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
Motivation III:
“Bad Thing” is usually UCIT

- About 60% of CERT/CC advisories deal with Unauthorized Control Information Tampering [XKI03]

- E.g.: Overflow buffer to overwrite return address

- Other bugs can also divert control

[Diagram of 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?

Figure 2: Canary Word Next to Return Address
This Research

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

Part of new project: 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

\[
\begin{align*}
F_A & \quad \Rightarrow \quad \text{if}(*fp \neq \text{nop IMM}_1) \text{ halt}
\text{call fp} \\
\text{nop IMM}_2 & \quad \Rightarrow
\end{align*}
\]

\[
\begin{align*}
F_B & \quad \Rightarrow \\
\text{nop IMM}_1 & \quad \Rightarrow \\
\text{if(**esp \neq \text{nop IMM}_2) halt} & \quad \Rightarrow \quad \text{return}
\end{align*}
\]

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

Maybe statically all we know is that $F_A$ can call any int → int function

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

$\text{CFG excerpt}$

$A_{\text{call}} \rightarrow B_1$

$succ(A_{\text{call}}) = \{B_1, C_1\}$
Imprecise Return Information

Q: What if $F_B$ can return to many functions?

A: Imprecise CFG

CFG excerpt

$A_{\text{call}+1} \rightarrow B_{\text{ret}}$

$D_{\text{call}+1} \rightarrow B_{\text{ret}}$

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

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

\[ A_{\text{call}} \rightarrow B_1 \]
\[ E_{\text{call}} \rightarrow C_1 \]

CFG excerpt

\[ A_{\text{call}} \rightarrow B_1 \]
\[ E_{\text{call}} \rightarrow C_{1A} \]
\[ E_{\text{call}} \rightarrow C_{1E} \]
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])

| If $Dc(M_c(pc)) = \text{then } (M_c | M_d, R, pc \rightarrow_n$ |
| --- |
| $\text{nop } w$ | $(M_c | M_d, R, pc + 1)$, when $pc + 1 \in \text{dom}(M_c)$ |
| $\text{add } r_d, r_s, r_t$ | $(M_c | M_d, R\{r_d \leftarrow R(r_s) + R(r_t)\}, pc + 1)$, when $pc + 1 \in \text{dom}(M_c)$ |
| $\text{addi } r_d, r_s, w$ | $(M_c | M_d, R\{r_d \leftarrow R(r_s) + w\}, pc + 1)$, when $pc + 1 \in \text{dom}(M_c)$ |
| $\text{movi } r_d, w$ | $(M_c | M_d, R\{r_d \leftarrow w\}, pc + 1)$, when $pc + 1 \in \text{dom}(M_c)$ |
| $\text{bgt } r_s, r_t, w$ | $(M_c | M_d, R, w)$, when $R(r_s) > R(r_t) \wedge w \in \text{dom}(M_c)$ |

| $(M_c | M_d, R, pc + 1)$, |

$Dc(M_c(pc)) = \text{jmp } r_s \quad \text{if } R(r_s) \in \text{dom}(M_c)$

$(M_c | M_d, R, pc \rightarrow_n (M_c | M_d, R, R(r_s))$

$\text{st } r_d(w), r_s$ | $(M_c | M_d\{R(r_d) + w \leftarrow R(r_s)\}, R, pc + 1)$, when $R(r_d) + w \in \text{dom}(M_d) \wedge pc + 1 \in \text{dom}(M_c)$ |

| Attack step: |

| General steps: |

$S \rightarrow_n S'$

$S \rightarrow S'$

$S \rightarrow S'$
Security Proof II: Instrumentation Algorithm

(1) Insert new \textit{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, \quad r_s, \quad 0 \\
\text{id} & \quad r_1, \quad r_0[0] \\
\text{movi} & \quad r_2, \quad \text{IMM}_X \\
\text{bgt} & \quad r_1, \quad r_2, \quad \text{HALT} \\
\text{bgt} & \quad r_2, \quad r_1, \quad \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.:

\[ Dc(M_c(a)) = jmp r_s \Rightarrow \]

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

- Using such constraints + the semantics,

**Theorem 6**
\[
\forall n \geq 0 \forall S_0..S_n \ \forall i \in \{0..(n-1)\} : \left( I(S_0, M_c) \land \\
S_0 \rightarrow S_1 \rightarrow ... \rightarrow S_n \Rightarrow \\
(S_i \rightarrow_a S_{i+1} \land S_{i+1}.pc = S_i.pc) \lor \\
(S_i \rightarrow_n S_{i+1} \land S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc)) \right)
\]
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 \( r_0, r_s, 0 \)
  - bgt \( r_0, \max(\text{dom}(M_C)), \text{HALT} \)
  - bgt \( \min(\text{dom}(M_C)), r_0, \text{HALT} \)
  - *[checks from orig. algorithm]*
  - jmp \( r_0 \)

  **NW code**
  - addi \( r_0, r_d, 0 \)
  - bgt \( r_0, \max(\text{dom}(M_D)) - w, \text{HALT} \)
  - bgt \( \min(\text{dom}(M_D)) - w, r_0, \text{HALT} \)
  - st \( r_0(w), r_s \)
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>Normalized Overheads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PIII</td>
</tr>
<tr>
<td>NOP IMM</td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>11%</td>
</tr>
<tr>
<td>Return</td>
<td>11%</td>
</tr>
<tr>
<td>Both</td>
<td>33%</td>
</tr>
<tr>
<td>SKIP IMM</td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>11%</td>
</tr>
<tr>
<td>Return</td>
<td>221%</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
References


End