Sandboxing Untrusted Code: Software-Based Fault Isolation (SFI)

Brad Karp
UCL Computer Science

CS GZ03 / M030
11th December 2015
Motivation: Vulnerabilities in C

- Seen dangers of vulnerabilities:
  - injection of arbitrary code
  - return-to-libc (no code injection; malicious invocation of existing code)
- Vulnerabilities are bugs—application behavior not intended by programmer
- Bugs in C often because memory operations not safe
  - many ways to overwrite stored pointer, cause it to point to arbitrary memory
Motivation: Vulnerabilities in C

• Seen dangers of vulnerabilities:
  – injection of arbitrary code
  – return to libc (no code injection; malicious)

Can we constrain behavior of application code to prevent bugs from corrupting memory, and thus allowing exploits?

• Bugs in C often because memory operations not safe
  – many ways to overwrite stored pointer, cause it to point to arbitrary memory
Motivation: Untrusted Extensions

• Users often wish to extend application with new functionality made available as a binary module, e.g.,
  – Flash player plugin for Firefox browser
  – Binary kernel module for new filesystem for Linux

• Key risk: code from untrusted source (e.g., web site), but will run in your application’s address space
  – What if code overwrites your app’s data?
  – Or calls functions in your app’s code with ill intent? (e.g., calls disable_certificate_check())
Motivation: Untrusted Extensions

• Users often wish to extend application with new functionality made available as a binary module, e.g.,
  – Flash player plugin for Firefox browser

N.B. extension code may be malicious or may merely be buggy

web site), but will run in your application’s address space
  – What if code overwrites your app’s data?
  – Or calls functions in your app’s code with ill intent? (e.g., calls disable_certificate_check())
Risks of Running Untrusted Code

- Overwrites trusted data or code
- Reads private data from trusted code’s memory
- Executes privileged instruction
- Calls trusted functions with bad arguments
- Jumps to middle of trusted function
- Contains vulnerabilities allowing others to do above
Allowed Operations for Untrusted Code

- Reads/writes own memory
- Executes own code
- Calls explicitly allowed functions in trusted code at correct entry points
Straw Man Solution: Isolation with Processes

- Run original app code in one process, untrusted extension in another; communicate between them by RPC
  - (Recall NFS over RPC, but between distinct hosts)
- Memory protection means extension cannot read/write memory of original app
- Not very transparent for programmer, if app and extension closely coupled
- Performance hit: context switches between processes
  - trap to kernel, copy arguments, save and restore registers, flush processor’s TLB
Straw Man Solution: Isolation with Processes

- Run original app code in one process, untrusted extension in another; communicate between them by RPC
  - (Recall NFS over RPC, but between distinct hosts)
- Memory protection means extension cannot

Can we do better?

- Not very transparent for programmer, if app and extension closely coupled
- Performance hit: context switches between processes
  - trap to kernel, copy arguments, save and restore registers, flush processor’s TLB
Today’s Topic: Software-Based Fault Isolation

- Run untrusted binary extension in **same process (address space)** as trusted app code
- 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
- Idea: add instructions before memory writes and jumps to **inspect their targets and constrain their behavior**
SFI Use Scenario

- Developer runs **sandboxer** on unsafe extension code, to produce safe, sandboxed version:
  - adds instructions that sandbox unsafe instructions
  - transformation done by compiler or by binary rewriter

- Before running untrusted binary code, user runs **verifier** on it:
  - checks that safe instructions don’t access memory outside extension code’s data
  - checks that sandboxing instructions in place before all unsafe instructions
SFI Use Scenario

- Developer runs sandboxer on unsafe extension code, to produce safe, sandboxed version:
  - adds instructions that sandbox unsafe instructions

User need not trust sandboxer; only verifier

- Before running untrusted binary code, user runs verifier on it:
  - checks that safe instructions don’t access memory outside extension code’s data
  - checks that sandboxing instructions in place before all unsafe instructions
SFI Unit of Isolation: Fault Domain

- SFI confines untrusted code within a fault domain, in same address space (process) as trusted code
- Fault domain consists of:
  - Unique ID (used 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
- Segment ID: unique high-order bits for a segment
Fault Domain Example

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

<table>
<thead>
<tr>
<th>virtual address</th>
<th>Code Segment</th>
<th>Data Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10100000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10100000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **app memory**: 0x10200000
Sandboxing Memory

• 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 with above

• Two types of memory addresses in instructions:
  – direct: complete address is specified statically in instruction
  – indirect: address is computed from register’s value
Sandboxing Memory (2)

- For directly addressed memory instructions, sandboxer should only emit:
  - directly addressed jumps and calls whose targets fall in fault domain’s code segment
    - e.g., JUMP 0x10030000
  - directly addressed stores whose targets fall in fault domain’s data segment
    - e.g., STORE 0x10120000, R1

- Directly addressed jumps, calls, stores can be made safe statically
Sandboxing Indirectly Addressed Memory

• Indirectly addressed jumps, calls, stores harder to sandbox—full address depends on register whose value not known statically
  – e.g., STORE R0, R1
  – e.g., JR R3

• These are unsafe instructions that must be made safe at runtime
Sandboxing Indirectly Addressed Memory (2)

• Suppose unsafe instruction is
  \[\text{STORE R0, R1 } \quad \text{; write R1 to Mem[R0]}\]

• Sandboxer rewrites code to:
  \[
  \begin{align*}
  &\text{MOV Ra, R0 } \quad \text{; copy R0 into Ra} \\
  &\text{SHR Rb, Ra, Rc } ; \quad \text{Rb = Ra >> Rc, to get segment ID} \\
  &\text{CMP Rb, Rd } \quad ; \quad \text{Rd holds correct data segment ID} \\
  &\text{BNE fault } \quad ; \quad \text{wrong data segment ID} \\
  &\text{STORE Ra, R1 } \quad ; \quad \text{Ra in data segment, so do write}
  \end{align*}
  \]

• Ra, Rc, and Rd are \textit{dedicated}—may not be used by extension code
Sandboxing Indirectly Accessed Memory (3)

- Why does rewritten code use
  \texttt{STORE Ra, R1}
- and not
  \texttt{STORE R0, R1}
- After all, R0 has passed the check!
- Extension code may jump directly to \texttt{STORE}, \underline{bypassing check instructions}!
- Because Ra, Rc, Rd are dedicated, Ra will \underline{always contain safe address inside data segment}
Sandboxing Indirectly Accessed Memory (3)

- Why does rewritten code use 
  \texttt{STORE Ra, R1}
- and not 
  \texttt{STORE R0, R1}

\textbf{Remember: extension code may not set dedicated registers!}

- Extension code may jump directly to \texttt{STORE}, \textbf{bypassing check instructions}!
- Because Ra, Rc, Rd are dedicated, Ra will always contain safe address inside data segment
Sandboxing Indirectly Accesssed Memory (4)

- 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 extension)
    - example used 3 dedicated registers, but need 2 more for sandboxing unsafe code addresses
- Can we do better, and get away with fewer added instructions?
- Yes, if we give up being able to identify which instruction accessed outside sandbox!
Faster Sandboxing of Indirect Addresses

• Idea: don’t check if target address is in segment; **force it to be in segment**

• So we 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

• Very similar to data address sandboxing

• Transform \texttt{JR R0} as follows:
  \begin{verbatim}
  AND Rg, R0, Re ; clear segment ID bits in Rg
  OR Rg, Rg, Rh ; set segment ID to correct value
  JR Rg ; do jump to safe target address
  \end{verbatim}

• N.B. use of separate dedicated registers
  Rg for code target address, Rh for code segment ID

• Return from function similar, too (to sandbox return address)
Optimization: Guard Zones

- Some instructions use “register+offset” addressing: they use register as base, and supply offset for CPU to add to it.
- To sandbox such an instruction, SFI would need to do additional ADD to compute base+offset.
- Clever insight: offsets are of limited size, because of instruction encoding (+/- 64K on MIPS).
- So if base in correct segment, offset could stray no more than 64K outside that segment.
Guard Zones (2)

• Surround each segment with 64K guard zone of unmapped pages
• Ignore offsets when sandboxing!
• Accesses to guard zones cause traps
• Saves one ADD for reg+offset instrs
Optimization: Stack Pointer

- 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

• Upon receiving (supposedly) sandboxed binary, verifier must ensure all instructions safe

• 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
Verifier (2)

- Verifier must ensure no privileged instructions in code
- Verifier must ensure PC-relative branches fall in code segment
- If sandboxed code fails any of these checks, verifier rejects it
- Otherwise, code is correctly sandboxed
SFI Limitations on x86

• MIPS instructions fixed-length; x86 instructions variable-length
  – Result: can jump into middle of x86 instruction!
  – e.g., binary for AND eax, 0x80CD is 25 CD 80 00 00
  – If adversary jumps to second byte, he executes the instruction CD 80, which traps to a system call on Linux!
  – Jump to mid-instruction on x86 may even jump out of fault domain into app code!

• x86 has very few registers (4 general-purpose ones), so cannot dedicate registers easily
SFI vs. Exploits

• Simple stack-smashing, injecting code in 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 do return-to-libc within extension

• Format string vulnerabilities?
  – same story as above
SFI vs. Exploits: Lessons

- 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 vs. Exploits: Lessons

• SFI allows write (including buffer overrun, %n overwrite) to extension’s data

To be fair, SFI wasn’t designed for x86, and wasn’t designed to prevent exploits, but rather to isolate untrusted extension from main application.

• ...and on x86, can probably cause jump out of fault domain
SFI Summary

- Confines writes and control transfers in extension’s data and code segments, respectively.
- Can support direct calls to allowed functions in trusted (app) code.
- Prevents execution of privileged instructions.
- Any write or control transfer within extension’s memory is allowed.
- Requires dedicated registers.
CFI: Control-Flow Integrity

- Follow-on to SFI; works on x86
- Idea: examine control flow graph (CFG) of program, which includes all functions and all transfers of control between them (e.g., calls of named functions, returns from them)
- Doesn’t require dedicated registers like SFI
- Finds all instruction boundaries
- Adds instructions to enforce that all jumps, branches, calls, returns transfer control to valid target found in CFG
CFI (2)

- Prevents return to injected code by overwriting return address:
  - transition to return address of injected code not in CFG
- Prevents return-to-libc attack:
  - enforces that return instruction in function f() can only transfer control to next instruction in some function that calls f()
- Further reading (not examinable): Abadi et al., Control-Flow Integrity, CCS 2005