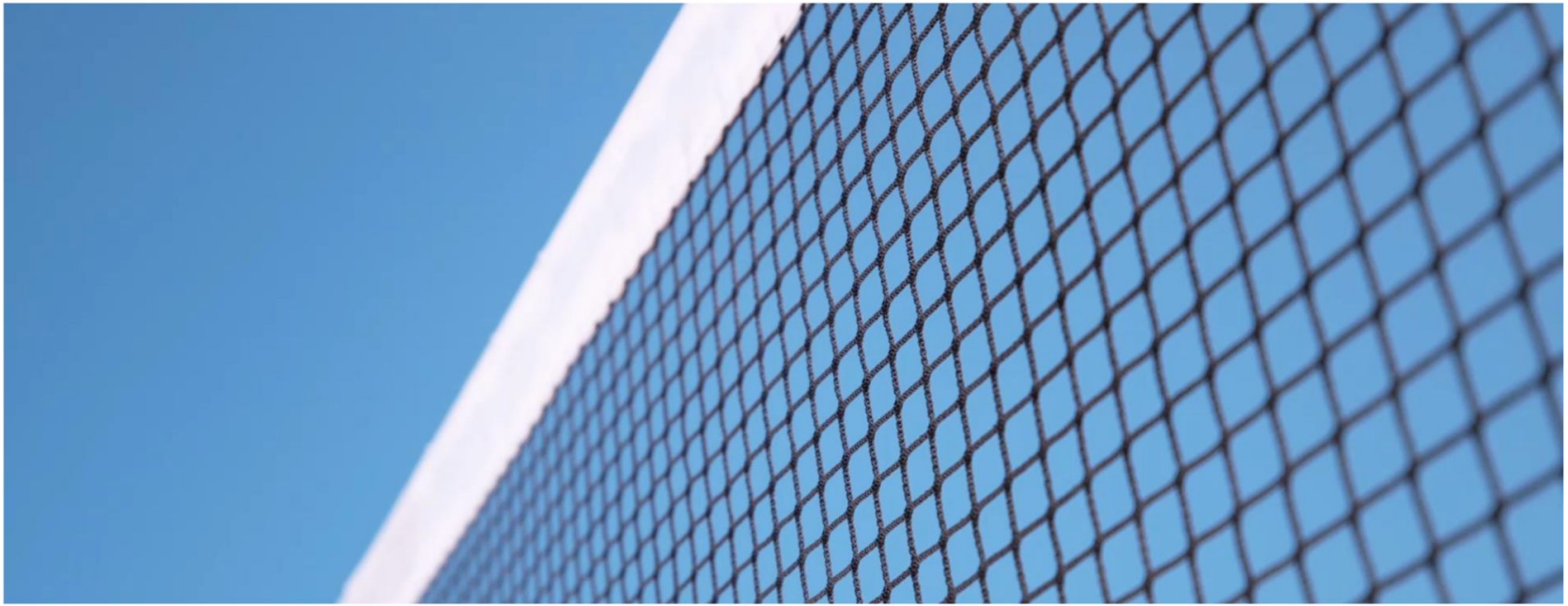


THE UNIVERSITY OF
SYDNEY

Type Safety

- Type safety ensures that a program only performs operations on compatible types.
- **Compile-time (Static) Type Safety**:
  - Detected by the compiler before execution.
  - Examples: Java, Haskell, C#
  - Advantage: Many type errors are caught early → safer and easier to fix.
- **Runtime (Dynamic) Type Safety**:
  - Checked while the program executes.
  - Examples: Python, JavaScript, Ruby
  - Risk: Errors may appear only under certain runtime conditions → harder to detect, potentially exploitable.

4. Defences Against Software Vulnerabilities



Software Lifecycle Security

- Ordinarily, vulnerabilities are thought to be introduced by flaws in code logic, e.g. not considering a particular edge case.
- Vulnerabilities can manifest during any stage of software lifecycle:
  - Development - e.g. use of third party libraries
  - Installation, Deployment - e.g. insecure default configurations
  - Maintenance - Upgrade, Patching - e.g. vendor hacked to distribute malware
- From an attacker’s perspective, a highly effective way of spreading malware and stealing secrets at scale.

Software Supply Chain: Third-Party Libraries

- Third party libraries reduce the amount of time needed to program functions
  - Often distributed through package managers backed by online repositories
    - e.g. `pypi`, `npm`, `rubygems`, `Maven Central`, etc.
  - Experienced massive growth in the past 10 years

Software Supply Chain: Package Managers

- By using third party libraries, there is an implicit dependency on the security practices of the library's developers:
  - To protect write access to their code
    - Eslint hacked to steal credentials! https://www.theregister.com/2018/07/12/npm_eslint/
- For example, not using 'password' as the password
  - A study of weak npm credentials (https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md
  - Koa, a JavaScript framework with 160,000+ weekly downloads had password=password.
- To keep their own dependencies patched and up to date.
  - i.e., Package A lists Package B as direct dependency. Package B depends on a package C affected by vulnerability.
  - Package A transitively depends on Package C and may be vulnerable

Software Supply Chain: Containerization

- Similarly, malicious images can be distributed through DockerHub, a repository for docker containers.
  - Cryptominers deployed over Docker: https://threatpost.com/malicious-docker-containers-earn-crypto-miners-90000/132816/
  - Running automated scripts found on the internet means that a developer can unknowingly deploy cryptominers on their infrastructure!
  - **Untrusted input does not always come in through web forms.**

Risk Mitigations

- **When depending on third party libraries and software:**
  - Dependency checker tools to fend against known vulnerabilities.
  - e.g. GitHub dependency graph alerts: https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts
  - OWASP dependency check: https://owasp.org/www-project-dependency-check/
  - npm audit: https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities
- Update and patch third party software under your control regularly - much easier said than done.
- **When developing libraries others use:**
  - Use password managers to generate unguessable passwords
  - Protect commit access with multi factor authentication

Zero Days

- Up until now, all the vulnerabilities and mitigations have been about **known vulnerabilities**
- What about vulnerabilities that haven’t been disclosed yet?
- How can we detect something that we don’t know?

Zero Days Risk Mitigations

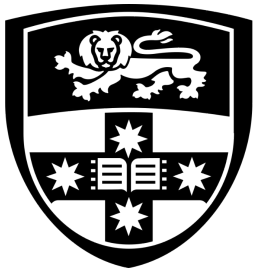
- **Honeypots:** a server that is masked to look like an unsecured server.
  - Can be used to study attacker's tactics and motives within network.
  - Used for detection: when credential from this server is used, alerts can be issued.

- **Behavioural analysis:** What common sequences of commands do hackers run after compromising hosts? e.g. whoami, cat /etc/passwd

Recap

- **We discussed:**
  - Basics of Firewalls
  - The problems in software security: language safety and memory safety
  - Memory segments in general and in C
  - Stack frame, register
  - Buffer overflow: stack-based buffer overflow
  - How to exploit a typical stack-based buffer overflow
  - Various ways to defend against buffer overflow
  - Programming errors that create vulnerabilities
  - Software supply chain
  - Risk mitigations for other vulnerabilities
  - Zero days





THE UNIVERSITY OF
SYDNEY