CSC 591
Systems Attacks and Defenses

Sandboxing Applications

Alexandros Kapravelos
akaprav@ncsu.edu

Native code

- Performance
- Legacy code
- Various languages

Run a random binary on my system?
No way!

Sandboxing Native Code

Trust the developer

- ActiveX, browser plug-ins, Java, etc.
- Code is signed
- Ask user if developer should be trusted
  - Good for known developers
  - Tricky for web applications

Hardware/OS sandboxing

- Virtual machines
- Containers
- Capsicum, seccomp
- OS kernel vulnerability
- OS incompatibility
  - System calls, threads, etc
  - Virtual memory layout
  - OS might not have a sandboxing mechanism
  - Might need to run it as root
- Hardware vulnerabilities

Software Fault Isolation

- Before running a binary, verify it's safe
  - Static analysis
    - Self-modifying code?
    - Overlapping instructions?
- Safe instruction
  - Math, mov, etc
- Unsafe instruction
  - Memory access
  - Privileged instruction
- How to deal with unsafe instructions
  - Instrument
  - Prohibit

Trusted Service Runtime

- Code that can be trusted and will perform the sensitive operations
  - Allocate memory
  - Threads
  - Message passing
- After verifying, safely run it in same process as other trusted code
- Allow the sandbox to call into trusted service runtime code

Safety

- No disallowed instructions
  - Syscall, int
- All code and data within bound of module
  - Module cannot corrupt service runtime data structures
  - Module does not jump into existing code
    - ret2libc
    - ROP
  - Everything else should be protected from the module

Checks

- Scan the binary and look for “int” and “syscall” opcodes
  - If check passes, can start running code
  - All code is marked as read-only
  - All writable memory is non-executable

Is this enough?

Check complications

- x86 has variable-length instructions
  - "int" and "syscall" instructions are 2 bytes long
  - Other instructions could be anywhere from **1 to 15 bytes**

```
25 CD 80 00 00
AND %eax, $0x000080cd
INT $0x80 # Linux syscall
```

Should we scan the binary from every offset?

Reliable Execution

- Ensure code executes only instructions that verifier knows about
- Scan forward through all instructions, starting at the beginning
- If we see a jump instruction, make sure it's jumping to address we saw
  - Easy to ensure for static jumps (constant addr)
  - Cannot ensure statically for computed jumps (jump to addr from register)

Computed jumps

- Add dynamic checks before jumps
- Checks for jumping to a register

```
AND $0xffffffe0, %eax # Clear last 5 bits
JMP *%eax
```

- Ensures that jumps go to multiples of 32 bytes
  - Longer than the maximum instruction length
  - Power of 2
  - Fits trampoline code
  - We don’t want to waste space
- `nacljmp`

Computed jumps

- No instructions span a 32-byte boundary
- Compiler's job is to ensure these rules
  - Replace every computed jump with the two-instruction sequence
  - Add NOP instructions if some other instruction might span 32-byte boundary
  - Add NOPs to pad to 32-byte multiple if next instr is a computed jump target
  - Always possible because NOP instruction is just one byte

Guarantees

- Verifier checked all instructions starting at 32-byte-multiple addresses
- Computed jumps can only go to 32-byte-multiple addresses
- What prevents the module from jumping past the AND, directly to the JMP?
  - The NaCl jump instruction will never be compiled so that the AND part and the JMP part are split by a 32-byte boundary. Thus, you could never jump straight to the JMP part

What about RET instructions?

- Effectively a computed jump, but with address stored on stack
- Race condition
  - If we check the address on the stack, TOCTOU with another thread
- Prohibited
- `pop` + `nacljmp` code

Segmentation

- We need to prevent jumps outside of the code
- x86 hardware provides "segments"
- Relative address within some segment
  - Segment specifies base+size
- Address translation:

(segment selector, addr) -> (segbase + addr % segsize)

Invoking trusted code from sandbox

- Trampoline undoes the sandbox, enters trusted code
  - Starts at a 32-byte multiple boundary
  - Loads unlimited segment
  - Jumps to trusted code that lives above 256MB
- Trampoline must fit in 32 bytes
- Trusted code first switches to a different stack
- Trusted code reloads other segment selectors

Service Runtime

- Memory allocation: sbrk/mmap
- Thread operations: create, etc
- IPC: initially with Javascript code on page that started this NaCl program
- Browser interface via NPAPI: DOM access, open URLs, user input, etc.
- No networking: can use Javascript to access network

Limiting code/data

- New segment with offset=0, size=256MB
- Set all segment selectors to that segment
- Modify verifier to reject any instructions that change segment selectors
- Ensures all code and data accesses will be within [0..256MB)

How secure is Native Client

- Inner sandbox: validator has to be correct
- Outer sandbox: OS-dependent plan
- Why the outer sandbox?
  - Possible bugs in the inner sandbox.
- What could an adversary do if they compromise the inner sandbox?
  - Exploit CPU bugs.
  - Exploit OS kernel bugs.
  - Exploit bugs in other processes communicating with the sandbox process
- Service runtime: initial loader, runtime trampoline interfaces.
- Inter-module communication (IMC) interface + NPAPI: complex code, can (and did) have bugs

What about buffer overflows?

- Any computed call (function pointer, return address) has to use 2-instr jump
- Only jump to validated code in the module's region
- Buffer overflows might allow attacker to take over module
- However, can't escape NaCl's sandbox

Overhead

- CPU overhead dominated by code alignment requirements
  - Larger instruction cache footprint
- Minimal overhead for added checks on computed jumps
- Call-into-service-runtime performance seems comparable to Linux syscalls
- Average overhead is less than 5%

Limitations

- Static code
  - No JIT
  - No shared libraries
- Dynamic code is possible to sandbox though!
  - Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code
- No standard API to communicate with the DOM
  - If any other browser wants to implement NaCl, they have to reverse-engineer Google's version to be compatible

You can use NaCl

```
<embed name="nacl_module" id="hello_world"
  width=0 height=0 src="hello_world.nmf"
    type="application/x-nacl" />
```

Source: https://developer.chrome.com/native-client/devguide/tutorial/tutorial-part1

Mozilla's approach: asm.js

- a strict subset of JavaScript
- can be used as a low-level, efficient target language for compilers
- abstraction similar to the C/C++ virtual machine:
  - a large binary heap with efficient loads and stores, integer and floating-point arithmetic, first-order function definitions, and function pointers
- C/C++ => LLVM => Emscripten => JavaScript

asm.js code example

```
function strlen(ptr) { // calculate length of C
string
    ptr = ptr | 0;
    var curr = 0;
    curr = ptr;
    while (MEM8[curr] | 0 != 0) {
        curr = (curr + 1) | 0;
    }
    return (curr - ptr) | 0;
}
```

The future: WebAssembly

- Google is moving away from NaCl/PNacl
  - One of the main reasons is the lack of standardized API
  - Everyone is familiar with JavaScript
- Binary encoding of the AST
  - No more parsing code
  - More compact
- Builds on top of asm.js
- Great browser support
- DOM manipulation done through JS API

WebAssembly browser support

WebAssembly - OTHER

Global 58.25%

WebAssembly or "wasm" is a new portable, size- and load-time-efficient format suitable for compilation to the web.

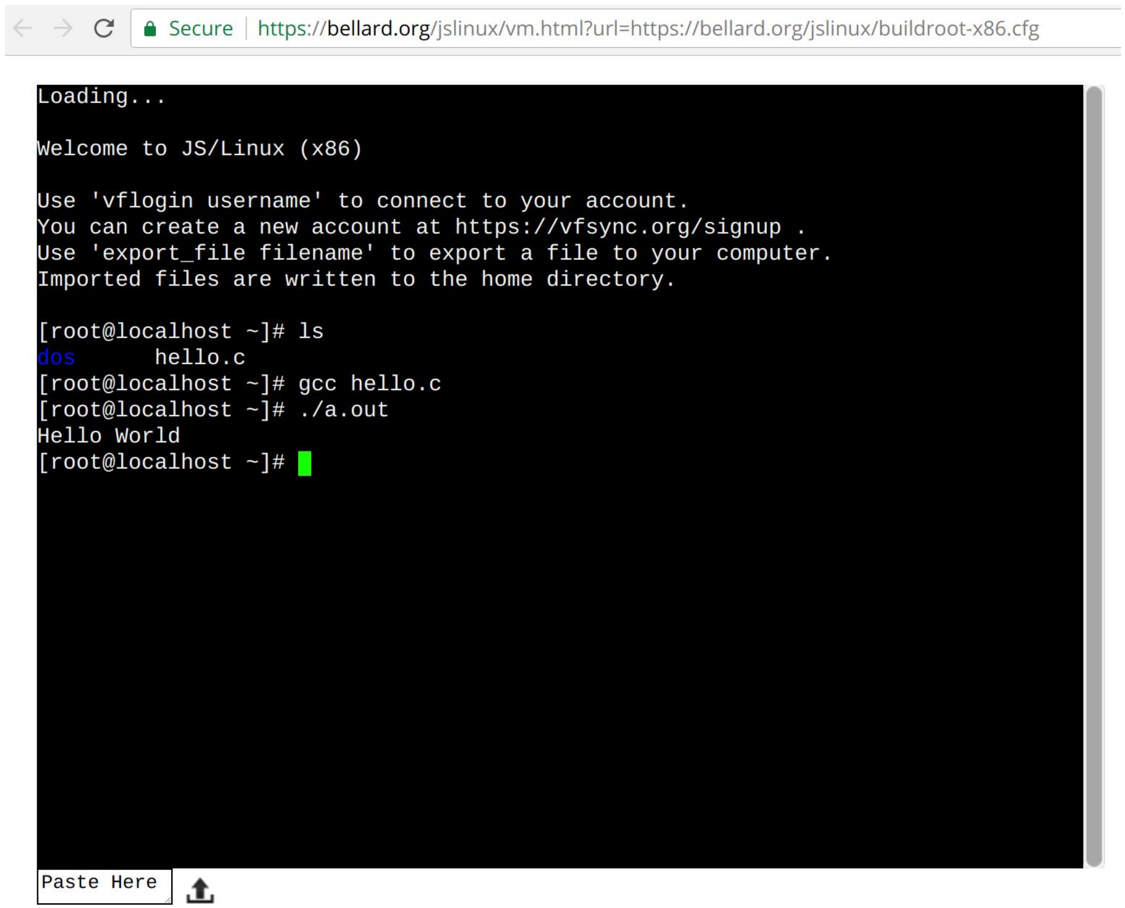
Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android
			49						
			59			10.2		4.4	
	14	54	60	10.1		10.3		4.4.4	
11	3 15	55	61	11	47	11	all	56	61
	16	56	62	TP	48				
		57	63		49				
		58	64						

source: http://caniuse.com/#feat=wasm

Your Security Zen

JS/Linux (x86)



source: https://bellard.org/jslinux/