Programming language & software security

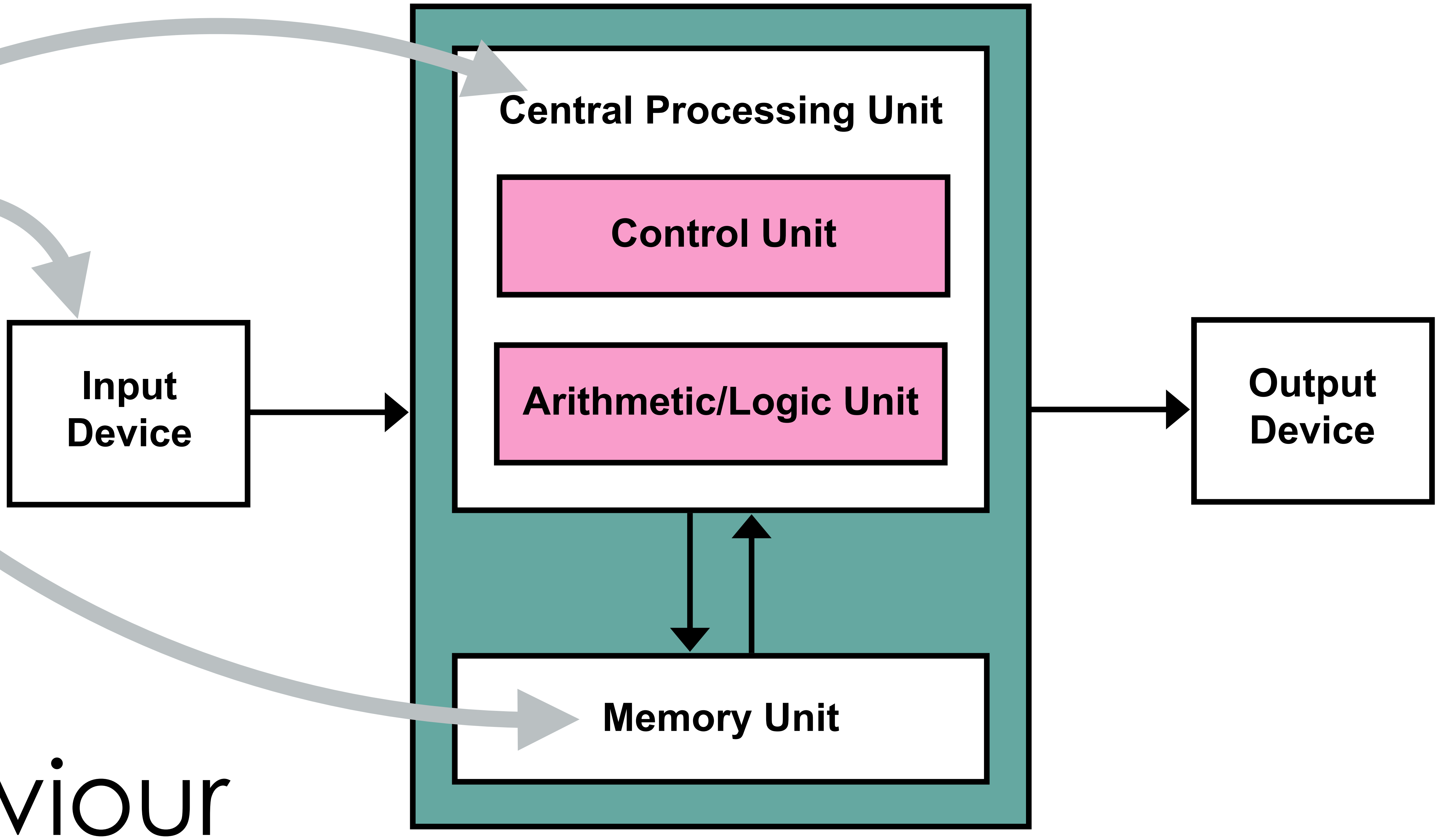
COSC312 / COSC412

Learning objectives

- **Choice of programming language** can affect security
  - … although choice of PL is almost certainly not a panacea
- High level view of causes of **PL/software security issues**
  - Provide a roadmap into which to fit common attack types
- Many **software security checks** can be regimented
  - Also that there are many, many vectors for security attacks!

Typical computing machine model

- Programming language security depends on machine:
  - we'll assume typical **von Neumann architecture** depicted
- **CPU** runs imperative code
- **I/O** devices (input/output)
- **Memory** (code and data)
  - hierarchy of memory levels
- First we'll focus on CPU behaviour



Security from machine model's perspective

- Risks in terms of the CPU going awry:
  - **Space**—CPU interacts with memory in unintended manner
    - *e.g.*, 'buffer overrun'—a data structure **overflows its allocation**
  - **Time**—CPU interacts with resources no longer validly, *e.g.*,
    - 'use after free'—a resource that **was deallocated is used**
    - 'TOCTOU' races—security property **check decoupled from use**
- Both approaches can affect either/or:
  - **code**: CPU ends up running software it shouldn't
  - **data**: CPU reads and/or writes data that it shouldn't

Memory model for programming

- CPU's machine code **needn't be procedure oriented**:
  - e.g., code can (potentially conditionally) jump to other code

- Programming languages (PLs) usually more structured:
  - **Stack**: FIFO; memory use lifecycle connected to PL functions
  - **Heap**: memory use lifecycle decoupled from program flow

- CPUs very likely to support **call stack** explicitly
  - e.g., dedicated CPU registers for managing stack frames

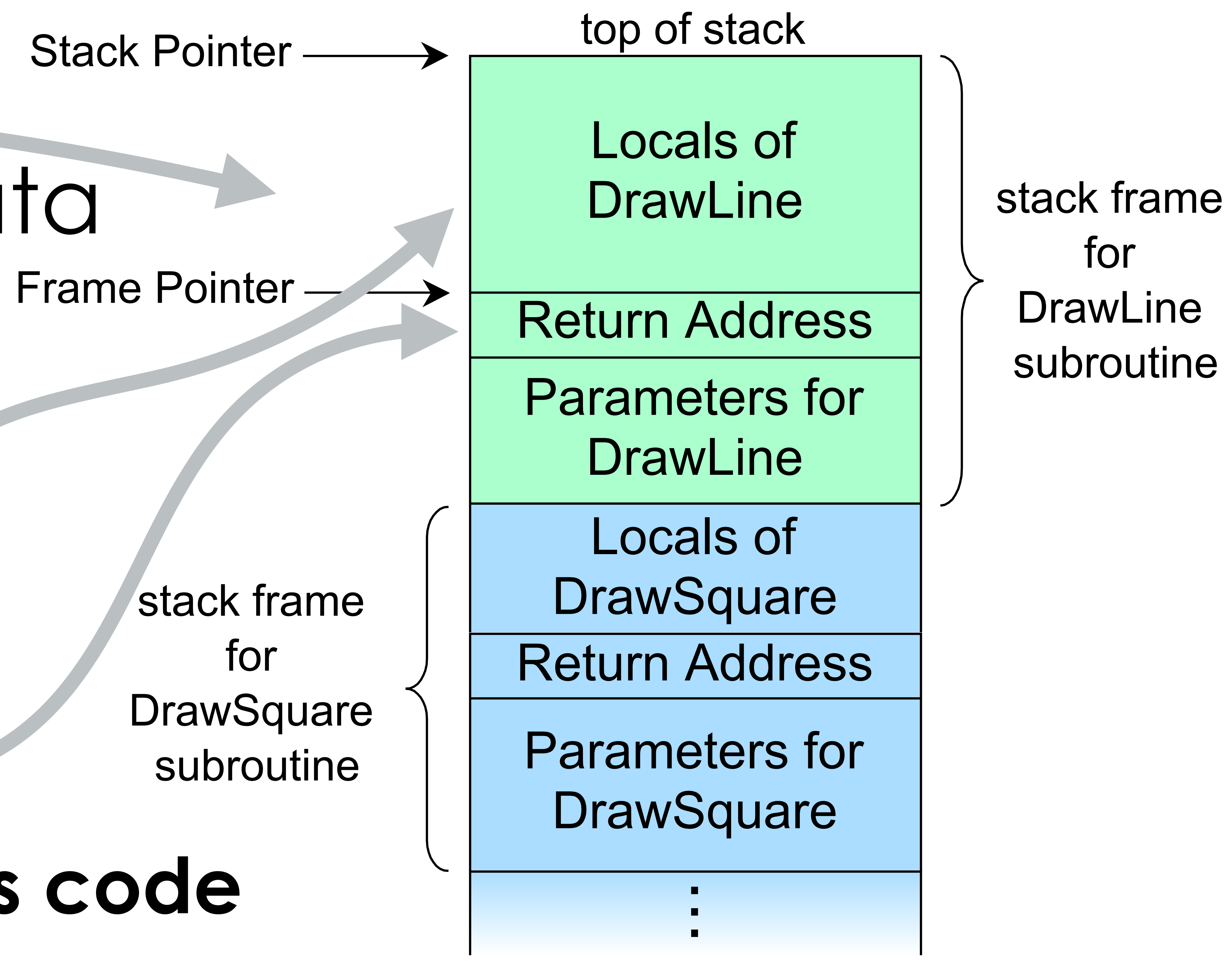
Heap-based (space) attacks

- Consider a C program that `malloc`'s two 16 byte arrays
  - These arrays will be **allocated on the heap** (by libc and OS)
  - Pretend the arrays are allocated in sequential addresses

- Buffer overflow attack: (oversimplified)
  - If read/write index to first array **not bounds checked** to be <16
    - … then reads/writes on first array actually affect second array
    - Real instances have overwritten program code rather than data
  - `strcpy` copies C strings without bounds check; use `strlcpy` !

Stack-based (space) attacks

- Call stack pertains to code & data
  - Data: **local variables**, **parameters**
  - Code relevant: **return address**



- Buffer overflow a local variable?
  - potentially **rewrite return address**
  - function returns **control to attacker's code**

- Note: stack address growth direction is CPU-dependent
  - x86 grows downwards (overflow of locals will reach return addr.)

Operating system (OS) memory protection

- Most CPUs have a **memory management unit** (MMU)
  - Exception is old architectures and smaller embedded systems
- MMU implements **usage restrictions on memory pages**
  - Pages are often 4KiB blocks of memory
  - Effects isolation of different operating system processes
  - Also separates applications from underlying OS kernel
- **Privilege escalation** attacks: into kernel from user code

Stop code being executed from data pages

- Execution space protection: **split code/data memory**
  - Prevents data pages having code executed from them
  - Represents OS+CPU increasingly **locking down memory**
  - Must couple with **address space layout randomisation** (ASLR)
- OS loader configures memory protection for app.
  - Application built with **clearly separated regions** (c.f., ELF)
- Mitigates attacks where **buffer overrun modifies code**
  - CPU capability: `NX` bit in AMD and `XD` bit in Intel CPUs

Code gadgets—attacks using existing code

- Execution space prot. stops attacker injecting code
  - … however there’s **already lots of code** on any target system
- Attacker can scan for ‘**gadgets**’: abuse existing code
  - e.g., can jump into the middle of a destructive library function
  - may even be able to control parameters for those functions
- Significantly **raises the difficulty** of performing attacks
  - … but many attackers are well resourced, and patient …

Return oriented programming (ROP)

- Shown that **call stack attacks can chain gadgets**
- Attacker modifies return address and parameters
- Attacker isn't introducing code: changing return addr.
- However net **effect is close to code injection**

- Not straightforward to distinguish attacks
  - solutions have proposed **integrity checks on return addresses**
  - … but need to ensure that overheads are worth the expense

Vulnerabilities from parsing bugs

- **Parsing structured data** from simpler data can be risky:
  - Typical example, **SQL injection** (see next slide)
  - … but also watch entities such as **file paths stored in strings**

- Many **forms of data parsing** are commonplace, *e.g.*:
  - **XML** documents
  - **Unicode** strings—*e.g.*, UTF-8
  - **URIs**—*e.g.*, wherein spaces are replaced by `%20` within in URLs
  - **Serialisation** of program objects (*e.g.*, Java, Python, …)

SQL injection

- (You've likely encountered this concept previously…)
- Want to submit query to database, **code builds a string**
- However `SELECT * FROM t WHERE t.name='$VAR'` is risky:
  - `$VAR` needs to be checked to **stop it escaping SQL statement**
  - e.g., `$VAR` should **not contain single quotes**
    - consider `$VAR` being `Robert'; DROP TABLE t` (c.f. XKCD comic 327)

- A solution: **SQL prepared statements** (`?` is placeholder)
  - `SELECT * FROM t WHERE t.name=?`—later bind variable to `?`

File paths and potential security risks

- Common practice to store **file paths in string variables**
  - Pain may be on offer regarding directory separator slash style
- Actually, **paths are far more subtle** than string suggests
  - Most operating systems can mount filesystems at any subpath
  - Different parts of one path string may be **case sensitive or not**!
- Another risk area: **simplification** using lexical processing
  - e.g., `thisDir/aDir/../otherDir`—what if `aDir` is a symlink?

XML vulnerabilities

- Be careful parsing **untrusted source code**, here for XML
  - **Entity attacks**, such as “Billion Laughs”:
    - `<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">` *etc.*
    - Very small file can easily explode to **occupy impractical resources**
  - External entity resolution: parser looks up **remote URLs / files**
  - **XSLT is Turing complete**! (XML stylesheet transformation)

- Use an **existing parser** implementation
  - … and continue installing its (likely many) **security updates**

Unicode handling can cause security issues

- Unicode: standard representing language characters
- UTF-8 is **variable-width** ASCII-compatible 8-bit encoding
  - Upper 128 values include mode shifts to multi-byte characters
  - Combining characters affect other characters:
    - *e.g.*, <i `U+00ED`><diaeresis on previous `U+0308`> versus <ï `U+00EF`>
    - normalisation required to switch to longest / shortest form

- Security risk can relate to visual confusion or encoding
  - *e.g.*, **normalisation failure** may lead to incorrect equality tests
  - Further risks **when embedded** in other forms, *e.g.*, URI encoding

PL support for secure parsing

- Some PL syntax can **include XML directly**, *e.g.*, Scala
  - Scala's parser accepts XML as the RHS of assignment:
    - `var myVariable = <p>A simple XML tree</p>`
  - (Scala is used in industry: *e.g.*, LinkedIn, Twitter, Airbnb, Netflix, …)

- PLs may also help security of **processing Unicode**:
  - *e.g.*, Apple's Swift language ensures Unicode-correct handling
  - Would be non-idiomatic code to look at Swift string's bytes

- (Past project of mine added SQL&XML parsing to Python!)

Programming language choices for security

- Some situations require use of low-level, ‘unsafe’ PLs
  - *e.g.*, directly **driving hardware devices** may need assembly
  - Most **mainstream OSs** have been largely coded in C/C++
- Applications can choose interpreted or compiled PLs
  - Security concerns are different, but **both have OS interfaces**
  - Compiled: result may have have machine code vulnerabilities
  - Interpreted: likely rely on ‘foreign’ function interface (*e.g.*, C)
    - *e.g.*, Python often effectively logical glue between C code libraries

Runtime support for managed PLs

- Manually performing **memory management** is riskier
  - … although also necessary for corner cases
- Useful to seek **runtime systems that manage resources**
  - *e.g.*, lifecycle of heap objects; OS interactions
  - Pay the 'price' of not directly controlling CPU with your code
- Java Virtual Machine: garbage collection; serialisation
  - JVM is now a target platform for other languages (*e.g.*, Scala)

Functional programming languages

- Pure functional PLs **don’t have intermediate state**
  - *e.g.*, Haskell—variables are labels, not memory pigeonholes
  - … but PLs have to **interact with underling OS** so pass state to fns:
    - Monads wrap functions and return values into efficient pipelines
- Many functional PLs are ‘impure’ with some state
  - *i.e.*, state will likely involve mutable data structures
- DNS server ported to OCaml was more efficient than C
  - Allowed safe reuse of memory where C code made copies

D (dlang) programming language

- Builds **pragmatic extensions** over C++
  - Bring desirable high-level functionality to low-level language
    - Aims to be as efficient as equivalent C++ but **terser and safer**
    - Still **supports inline assembly** (unlike C#, Java, *etc.*)

- Features that help security include integration of:
  - **Bounds-checked arrays**; garbage collection; strings are arrays
  - **@safe annotation** ensures valid lifetime of references
    - Compile-time check to **preclude use-after-free** types of errors
    - **'Better C' subset** removes D runtime, keeps bounds-checking, *etc.*

E—OO, secure, distributed PL

- Likely you'll never see/use E, but it is very well designed

- Method call = sending message to local/remote object
  - immediately—essentially like a function call (synchronous)
  - deferred—asynchronous, **caller gets 'promise'** immediately

- **E objects are capabilities**, actually: controls visibility
  - Can use **sealer/unsealer pairs** to lock down object access
  - Can include guards to **check runtime conditions** (`balance>=0`)

Rust—low-level PL more secure than C

- Began in Mozilla: e.g., for Servo secure browser (RIP)
  - Gaining adoption in Linux kernel alongside C
  - Benefits in its **use of LLVM compiler framework** over C/C++

- Key feature is the notion of **ownership typing**
  - If caller passes object to callee, caller can't modify it anymore
  - Rust borrow checker: ownership violations are **compiler errors**

- Rust provides built-in **build system** & **package manager**

Engineering secure software

- Need security functionality? **Use existing libraries**!
  - *e.g.*, NaCL; XACML; SAML; Kerberos GSS-API
  - … you also need to **assess dependencies** and **apply updates**

- Apply defence in depth: **multiple layers of security**
  - Interacting with filesystem? Try to add a **chroot**
  - Handling sensitive data? Apply **encryption** defensively
  - Database to be read only? Make it a **read-only replica**
  - Trade off additional computing cost for extra security
    - Use **short-lived OS processes** rather than risking memory leaks

In summary

- Described typical **machine model** and causes for **security problems** in time and space
- Outlined **machine code attack & defence evolution**
  - Discussed numerous common attack vectors
- Indicated how **choices of PL** can help security
  - New languages are still being developed…