Previously

Stages of code injection

1. Inject code
2. Hijack control flow

But step 1 is getting harder!

Why?

What if...

~~0. Inject code~~

1. Hijack control flow

What code do we execute?

ASLR

Address Space Layout Randomization

Not super-helpful on 32b platforms

Increases "work factor"

But maybe not by as much as you think!*

---

* "ASLR on the Line: Practical Cache Attacks on the MMU", Gras, Razavi, Bosmen, Box an Giuffrida, *Proceedings of the 2017 Networked and Distributed Systems Security Symposium*, 2017. DOI: https://dx.doi.org/10.14722/ndss.2017.23271.

Defenders can make the attacker's life harder by ensuring that `libc` (and other code) isn't loaded at the same location every time.

On a 32b machine, however, we might only have 16b or even 8b available for randomization. A lack of randomness *seems* bad in a defensive technique called "randomization", but why? What would more randomness give us?

ASLR ____. Unlike other security techniques, it won't always say "no" to an attack. What it will do is make an attacker have to do ______________________. For example, on a 32b system, an attacker might have to ____ in order to succeed.

Practical attacks exist that use low-level properties of things like memory management units (MMUs) to break ASLR, even from JavaScript code!

Code reuse attacks

~~0. Inject code~~

1. Hijack control flow

How do we stop the hijacking?

Stopping hijacking

Stack protection

Canaries (`-fstack-protector`)

CFI: control flow integrity

Static analysis, dynamic enforcement

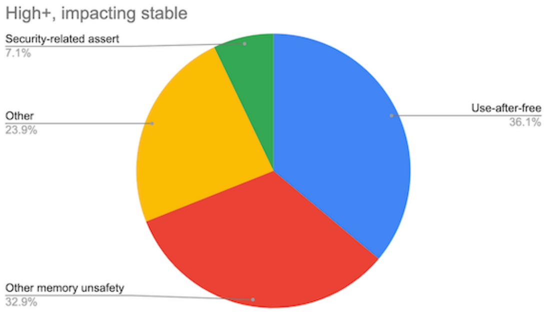
Full *memory safety*

... which we'll discuss next time!

Memory safety

How can we perfectly prevent such attacks?

- write perfect software!
- *memory-safe* languages (**partial** answer)



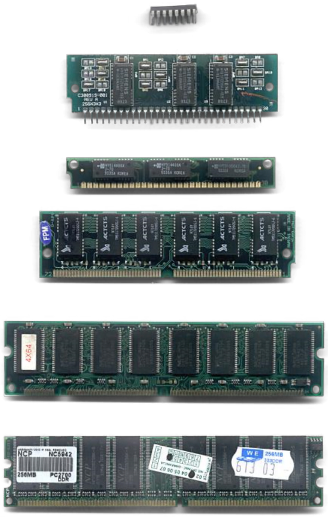
Source: Chromium project

Program execution

Q: how do we load a value from memory?

A: it depends on the language!

- compiled
- interpreted
- bytecode-interpreted



Different languages provide for different modes of memory access.

How do we categorize languages?

- programing **paradigm** (OO, functional, etc.)
- memory management (manual vs **garbage-collected**)
- **compiled** vs **interpreted**

Compiled languages

Examples?

Where are memory access decisions made?

Examples of languages that compile to machine instructions: _________, ___________, ______________, __________, ______________, ___________...

The _______________ may prevent certain kinds of accesses at compile time. For example, some code is supposed to be able to access __________________ but other code isn't (see example: private.cpp). However, at runtime, all we have are _________________________ that ___________ and ___________ values.

Bytecode-interpreted languages

What's different?

Why?

A bytecode-interpreted language (e.g., anything that runs on the ____________) includes a _____ ______________ for its bytecode. Instead of interpreting Java or Scala, those languages can be compiled to the Java bytecode format, which is executed by a lower-level _________ _________. This is also true for ______________ _____: you can compile languages like _________, __________, ____________ and ____________ (see: https://github.com/appcypher/awesome-wasm-langs) into _______________ _____ and then execute the result in any Web browser with much greater speed than interpreting from source.

In a bytecode-interpreted language, we get some of the benefits of compilation, e.g., we don't have to parse a bunch of program text every time we run the program. We *also* get some of the benefits of an interpreter, such as ___________________ __________ ____________! That means we can't, for example, walk off the end of an array.

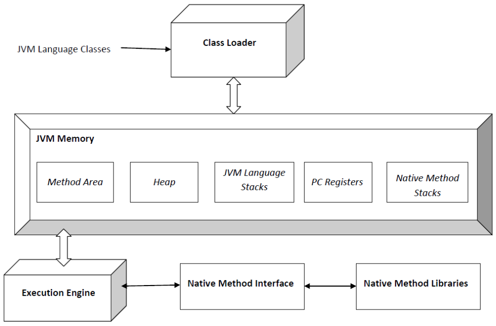
`Example: Java`

Memory management

Memory access

Bytecode and TCBs

`SecurityManager`



Li Gong *et al.*, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", in *USITS '97: Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1997.

However, there is no such thing as a free lunch. One of the costs of using any sort of interpreter is that the interpreter becomes ______________________... and thus we tend to have a ______________________!

Java, in particular, also has interesting facilities for disabling features like reflection, which by design circumvent the normal type rules of the language.

So... perfection?

Write all software in a memory-safe language?

TCB considerations

Memory safety in compiled languages

1. Compiler-added run-time safety checks
2. Limited unsafety
3. Continued dangers of native instructions

High-level language interpreters have to be written in something. You might be able to write a lot of a Java interpreter in Java, but at the lowest levels you will find lots and lots of C++ code. At the lowest levels of the C standard library, you will find ____________________, sometimes ___________________________.

Languages like __________ and ____________ claim to provide memory safety, but they are compiled languages. How is this possible?

The compiler can add extra code to check some accesses at run time. For example, if you are indexing within an array, the compiler can implicitly add code such as `if 0 <= i < n`.

Languages that aspire to "systems programming" (i.e., things that have to be aware of or manipulate the lowest-level primitives such as hardware registers) have to allow for unsafe operations. There is no memory-safe way to perform arbitrary register, memory or I/O operations, so these kinds of languages have to provide some way to break abstraction layers. C code can include assembly via the `asm` keyword. Rust code can explictly violate memory safety guarantees if it uses the `unsafe` keyword.

Even with those checks, however, if you load someone else's native instructions and execute them, ____________________!

Safe compiled code?

What is a language?

Software

AddressSanitizer, CCured, Cyclone, "fat pointers", Go, Rust, ...

Hardware:

Arm MTE, CHERI, Hardbound, MPX, segmentation, Watchdog, ...

When we think of a language, we typically think about ____________________ and the ______________ for writing it. However, in addition to ______________, we also have ______________ that are defined by language specifications and — crucially — ____________ ____________________. If we take this expanded view of what makes a language, we can see a number of approaches applied in various places that can be used to improve the security of compiled code, too.

Software

AddressSanitizer (and other "sanitizers" like Thread Sanitizer and the Undefined Behaviour Sanitizer) can help spot memory errors during testing that might otherwise have gone unnoticed. CCured is an example of an approach that uses static analysis to figure out how pointers in a C program are "meant" to be used and dynamic analysis to ensure that they are, in fact, used that way. Cyclone is a C dialect with better memory safety properties than vanilla C, which it is designed to be compatible with (or at least easy to adapt from). Newer languages like Go and Rust have more expressive type systems that make it possible to write memory-safe code even in high-performance compiled languages with limited run-time checking.

Hardware

Arm MTE has been adopted by Android to detect memory safety violations at run time. Hardbound, MPX and Watchdog attempt to provide various forms of hardware memory safety enforcement. CHERI is a designed-for-security instruction set extension for ARM and MIPS that is just about to ship its first hardware prototypes; it has the potential to change ________________ by allowing high-level object accesses to be precisely enforced by hardware.

Summary

Memory safety

Memory-safe language concepts

Safe unsafe languages?