Approaches for Ensuring Memory Safety

- Use a memory-safe language ("safe by design")
- Use a non-memory-safe language, and check bounds in your code
- Use a non-memory-safe language, and harden the code against common exploits
Memory safe languages

- You can spare yourself this work by using a memory-safe language
  - Turns "undefined" memory references into an immediate exception or program termination
  - Now you simply don't have to worry about buffer overflows and similar vulnerabilities

- Plenty to choose from:
  - Python, Java, Go, Rust, Swift, C#, … Pretty much everything other than C/C++/Objective C
Code Hardening
Defending Legacy Code

- A large back-and-forth arms race trying to prevent memory errors from being *exploitable for code injection*
  - An attacker can still use them to crash the program
  - An attempt at defense-in-depth

- Stack Canaries
- Non-Executable Pages (aka DEP or W^X)
- Address Space Layout Randomization (ASLR)
Stack Canaries

- **Goal:** protect the return pointer from being overwritten by a stack buffer…
- **Store canary before saved return address**
  - “Stack canary” = random value chosen when program starts
  - Function prologue pushes canary, epilogue checks canary against stored value to see if it has changed
Attacks on Stack Canaries

• Learn the value of the canary, and overwrite it with itself
  • e.g., a format string vulnerability, an information leak elsewhere that dumps it
• Random-access write past the canary
  • Canary only defends against consecutive writes
• Overflow in the heap
• Overwrite function pointer or C++ object on the stack
• Bottom line: Bypassable but raises the bar
  • A simple stack overflow doesn’t work anymore: Need something a bit more robust
  • Minor but nearly negligible performance impact
Attack: Guessing the Canary

• On 32-bit x86, the canary is a 32-bit value
  • It is 64 bits on x86-64

• One byte of the canary is always 0x00
  • Since some buffer overflows can’t include null bytes: e.g. if the vulnerability is in a bad call to `strcpy`

• This means you can (possibly) brute-force the canary
  • Need to try about $2^{24}$ times or so
Non-Executable Pages (aka DEP, W^X)

- Each page of memory has separate access permissions:
  - R -> Can Read, W -> Can Write, X -> Can Execute
- Defense: mark writeable pages as non-executable
  - Now you can’t write code to the stack or heap
- No noticeable performance impact
Attacks on Non-Executable Pages

• Return into libc: set up the stack and “return” to exec()
  • Overwrite stuff above saved return address with a “fake call stack”, overwrite saved return address to point to the beginning of exec() function
  • Especially easy on x86 since arguments are passed on the stack
• Return Oriented Programming
arguments
return address
saved frame pointer
exception handlers
local variables
callee saved registers
Arguments

Return address

Saved frame pointer

Exception handlers

Local variables

Callee saved registers

Arguments for exec

Return address for exec

Modified sfp
Return Oriented Programming

- Idea: chain together “return-to-libc” idea many times
  - Find a set of short code fragments (gadgets) that when called in sequence execute the desired function
  - Inject into memory a sequence of saved "return addresses" that will invoke them
  - Sample gadget: add one to EAX, then return

- ROP compiler
  - Find enough gadgets scattered around existing code that they’re Turing-complete
  - Compile your malicious payload to a sequence of these gadgets

- Tools democratize things for attackers:
  - Yesterday's Ph.D. thesis or academic paper is today's Intelligence Agency tool and tomorrow's Script Kiddie download
Address Space Layout Randomization

• Randomly relocate everything in memory
  • Every library, the start of the stack & heap, etc…
  • With 64-bit architecture you have lots of entropy
  • 32-bit? Hard to get enough entropy, as segments need to be page-aligned (i.e., start at a 4096-byte boundary), so attacker might be able to brute-force it
ASLR Efficiency

- Performance overhead is close to 0%
  - Everything needs to be *relocatable* anyway:
    Modern systems use relocatable code and dynamically load all the desired libraries
ASLR + DEP

• ASLR + DEP make many exploits harder
  • Typically, need two vulnerabilities: both a buffer overrun and a separate information leak (such as a way to find the address of a function)
  • Information leak needed to fill in the return addresses for ROP chain
Defense In Depth in ALSR + DEP: Attacker Requirements

- Attacker first needs to discover a way to read memory
  - Just a single pointer to a known library will do, however
    - The return address off the stack is often a great candidate
    - Or a vtable pointer for an object of a known type

- Armed with this, the attacker now can create a ROP chain
  - Since the attacker has a copy of the library of their own and has already passed it through a ROP compiler, it just needs to know the starting point for the library

- Now the attacker needs to write memory
  - Writes the ROP chain and overwrites a control flow pointer
Defenses-In-Depth in Practice

- Apple iOS uses ASLR in the kernel and userspace, W^X whenever possible
  - All applications are sandboxed to limit their damage: The kernel is the TCB
- The "Trident" exploit was used by a spyware vendor, the NSO group, to exploit iPhones of targets
- So to remotely exploit an iPhone, the NSO group's exploit had to...
  - Exploit Safari with a memory corruption vulnerability
    - Gains remote code execution within the sandbox: write to a R/W/X page as part of the JavaScript JIT
  - Exploit a vulnerability to read a section of the kernel stack
    - Saved return address & knowing which function called breaks the ASLR
  - Exploit a vulnerability in the kernel to enable code execution
Safari Exploit: More Details

• Basic idea: can corrupt a JavaScript object (due to interaction with garbage collector) to trigger a use-after-free issue
  • Attacker’s JavaScript has access to both objects that share the same memory:
    • Newly allocated object is an array of integers
    • Old object changes the length of the array to be 0xFFFFFFFF

• Now attacker has a "read/write" primitive
  • The array can see a huge fraction of the memory space
    • First thing, find out the offset of the array itself, then any other magic numbers needed

• Turning it into execution
  • Take another JavaScript object that will get compiled (the "Just In Time" compiler)...
  • That object's code pointer will point into space that is writeable and executable
Fuzz testing

• Automated testing is surprisingly effective at finding memory-safety vulnerabilities

• How do we tell when we’ve found a problem? Program crashes

• How do we generate test cases?
  • Random testing: generate random inputs
  • Mutation testing: start from valid inputs, randomly flip bits in them
  • Coverage-guided mutation testing: start from valid input, flip bits, measure coverage of each modification, keep any inputs that covered new code
Why does software have vulnerabilities?

• Programmers are humans. And humans make mistakes.
  • Use tools

• Programmers often aren’t security-aware.
  • Learn about common types of security flaws.

• Programming languages aren’t designed well for security.
  • Use better languages (Java, Python, …).
Some Approaches for Building Secure Software/Systems

• Run-time checks
  • Automatic bounds-checking (overhead)

• Code hardening
  • Address randomization
  • Non-executable stack, heap

• Monitor code for run-time misbehavior
  • E.g., illegal calling sequences
  • But again: what do you if detected?
Approaches for Secure Software, con’t

• Program in checks / “defensive programming”
  • E.g., check for null pointer even though sure pointer will be valid

• Use safe libraries
  • E.g. `strlcpy`, not `strcpy`; `snprintf`, not `sprintf`

• Bug-finding tools

• Code review
Approaches for Secure Software, con’t

• Use a memory-safe language
  • E.g., Java, Python, C#, Go, Rust

• Defensive validation of untrusted input
  • Constrain how untrusted sources can interact with the system

• Contain potential damage
  • Privilege separation, run system components in least-privilege jails or VMs