Software Control Flow Integrity
Techniques, Proofs, & Security Applications

Jay Ligatti summer 2004 intern work with: Úlfar Erlingsson and Martín Abadi
Motivation I: Bad things happen

- DoS
- Weak authentication
- Insecure defaults
- Trojan horse
- Back door

- Particularly common: buffer overflows and machine-code injection attacks

Source: http://www.us-cert.gov
Motivation II: Lots of bad things happen

Source: http://www.cert.org/stats/cert_stats.html

(only Q1 and Q2 of 2004)
Motivation III: “Bad Thing” is usually UCIT

• About 60% of CERT/CC advisories deal with Unauthorized Control Information Tampering [XKl03]

E.g.: Overflow buffer to overwrite return address

• Other bugs can also divert control

- Attack Code
- Hijacked PC pointer
- Can be anything
- Garbage
Motivation IV: Previous Work

Ambitious goals, Informal reasoning, Flawed results

StackGuard of Cowan et al. [CPM+98] (used in SP2)

“Programs compiled with StackGuard are safe from buffer overflow attack, regardless of the software engineering quality of the program.” [CPM+98]

Why can’t an attacker learn/guess the canary?

What about function args?
Goal:

Provably correct mechanisms that prevent powerful attackers from succeeding by protecting against all UCIT attacks

Part of new project: \textit{Gleipnir}

...in Norse mythology, is a magic chord used to bind the monstrous wolf Fenrir, thinner than a silken ribbon yet stronger than the strongest chains of steel. These chains were crafted for the Norse gods by the dwarves from "the sound of a cat’s footfall and the woman’s beard and the mountain’s roots and the bear’s sinews and the fish’s breath and bird’s spittle."
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 our reserved registers (in the handful of places where we need them)

**Few Assumptions:**
- Data memory is Non-Executable *
- Code memory is Non-Writable *
- Also... currently limited to whole-program guarantees (still figuring out how to do dynamic loading of DLLs)
Our Mechanism

NB: Need to ensure bit patterns for nops appear nowhere else in code memory

CFG excerpt

\[ A_{\text{call}} \rightarrow B_1 \]

\[ A_{\text{call}+1} \rightarrow B_{\text{ret}} \]
More Complex CFGs

Maybe statically all we know is that \( F_A \) can call any int → int function

 CFG excerpt

\[
\text{succ}(A\_\text{call}) = \{B_1, C_1\}
\]

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

\[
\begin{align*}
F_A & \xrightarrow{\text{call fp}} F_B \quad \text{nop IMM}_1 \\
F_B & \xrightarrow{\text{if(*fp != nop IMM)}_1 \text{ halt}} F_C \quad \text{nop IMM}_1 \\
F_C & \rightarrow B_1 \\
F_C & \rightarrow C_1
\end{align*}
\]
Imprecise Return Information

Q: What if \( F_B \) can return to many functions?
A: Imprecise CFG

\[ \text{nop IMM}_2 \]

\[ \text{call } F_B \]

\[ \text{call } F_B \]

\[ \text{nop IMM}_2 \]

\[ \text{nop IMM}_2 \]

\[ \text{if}(**\text{esp} != \text{nop IMM}_2) \text{ halt} \]

\[ \text{return} \]

\[ \text{CFG excerpt} \]

\[ \text{succ}(B_{\text{ret}}) = \{ A_{\text{call+1}}, D_{\text{call+1}} \} \]

\[ A_{\text{call+1}} \]

\[ D_{\text{call+1}} \]

\[ B_{\text{ret}} \]

\[ \text{CFG Integrity:} \]

Changes to the PC are only to valid successor PCs, per succ().
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

\[ \text{A}_{\text{call}} \rightarrow \text{B}_1 \]
\[ \text{E}_{\text{call}} \rightarrow \text{C}_1 \]

CFG excerpt

\[ \text{A}_{\text{call}} \rightarrow \text{B}_1 \]
\[ \text{C}_{1A} \]
\[ \text{E}_{\text{call}} \rightarrow \text{C}_{1E} \]
Security Proof Outline

• Define machine code semantics
• Model a powerful attacker
• Define instrumentation algorithm
• Prove security theorem
Security Proof I: Semantics

“Normal” steps:
(an extension of [HST+02])

<table>
<thead>
<tr>
<th>Step Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nop</strong></td>
<td>( (M_c</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>( (M_c</td>
</tr>
<tr>
<td><strong>addi</strong></td>
<td>( (M_c</td>
</tr>
<tr>
<td><strong>movi</strong></td>
<td>( (M_c</td>
</tr>
<tr>
<td><strong>bgt</strong></td>
<td>( (M_c</td>
</tr>
</tbody>
</table>

\[
D_c(M_c(pc)) = \text{jmp} \ r_s \quad R(r_s) \in \text{dom}(M_c)
\]

\[
(M_c | M_d, R, pc) \rightarrow_n (M_c | M_d, R, R(r_s))
\]

**Attack step:**

\[
(M_c | M_d, R_0 - 2 | R_{3 - g1}, pc) \rightarrow_a (M_c | M_d', R_0 - 2 | R_{3 - g1}', pc)
\]

**General steps:**

\[
S \rightarrow_n S' \quad S' \rightarrow S''
\]

\[
S \rightarrow_a S' \quad S \rightarrow S''
\]
Security Proof II: Instrumentation Algorithm

(1) Insert new *illegal* instruction at the end of code memory

(2) For all computed jump destinations \( d \) with destination id \( X \), insert “nop X” before \( d \)

(3) Change every jmp \( r_s \) into:

\[
\begin{align*}
\text{addi} & \quad r_0, & r_s, & 0 \\
\text{id} & \quad r_1, & r_0[0] \\
\text{movi} & \quad r_2, & \text{IMM}_X \\
\text{bgt} & \quad r_1, & r_2, & \text{HALT} \\
\text{bgt} & \quad r_2, & r_1, & \text{HALT} \\
\text{jmp} & \quad r_0
\end{align*}
\]

Where \( \text{IMM}_X \) is the bit pattern that decodes into “nop X” s.t. \( X \) is the destination id of all targets of the jmp \( r_s \) instruction.
Security Proof III: Properties

- Instrumentation algorithm immediately leads to constraints on code memory, e.g.:

\[
[I\text{-Jmp}] \quad \forall M_c \quad \forall a \in \text{dom}(M_c) \quad \forall r_s : \\
Dc(M_c(a)) = jmp r_s \Rightarrow \\
\exists r'_s : Dc(M_c(a - 5)) = addi r_0, r'_s, 0 \quad \land \\
Dc(M_c(a - 4)) = ld r_1, r_0(0) \quad \land \\
\exists w_1 \quad \exists w_2 \quad \forall a' \in \text{dom}(M_c) : \\
Dc(M_c(a - 3)) = movi r_2, w_1 \quad \land \\
Dc(w_1) = nop w_2 \quad \land \\
Dc(M_c(a')) = nop w_2 \Rightarrow a' \in \text{succ}(M_c, a) \quad \land \\
\exists w_3 : \\
Dc(M_c(a - 2)) = bgt r_1, r_2, w_3 \quad \land \\
Dc(M_c(a - 1)) = bgt r_2, r_1, w_3 \quad \land \\
Dc(M_c(w_3)) = \text{illegal} \quad \land \\
r_s = r_0
\]

- Using such constraints + the semantics,

**Theorem 6**
\[
\forall n \geq 0 \quad \forall S_0\ldots S_n \quad \forall i \in \{0\ldots(n-1)\} : \\
\begin{cases}
I(S_0.M_c) \land \\
S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n \\
\Rightarrow \\
(S_i \rightarrow_a S_{i+1} \land S_{i+1}.pc = S_i.pc) \lor \\
(S_i \rightarrow_n S_{i+1} \land S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc))
\end{cases}
\]
SMAC Extensions

• In general, our CFG integrity property implies *uncircumventable sandboxing* (i.e., safety checks inserted by instrumentation before instruction X will always be executed before reaching X).

• Can remove NX data and NW code assumptions from language (can do SFI and more!):

  **NX data**
  
  addi r₀, rₛ, 0  
  bgt r₀, max(dom(M₃)), HALT  
  bgt min(dom(M₃)), r₀, HALT  
  [checks from orig. algorithm]  
  jmp r₀

  **NW code**
  
  addi r₀, rₜ, 0  
  bgt r₀, max(dom(M₄)) - w, HALT  
  bgt min(dom(M₄)) - w, r₀, HALT  
  st r₀(w), rₛ
Runtime Precision Increase

- Can use SMAC to increase precision
- Set up protected memory for dynamic information and query it before jumps
- E.g., returns from functions
  - When A calls B, B should return to A not D
  - Maintain return-address stack untouchable by original program
Efficient Implementation?

• Should be fast (make good use of caches):
  + Checks & IDs same locality as code
    – Static pressure on unified caches and top-level iCache
    – Dynamic pressure on top-level dTLB and dCache

• How to do checks on x86
  ▪ Can implement NOPs using x86 prefetching etc.
  ▪ Alternatively add 32-bit id and SKIP over it

• How to get CFG and how to instrument?
  ▪ Use magic of MSR Vulcan and PDB files
Microbenchmarks

- Program calls pointer to “null function” repeatedly
- Preliminary x86 instrumentation sequences

<table>
<thead>
<tr>
<th></th>
<th>PIII</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOP IMM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>11%</td>
<td>55%</td>
</tr>
<tr>
<td>Return</td>
<td>11%</td>
<td>54%</td>
</tr>
<tr>
<td>Both</td>
<td>33%</td>
<td>111%</td>
</tr>
<tr>
<td><strong>SKIP IMM</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>11%</td>
<td>19%</td>
</tr>
<tr>
<td>Return</td>
<td>221%</td>
<td>181%</td>
</tr>
<tr>
<td>Both</td>
<td>276%</td>
<td>195%</td>
</tr>
</tbody>
</table>

**PIII** = XP SP2, Safe Mode w/CMD, Mobile Pentium III, 1.2GHz

**P4** = XP SP2, Safe Mode w/CMD, Pentium 4, no HT, 2.4GHz
Future Work

• Practical issues:
  ▪ Real-world implementation & testing
  ▪ Dynamically loaded code
  ▪ Partial instrumentation

• Formal work:
  ▪ Finish proof of security for extended instrumentation
  ▪ Proofs of transparency (semantic equivalence) of instrumented code
  ▪ Move to proof for x86 code


End