Control Flow Integrity

Lujo Bauer
18-732
Spring 2015
Control Hijacking Arms Race

Control Flow Hijacks

Attack
- Buffer Overflows
- Format String Vulnerabilities
  - More Buffer Overflows
  - Mem Read
  - Mem Write

Defense
- Computation
  - ret2libc
  - Return-Oriented Programming
- DEP/NX

Control Flow Integrity Attacks

http://propercourse.blogspot.com/2010/05/i-believe-in-duct-tape.html
CFI: Goal

Provably correct mechanisms that prevent powerful attackers from succeeding by protecting against all control flow integrity attacks
CFI: Idea

During program execution, whenever a machine-code instruction transfers control, it targets a valid destination, as determined by a Control Flow Graph (CFG) created ahead of time
Attack Model

Powerful Attacker: **Can at any time arbitrarily overwrite any data memory and (most) registers**
- Attacker cannot directly modify the PC
- Attacker cannot modify reserved registers

Assumptions:
- Data memory is Non-Executable
- Code memory is Non-Writable
Lecture Outline

- CFI: Goal
- Background: Control Flow Graph
- CFI: Approach
- Building on CFI
  - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study
Basic Block

A consecutive sequence of instructions / code such that
- the instruction in each position always executes before (dominates) all those in later positions, and
- no outside instruction can execute between two instructions in the sequence

control is “straight”
(no jump targets except at the beginning, no jumps except at the end)
CFG Definition

A static Control Flow Graph is a graph where
- each vertex $v_i$ is a basic block, and
- there is an edge $(v_i, v_j)$ if there may be a transfer of control from block $v_i$ to block $v_j$

Historically, the scope of a “CFG” is limited to a function or procedure, i.e., intra-procedural
Call Graph

- Nodes are functions
- There is an edge \((v_i, v_j)\) if function \(v_i\) calls function \(v_j\)

```c
void orange()
{
    1. red(1);
    2. red(2);
    3. green();
}

void red(int x)
{
    ..
}

void green()
{
    green();
    orange();
}
```
Superimpose CFGs of all procedures over the call graph

```c
void orange() {
    1. red(1);
    2. red(2);
    3. green();
}

void red(int x) {
    ..
}

void green() {
    green();
    orange();
}
```

A context sensitive super-graph for orange lines 1 and 2
Precision

The more precise the analysis, the more accurately it reflects the “real” program behavior

- Limited by soundness/completeness tradeoff
- Depends on forms of sensitivity of analysis
Soundness

If analysis says X is true, then X is true

Trivially sound: Say nothing

Sound and Complete: Say exactly the set of true things!

Completeness

If X is true, then analysis says X is true

Trivially complete: Say everything

True things

Things I say

True things
Context Sensitivity

Different calling contexts are distinguished

```c
void orange()  void red(int x)  void green()
{
    {  
        1. red(1); ..
        2. red(2);
        3. green();
        }  
    }  
green();
orange();
goal('Context sensitive distinguishes 2 different calls to red()');
```
Context Sensitive Example

\[
a = \text{id}(4);
\]
\[
b = \text{id}(5);
\]
\[
\text{void id(int } z) \{ \text{return } z; \}
\]

Context-Sensitive

Context-Insensitive (note merging)
Lecture Outline

- CFI: Goal
- Background: Control Flow Graph
- CFI: Approach
- Building on CFI
  - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study
CFI Overview

Invariant: Execution must follow a path in a control flow graph (CFG) created ahead of run time

Method:
- Build CFG statically, e.g., at compile time
- Instrument (rewrite) binary, e.g., at install time
  - Add IDs and ID checks; maintain ID uniqueness
- Verify CFI instrumentation at load time
  - Direct jump targets, presence of IDs and ID checks, ID uniqueness
- Perform ID checks at run time
  - Indirect jumps have matching IDs

Security Principle: Minimal Trusted Computing Base — Trust simple verifier, not complex rewriter
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}
Instrument Binary

- Insert a unique number at each destination
- Two destinations are equivalent if CFG contains edges to each from the same source

```c
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort( a, len, lt );
    sort( b, len, gt );
}
```

call 17, R: transfer control to R only when R has label 17
## Example of Instrumentation

### Original code

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]</td>
</tr>
<tr>
<td></td>
<td>; computed jump</td>
<td></td>
<td>; dst</td>
</tr>
</tbody>
</table>

### Instrumented code

- **Original Code:**
  - `B8 77 56 34 12 mov eax, 12345678h
  - `40 inc eax`
  - `39 41 04 cmp [ecx+4], eax`
  - `75 13 jne error_label`
  - `FF E1 jmp ecx`

- **Instrumented Code:**
  - `3E 0F 18 05 prefetchnta`
  - `78 56 34 12 mov eax, [esp+4]`
  - `8B 44 24 04 mov eax, [esp+4]`

- **Jump to the destination only if the tag is equal to “12345678”**
- **Abuse an x86 assembly instruction to insert “12345678” tag into the binary**
Verify CFI Instrumentation

- Direct jump targets (e.g., call 0x12345678)
  - Are all targets valid according to CFG?
- IDs
  - Is there an ID right after every entry point?
  - Does any ID appear in the binary by accident?
- ID checks
  - Is there a check before every control transfer?
  - Does each check respect the CFG?

Trust simple verifier, not complex rewriter
Revisiting Assumptions

- **UNQ: Unique IDs**
  - Required to preserve CFG semantics

- **NWC: Non-Writable Code**
  - Otherwise attacker can overwrite CFI dynamic check
  - Not true if code dynamically loaded or generated

- **NXD: Non-Executable Data**
  - Otherwise attacker could cause the execution of data labeled with expected ID
Security Guarantees

- Effective against attacks based on illegitimate control-flow transfer
  - Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge
- Does not protect against attacks that do not violate the program’s original CFG
  - Data-only attacks
  - Incorrect arguments to system calls
  - Substitution of file names
  - Incorrect logic in implementation
Evaluation

Fig. 6. Execution overhead of inlined CFI enforcement on SPEC2000 benchmarks.

x86 Pentium 4, 1.8 GHz, 512MB RAM; average overhead: 16%; range: 0-45%
Evaluation

- CFG construction + CFI instrumentation: ~10s
- Increase in binary size: ~8%
- Relative execution overhead:
  - crafty: CFI – 45%
  - gcc: CFI < 10%

- Security-related experiments
  - CFI protects against various specific attacks
    (read Section 4.3)
Lecture Outline

- CFI: Goal
- Background: Control Flow Graph
- CFI: Approach
- Building on CFI
  - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study
CFI implies *non-circumventable sandboxing* (i.e., safety checks inserted by instrumentation before instruction X will always be executed before reaching X)

**SFI**: Dynamic checks to ensure that target memory accesses lie within a certain range

– CFI makes these checks *non-circumventable*
SMAC: Generalized SFI

- **SMAC**: Different access checks at different instructions in the program
  - Isolated data memory regions that are only accessible by specific pieces of program code (e.g., library function)
  - SMAC can remove NX data and NW code assumptions of CFI
  - CFI makes these checks non-circumventable
Example: CFI + SMAC

```
call eax ; call a function pointer (destination address)

with CFI, and SMAC discharging the NXD requirement, can become:

and eax, 40FFFFFFFh ; mask to ensure address is in code memory
cmp [eax+4], 12345678h ; compare opcodes at destination
jne error_label ; if not ID value, then fail
call eax ; call function pointer
prefetchnta [AABBCCDDh] ; label ID, used upon the return
```

- Non-executable data assumption no longer needed since SMAC ensures target address is pointing to code
CFI as a Foundation for Non-circumventable IRMs

- Inlined Reference Monitors (IRM) work correctly assuming:
  - Inserted dynamic checks cannot be circumvented by changing control flow – enforced using CFI
  - IRM state cannot be modified by attacker – enforced by SMAC
CFI with Context Sensitivity

- Function F is called first from A, then from B; what’s a valid destination for its return?
  - CFI will use the same tag for both call sites, but this allows F to return to B after being called from A
  - Solution 1: duplicate code (or even inline everything)
  - Solution 2: use a shadow call stack
    - place stack in SMAC-protected memory region
    - only SMAC instrumentation code at call and return sites modify stack by pushing and popping values
    - Statically verify that instrumentation code is correct
Lecture Outline

- CFI: Goal
- Background: Control Flow Graph
- CFI: Approach
- Building on CFI
  - IRM, SFI, SMAC, Protected Shadow Call Stack
- Formal Study
Security Proof Outline

1. Define machine code semantics
2. Model a powerful attacker
3. Define instrumentation algorithm
4. Prove security theorem

Weakness of Abadi et al. work:
Formal study uses a simple RISC-style assembly language, not the x86 ISA
(cf. McCamant and Morrisett’s PittSFIeld 2006)
Machine Model

Execution State:

- $M_c$ (code memory): maps addresses to words
- $M_d$ (data memory): maps addresses to words
- $R$ (registers): maps register nos. to words
- $pc$ (program counter): a word
Operational Semantics

For each instruction, operational semantics defines how the instruction affects state
Operational Semantics (normal)

- Semantics of \textit{add rd, rs, rt}

\[
(M_c|M_d, R, pc) \rightarrow_n (M_c|M_d, R\{r_d \mapsto R(r_s) + R(r_t)\}, pc + 1)
\]

when \(M_c(pc)\) holds \textit{add rd, rs, rt} and \(pc + 1\) is in the domain of \(M_c\)

\(\rightarrow_n\) : Binary relation on states that expresses normal execution steps
Operational Semantics (attacker)

- Idea: Attacker may arbitrarily modify data memory and most registers at any time

- Formally, attacker transition captured by binary relation on states

\[ (M_c|M_d, R, pc) \rightarrow_a (M_c|M_d', R, pc) \]

Transitions → are either normal transitions →_n or attacker transitions →_a
Instrumentation Algorithm

- **I(Mc):** Code memory Mc is well-instrumented according to the CFI-criteria

- **Example:**
  - Every computed jump instruction is preceded by a particular sequence of instructions, which depends on a given CFG

Definition of CFG and instrumentation algorithm in paper
CFI Security Theorem

Let $S_0$ be a state with code memory $M_c$ such that $I(M_c)$ and $pc = 0$, and let $S_1, \ldots, S_n$ be states such that $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$. Then, for all $i \in 0..(n-1)$, either $S_i \rightarrow_a S_{i+1}$ or the $pc$ at $S_{i+1}$ is one of the allowed successors for the $pc$ at $S_i$ according to the given CFG.

- Requires definition of transition relation $\rightarrow$, instrumentation algorithm $I(M_c)$, and CFG.
- Property holds in the presence of attacker steps
- Proof is by induction on execution sequences
CFI Summary

Small Trusted Computing Base:
Trust simple verifier, not complex rewriter

Method:

- **Build CFG statically, e.g., at compile time**
- **Instrument (rewrite) binary, e.g., at install time**
  - Add IDs and ID checks; maintain ID uniqueness
- **Verify CFI instrumentation at load time**
  - Direct jump targets, presence of IDs and ID checks, ID uniqueness
- **Perform ID checks at run time**
  - Indirect jumps have matching IDs
Connections to Other Lectures

- Software analysis methods assume CFG accurately reflects possible executions of program
  - Software model checking (ASPIER, MOPS)
  - Static analysis (Coverity Prevent)

- Language-based methods
  - Type systems guarantee memory and control flow safety for programs written in that language (PCC, TAL)
  - No guarantees if data memory corrupted by another entity or flaw

- Run-time enforcement methods can be circumvented if CFG not respected
  - Software-based Fault Isolation (SFI)
  - Inlined Reference Monitors (IRMs)
Sources

- Some slides from J. Ligatti, D. Brumley, A. Datta.