Software Foundations of Security and Privacy (15-316, spring 2017)

Lecture 6: Sandboxing and Software Fault Isolation

Jean Yang

jyang2@andrew.cmu.edu

With material from Brad Karp

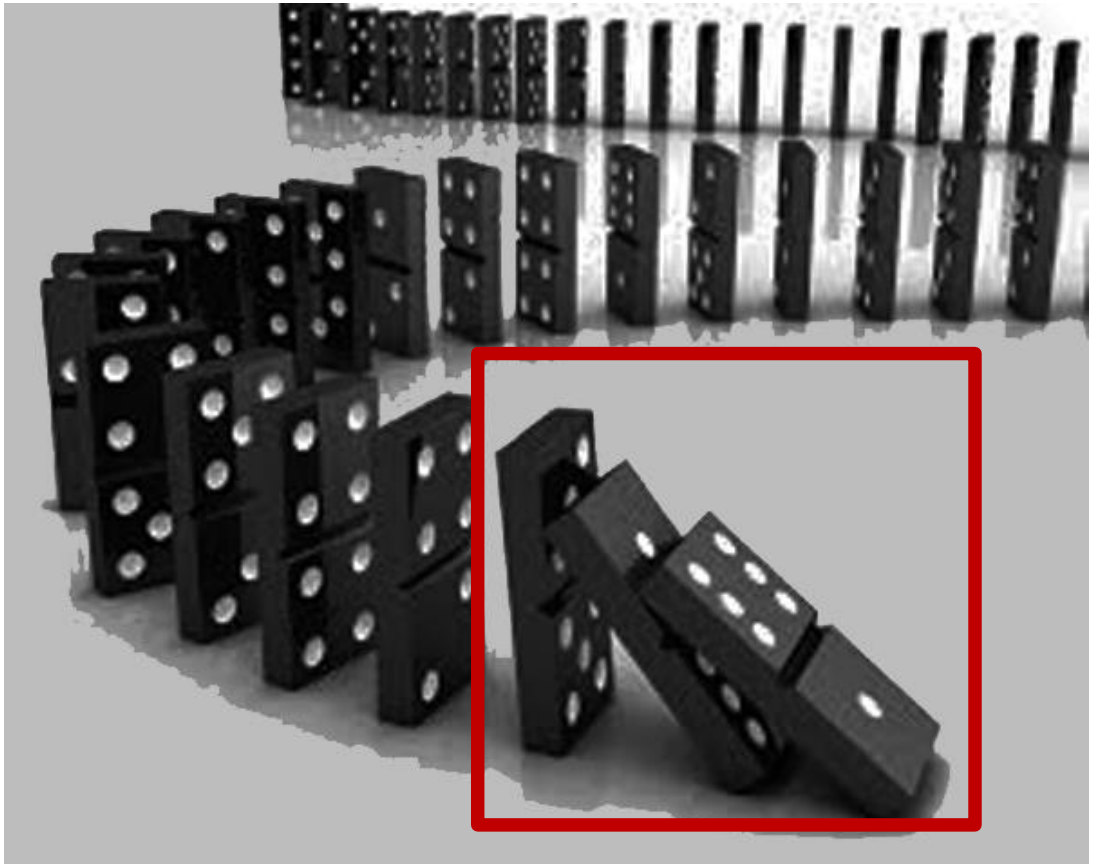
Northeast Blackout of 2003

- Affected estimated 10 million people in Ontario and 45 million people in 8 US states.
- Primary cause was software bug in alarm system at control room of FirstEnergy Co.
- Operators were not aware of need to redistribute power, triggering race condition on control software.

Cascading Failure on the Grid

"Unprocessed events queued up after the alarm system failure and the primary server failed within 30 minutes. Then all applications (including the stalled alarm system) were automatically transferred to the backup server, which itself failed at 14:54. The server failures slowed the screen refresh rate of the operators' computer consoles from 1–3 seconds to 59 seconds per screen. The lack of alarms led operators to dismiss a call from American Electric Power about the tripping and reclosure of a 345 kV shared line in northeast Ohio. But by 15:42, after the control room itself lost power, control room operators informed technical support (who were already troubleshooting the issue) of the alarm system problem."

- Technical Analysis of the August 14, 2003, Blackout



Part One: Sandboxing Software

Existential Threat: Untrusted Code

Material from Brad Karp

Goal: be able to run code from untrusted sources without invoking complete disaster.

- Sources of untrusted programs: browser plugins (*e.g.* Flash plugin); software packages (*e.g.* binary kernel module for Linux).
- Key risk: code from untrusted source will run in your application's address space.

Risks of Running Untrusted Code

Material from Brad Karp

- Overwrites trusted data or code
- Reads private data from trusted code's memory
- Executes privileged instructions
- Calls trusted functions with bad arguments
- Jumps to middle of trusted functions
- Contains vulnerabilities allowing others to do above

Running Lesson of the Course….

Using low-level languages like C is dangerous!

Ways to Mitigate Dangers of C

- Using memory-safe languages
- Bounds checking (reference monitor)
- Verifying programs
- Preventing bugs from spreading too far (sandboxing)

Today's lecture!

Allowed Operations for Untrusted Code

Material from Brad Karp

- Reads/writes own memory
- Executes own code
- Calls explicitly allowed functions in trusted code, at correct entry points

Solution: Sandboxing

- Each application runs in its own process, with its own system resources.
- There is a tight permissions framework that regulates which apps get access to data produced by other apps.
- Usually restricts behaviors such as network access, inspecting the host system, and reading from input devices.

Example: Android

Material from John Mitchell

- Isolation
  - Multi-user Linux operating system
  - Each application normally runs as different user
- Communication between applications
  - May share same Linux user ID
    - Access files from each other
    - May share same Linux process and Dalvik VM (Java platform)
  - Communicate through application framework

Android Application Sandbox

Material from John Mitchell

- Each application runs with its UID in its own Dalvik virtual machine
  - Provides CPU protection, memory protection
  - Authenticated communication protection using Unix domain sockets
  - Only ping, zygote (spawn another process) run as root
- Applications announce permission requirement
  - Create a whitelist model – user grants access (all questions asked at compile time)
  - Inter-component communication reference monitor checks permissions

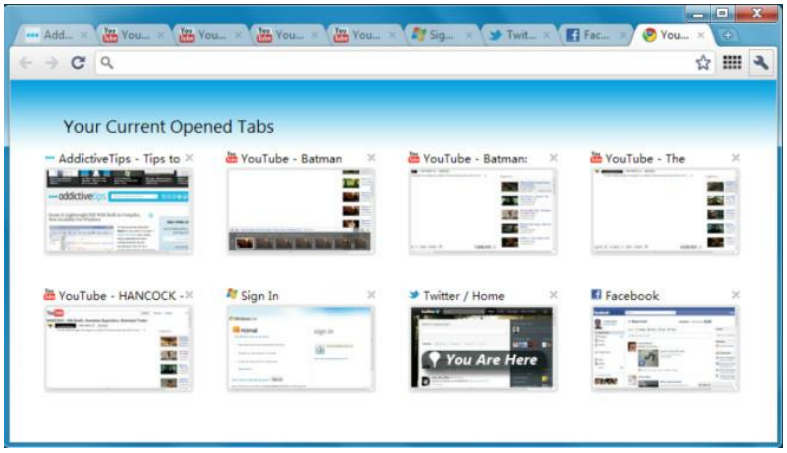
Example Android Permissions

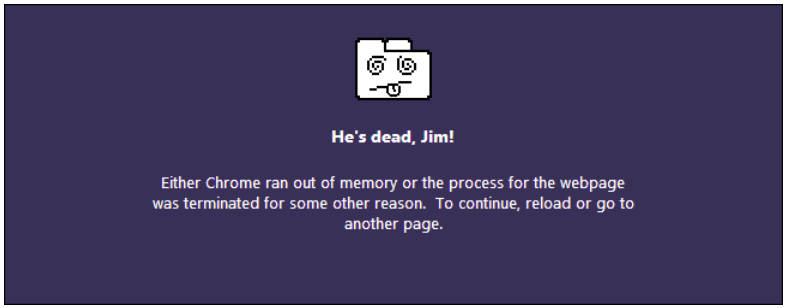
Material from John Mitchell

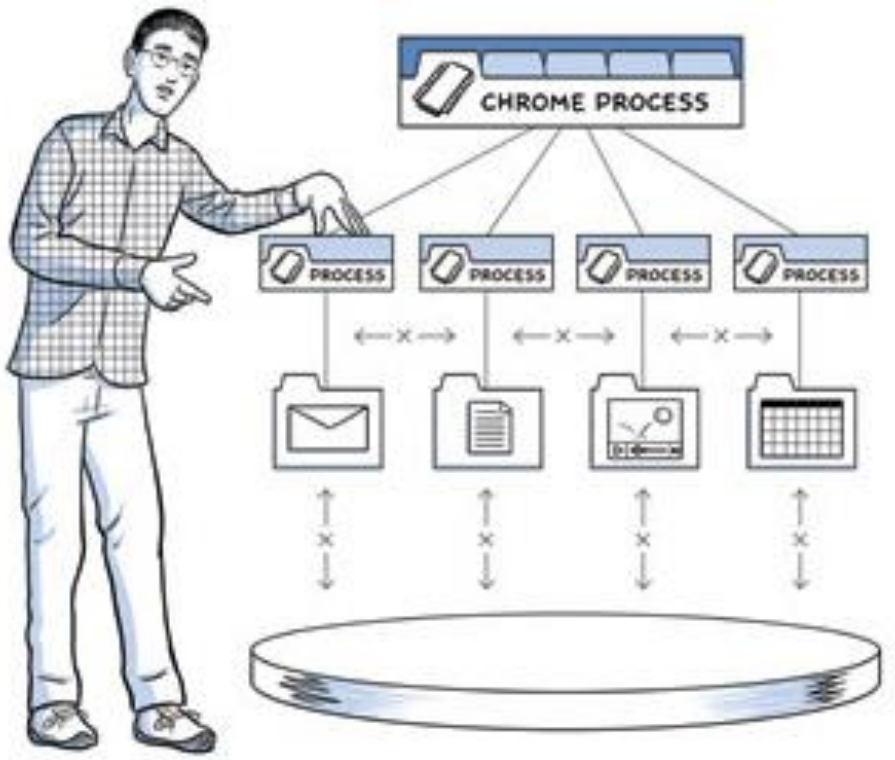
- “android.permission.INTERNET”
- “android.permission.READ_EXTERNAL_STORAGE”
- “android.permission.SEND_SMS”
- “android.permission.BLUETOOTH”

Bad things happen when applications get more permissions than they are supposed to!

Example: Chrome

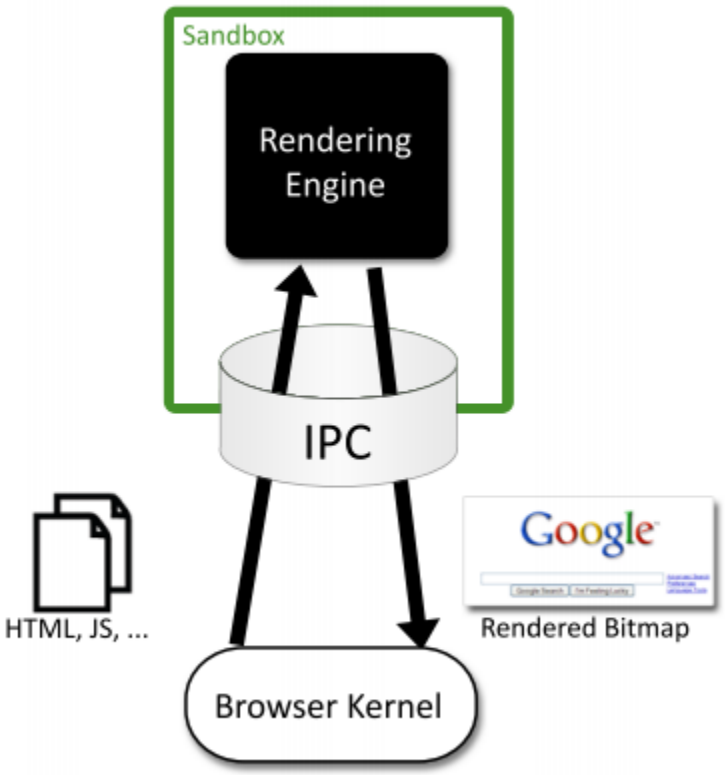






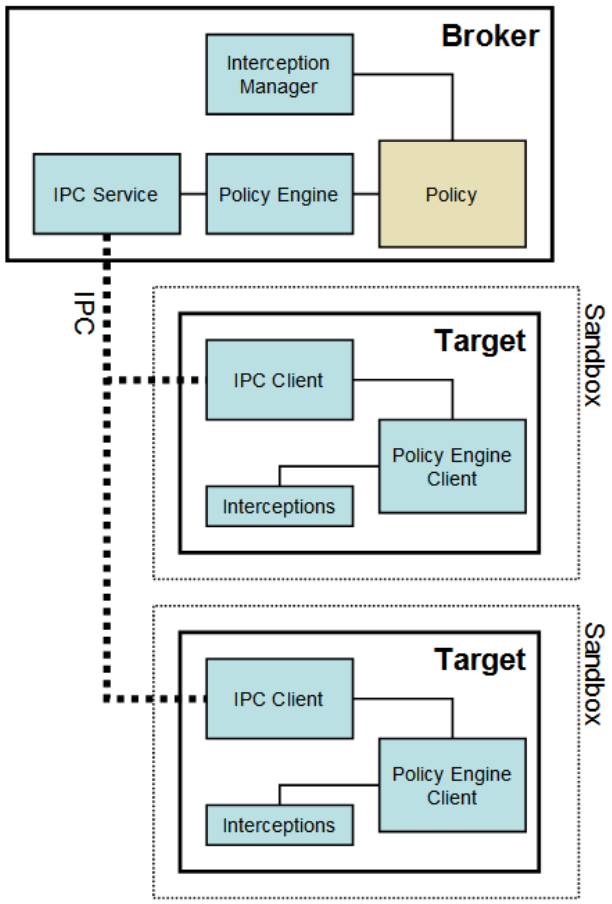
The Chrome browser is full of sandboxes!
Processes, plugins, and more…

Chromium Security Model



- Separate instance of rendering engine for each tab that displays content from the web.
- Each plug-in runs in separate host process, outside both rendering engines and browser kernel.

Chromium on Windows



- Uses OS-provided security to contain code execution.
- Principle of least privilege: sandbox works even if user cannot elevant to super-user.
- Limits severity of bugs: cannot install persistent malware; cannot read and steal arbitrary files.

Chromium on Windows (2)

Broker Process Responsibilities

1. Specify policy for each target process
2. Spawn target processes
3. Host sandbox policy engine service
4. Host sandbox interception manager
5. Host sandbox IPC service
6. Perform policy-allowed actions on behalf of target processes

Target Process Architecture

1. All code to be sandboxed
2. Sandbox IPC client
3. Sandbox policy engine client
4. Sandbox interceptions

Problems with Extensions and Separate Processes

Material from Brad Karp

- Not very transparent for programmer if application and extension are closely coupled
- Performance hit: need context switches between processes
  - Trap to kernel, copy arguments, save and restore registers, flush processor's TLB

Part Two: Running Untrusted Code in the Same Process

Solution: Software Fault Isolation

Material from Brad Karp

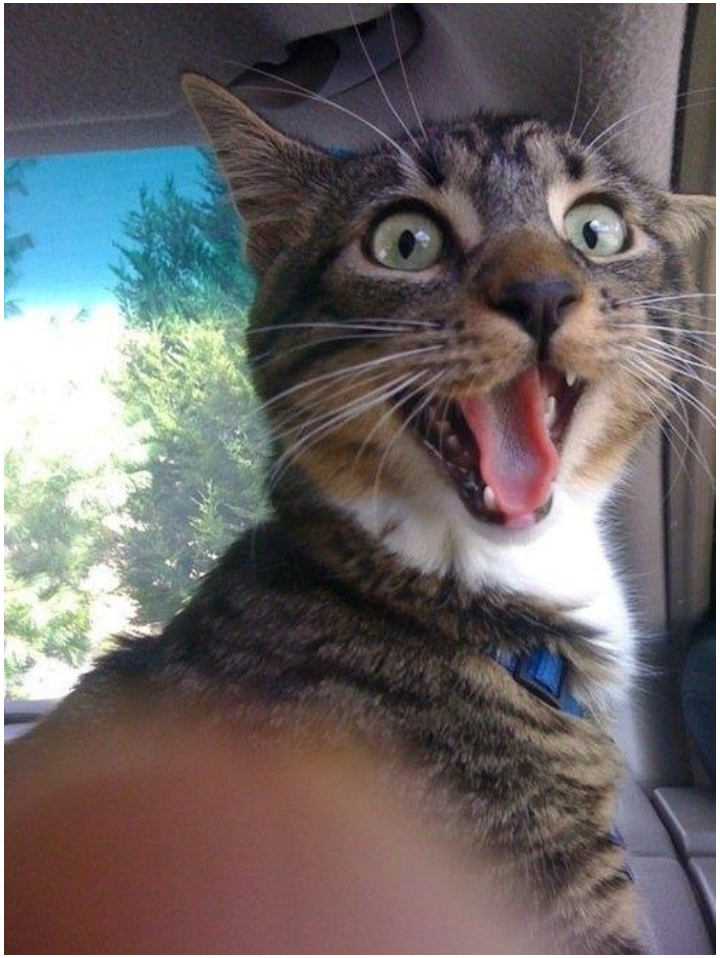
- Place extension's code and data in **sandbox**:
  - Prevent extension's code from writing to app's memory outside sandbox
  - Prevent extension's code from transferring control to app's code outside sandbox
- **Main idea:** add instructions between memory writes and jumps to inspect targets and constrain behavior

SFI Use Scenario

Material from Brad Karp

- Developer runs **sandboxer** on unsafe extension code to produce safe, sandboxed version:
  - Adds instructions that sandbox unsafe instructions
  - Transformation by compiler or binary rewriter
- Before running untrusted binary code, user runs **verifier**:
  - Checks that safe instructions don't access memory outside extension code's data
  - Checks that sandboxing instructions in place before all unsafe instructions

What Is Trusted?



Only the verifier!

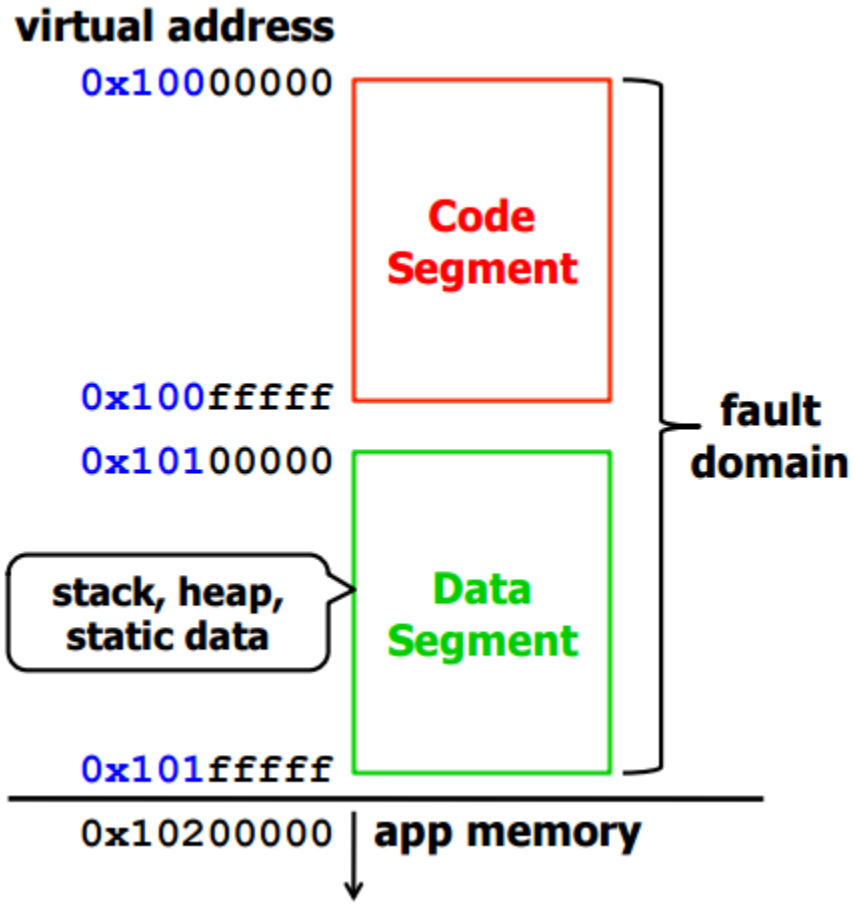
Unit of Isolation: Fault Domain

Material from Brad Karp

- SFI confines untrusted code within a **fault domain**, in same address space (process) as trusted code
- Fault domain consists of:
  - **Unique ID** (for access control on syscalls)
  - **Code segment:** virtual address range with same unique high-order bits, used to hold code
  - **Data segment:** virtual address range with same unique high-order bits, used to hold data

Fault Domain Example

Material from Brad Karp



- **Segment IDs** are 12 bits long
- Separate segments for code and data allow **distinguishing addresses** as falling in one or other

Sandboxing Memory

Material from Brad Karp

- Untrusted code should only be able to:
  - Jump within its fault domain's code segment
  - Write within its fault domain's data segment
- Sandboxer must ensure all jump, call, and memory store instructions comply.
- Two types of memory address in instructions:
  - **Direct:** complete address is specified statically in instruction
  - **Indirect:** address computed from register's value

Sandboxing Memory (2)

Material from Brad Karp

- For directly addressed memory instructions, sandboxer should only emit:
  - Directly addressed jumps and calls whose targets fall in fault domain's code segment
  - Directly addressed stores whose targets fall in fault domain's data segment
- Directly addressed jumps, calls, stores can be made safe **statically**

Indirectly Accessed Memory

Material from Brad Karp

- Indirectly addressed jumps, calls, stores harder to sandbox—full address depends on register **whose value is not known statically**
  - *e.g.* **`STORE R0, R1`**
  - *e.g.* **`JR R3`**
- These are **unsafe** instructions that must be made safe at runtime

Indirectly Accessed Memory (2)

Material from Brad Karp

Unsafe Instruction

```
STORE R0, R1
```

Sandboxer output

```
MOV Ra, R0  ; copy R0 into Ra
SHR Rb, Ra, Rc  ; Rb = Ra >> Rc
CMP Rb, Rd  ; Rd holds data
            ; segment ID
BNE fault ; wrong data segment ID
STORE Ra, R1 ; Do write
```

Ra, Rc, and Rd are **dedicated** registers that may not be used by extension code. Rd holds data segment ID.

Indirectly Accessed Memory (3)

Material from Brad Karp

- Why does the rewritten code use **`STORE Ra, R1`** and not **`STORE R0, R1`**? After all, **`R0`** has passed the check!
- But extension code may jump directly to **`STORE`**, bypassing check instructions!
- Because **`Ra`**, **`Rc`**, **`Rd`** are dedicated, **`Ra`** will always contain a safe address inside the data segment.

Indirectly Accessed Memory (4)

Material from Brad Karp

- Costs of first sandboxing scheme for indirectly addressed memory:
  - Adds 4 instructions before each indirect store
  - Uses 6 registers, 5 of which must be dedicated (never available to extensions)
- Can we do better, and get away with fewer added instructions?
- Yes, if we give up being able to identify which instruction access outside sandbox!

Faster Sandboxing of Indirect Addresses

Material from Brad Karp

- **Idea:** instead of checking whether target address is in segment, **force it to be in segment**
- Transform **`STORE R0, R1`** into:

```
AND Ra, R0, Re ; clear segment ID bits in Ra
OR Ra, Ra, Rf ; set segment ID to correct value
STORE Ra, R1  ; do write to safe target address
```

- Now segment ID bits in **`Ra`** will always be correct: can write anywhere in segment, but not outside it
- **Cost:** 2 added instructions; 5 dedicated registers

Faster Sandboxing of Indirect Jumps and Calls

Material from Brad Karp

- Transform JR R0 as follows:

```
AND Rg, R0, Re ; clear segment ID bits in Rg
OR Rg, Rg, Rh   ; set segment ID
JR Rg   ; do jump to safe target address
```

- Note use of separate dedicated registers Rg for code target address, Rh for code segment ID
- Return from function similar (to sandbox return address)

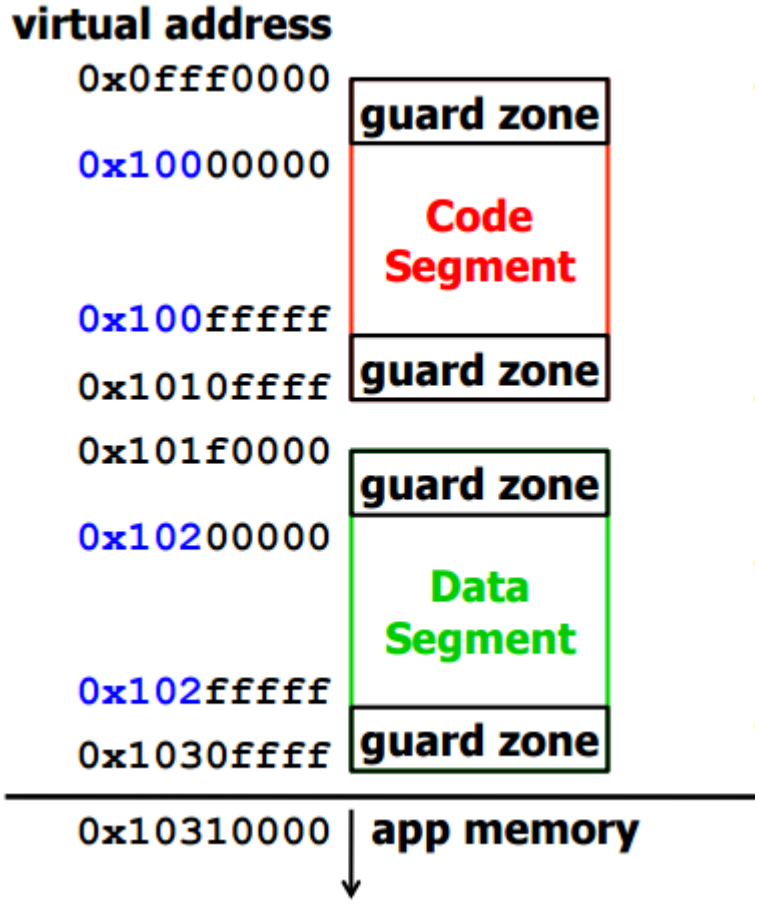
Optimization: Guard Zones

Material from Brad Karp

- Some instructions use “register+offset” addressing: use register as base and offset for CPU to add to it
- To sandbox such an instruction, SFI would need to do additional ADD to compute base+offset
- Insight: offsets are of limited size, so if base correct, offset could stray no more than 64K outside that segment

Guard Zones (2)

Material from Brad Karp



- Surround each segment with 64K **guard zone** of unmapped pages.
- Ignore offsets when sandboxing.
- Accesses to guard zones cause traps!

Optimization: Stack Pointer

Material from Brad Karp

- Insight: stack pointer is read far more often than it's written—used as base address for many reg+offset instructions
- SFI doesn't sandbox uses of stack pointer as base address; instead sandboxes setting of stack pointer, so stack pointer always contains safe value
- Reduces number of instructions that pay sandboxing overhead

Verifier

Material from Brad Karp

- For instructions that use **direct addressing**, easy to check **statically** that segment IDs in addresses are correct
- For those that use **indirect addressing**, verifier must ensure instruction preceded by full set of sandboxing instructions
- Ensures **no privileged instructions in code**
- Ensures **PC-relative branches fall in code segment**

SFI vs. Exploits

Material from Brad Karp

Exploit	SFI Protection
Stack-smashing (injecting code into stack buffer)	Can't execute own injected code—can't jump to data segment
Return-to-libc	Can overwrite return address with one within fault-domain's code segment—so can return-to-libc **within extension**
Format string vulnerabilities	""

But… SFI Limitations on x86

Material from Brad Karp

- MIPS instructions fixed-length while x86 instructions are variable-length
  - Result: can jump into middle of x86 instruction!
  - Consider the binary for AND eax, 0x00CD, which is 25 CD 80 00 00. If adversary jumps to second byte, they execute CD 8, which traps to a system call on Linux!
  - Jump to mid-instruction on x86 may even jump out of fault domain into app code! :o
- Also, x86 has very few registers, so cannot dedicate registers easily

SFI vs. Exploits: Lessons

Material from Brad Karp

- SFI allows write (including buffer overrun, %n overwrite) to extension's data
- SFI allows jumps anywhere in extension's code segment
- …so attacker can exploit extension's execution
- …and on x86, can probably cause jump out of fault domain

SFI was not designed to prevent exploits, but rather to isolate untrusted extension!

SFI in the Wild: NaCl

Material from Brad Karp

Native Client ("NaCl") is:

- A sandbox for untrusted x86 native code
- A browser-based framework designed to:
  - Allow browser-based applications to have the performance of native applications
  - Provide OS portability
  - Provide performance-oriented features such as threading, instruction set extensions, compiler intrinsics, and hand-coded assembly

SFI Summary

Material from Brad Karp

- Confines writes and control transfers in extension's data and code segments
- Can support direct calls to allowed functions in trusted (app) code
- Prevents execution of privileged instructions
- Any write or control transfer within extensions memory is allowed
- Requires dedicated registers

Summary: Motivations for SFI

- Initial motivation was performance: can achieve memory protection at low cost.
- Running programs without suitable protection in hardware.
- Programs that support plug-ins.
- Programs comprising many objects.

For Want of a Nail

For want of a nail the shoe was lost.
For want of a shoe the horse was lost.
For want of a horse the rider was lost.
For want of a rider the message was lost.
For want of a message the battle was lost.
For want of a battle the kingdom was lost.
And all for the want of a horseshoe nail.

Software Fault Isolation: When you lose a nail, you only lose your shoe.

Discussion Questions

- When might you want to use software fault isolation, as to other kinds of reference monitoring?
- What guarantees do we get from software fault isolation?
- What guarantees *don't* we get?
- How can SFI still be useful if we can sandbox faster and faster?