Stitching the Gadgets
On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection

Lucas Davi
Intel Collaborative Research Institute for Secure Computing @ TU Darmstadt, Germany

Joint Work with
Daniel Lehmann, Ahmad-Reza Sadeghi (TU Darmstadt)
Fabian Monrose (UNC Chapel Hill)
Open Question:
Practical and secure mitigation of code reuse attacks

Turing-completeness of return-oriented programming
Hot Research Topic: 
“Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed

- CCFIR [IEEE S&P’13]
- kBouncer [USENIX Sec’13]
- ROPecker [NDSS’14]
- CFI for COTS Binaries [USENIX Sec’13]
- ROPGuard [Microsoft EMET]
- MS BlueHat Prize [USENIX Sec’13]
- MS BlueHat Prize [NDSS’14]

This Talk:
Negative result – all current (published) coarse-grained CFI solutions can be bypassed
Our Contributions

Systematic Security Analysis

Pointing out the conceptual weaknesses of coarse-grained CFI solutions including Microsoft’s EMET
Our Contributions

Our Attack is Generic

Only CFI policies considered, not specific CFI solutions, bypassing even the Über-CFI that combines all CFI policies.

The only work showing Turing-completeness.

Systematic Security Analysis

Generic Attack

Real World Exploits
Our Contributions

Concrete Attack Instantiations under Strong Adversary Model

Very small code base: 840kb
ROP Adversary Model/Assumption

1. Adversary can hijack control-flow (buffer overflow)
2. Adversary knows the memory layout (memory disclosure)
3. Adversary can construct gadgets
4. Adversary can write ROP payload in the data area (stack/heap)

See our JIT-ROP attack
Snow et al, IEEE S&P 2013 & BlackHat USA
Generic ROP Defense: Control-Flow Integrity (CFI)
Restricting indirect control-flow targets to a pre-defined control-flow graph
Original CFI Label Checking

[Abadi et al., CCS 2005 & TISSEC 2009]

**BBL A**

**label_A**

ENTRY

asm_ins, ...

EXIT

**CFI CHECK**

EXIT(A) -> label_B ?

**BBL B**

**label_B**

ENTRY

asm_ins, ...

EXIT
Original CFI: Benefits and Limitations

- Fine-grained protection
- Blackbox Vulcan (unpublished)
- Require side info (debug symbols, compiler support)
- Performance overhead
Coarse-Grained CFI:
Aims at practical CFI for real-world deployment
General Idea

Reducing number of labels
Coarse-Grained CFI Proposals

- kBouncer [USENIX Sec’13]
- ROPecker [NDSS’14]
- ROPGuard [Microsoft EMET]

Win API / Critical Function

Application

Binary Instrumentation

HOOK

HOOK

HOOK

Paging

Last Branch Record (LBR)

Stack

POP

PUSH
Main Coarse-Grained CFI Policies

- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-preceded instruction

- **CFI Policy 2: Behavioral-Based Heuristics**

Threshold Setting
- kBouncer: \( N=8; S\leq 20 \)
- ROPecker: \( N=11; S\leq 6 \)

Application

```
CALL A
asm_ins
asm_ins
CALL B
asm_ins
```

RET

```
CALL C
asm_ins
```

\( > S \)

\( < S \)

\( < S \)

\( < S \)

\( > S \)

\( > S \)

\( < S \)
# Most Restrictive Coarse-Grained CFI

<table>
<thead>
<tr>
<th>CFI Policy</th>
<th>kBouncer</th>
<th>ROPecker</th>
<th>ROPGuard</th>
<th>CFI for COTS Binaries</th>
<th>Üüber-CFI</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI Policy 1</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><em>Call-Preceded Sequences</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CFI Policy 2</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td><em>Behavioral-Based Heuristics</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of CFI Check</td>
<td>WinAPI</td>
<td>2 Pages Sliding Window / Critical Functions</td>
<td>WinAPI/ Critical Functions</td>
<td>Indirect Branch</td>
<td>Any Time</td>
</tr>
</tbody>
</table>

- ✗ No Restriction
- ✓ CFI Policy

*Here only the core policies shown. However, we consider all other deployed policies in our analysis.*
Our Methodology

**Common Library**

*kernel32.dll*

- **Sequence Finder** (IDA Pro)
  - **List of Call-Preceded Sequences**

- **Sequence Filter** (D Program)
  - **Sequence Subset 1**
  - **Sequence Subset n**

**Search for Gadgets**

*Provide filters on Reg, Ins, Opnd, Length*

- **Gadget Generation** (manual)
  - MOV, ESP, LNOP, LOAD, ADD, CALL, XOR, STORE
**Gadget Type** | **CALL-Preceded Sequence ending in a RET instruction**  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOAD Register</strong></td>
<td></td>
</tr>
<tr>
<td>EBP := pop ebp</td>
<td></td>
</tr>
<tr>
<td>ESI := pop esi; pop ebp</td>
<td></td>
</tr>
<tr>
<td>EDI := pop edi; leave</td>
<td></td>
</tr>
<tr>
<td>ECX := pop ecx; leave</td>
<td></td>
</tr>
<tr>
<td>EBX := pop edi; pop esi; pop ebx; pop ebp</td>
<td></td>
</tr>
<tr>
<td>EAX := mov eax,edi; pop edi; leave</td>
<td></td>
</tr>
<tr>
<td>EDX := mov eax,[ebp-8]; mov edx,[ebp-4]; pop edi; leave</td>
<td></td>
</tr>
<tr>
<td><strong>LOAD/STORE Memory</strong></td>
<td></td>
</tr>
<tr>
<td>LD(EAX) := mov eax,[ebp+8]; pop ebp</td>
<td></td>
</tr>
<tr>
<td>ST(EAX) := mov [esi],eax; xor eax,eax; pop esi; pop ebp</td>
<td></td>
</tr>
<tr>
<td>ST(ESI) := mov [ebp-20h],esi</td>
<td></td>
</tr>
<tr>
<td>ST(EDI) := mov [ebp-20h],edi</td>
<td></td>
</tr>
<tr>
<td><strong>Arithmetic/Logical</strong></td>
<td></td>
</tr>
<tr>
<td>ADD/SUB := sub eax,esi; pop esi; pop ebp</td>
<td></td>
</tr>
<tr>
<td>XOR := xor eax,edi; pop edi; pop esi; pop ebp</td>
<td></td>
</tr>
<tr>
<td><strong>Branches</strong></td>
<td></td>
</tr>
<tr>
<td>unconditional branch 1 := leave</td>
<td></td>
</tr>
<tr>
<td>unconditional branch 2 := add esp,0Ch; pop ebp</td>
<td></td>
</tr>
<tr>
<td>conditional LD(EAX) := neg eax; sbb eax,eax; and eax,[ebp-4]; leave</td>
<td></td>
</tr>
</tbody>
</table>

8/21/2014
Long NOP Gadget

ROP Gadget 1 → Store Registers → Prepare Long NOP → Long NOP → Reset Registers

- ESI
- EDI
- EBX

Stack

Arbitrary Data Area (36 Bytes)

ROP Gadget 2
Real-World Exploitation

Adobe Reader 9.1
CVE-2010-0188

MPlayer Lite r33064 m3u
Buffer Overflow Exploit

Both detected by
Microsoft EMET and
coarse-grained CFI

Common Library
kernel32.dll

Exploit Transformation

Successful Exploit
Details in Paper
Lessons Learned

- Fundamental problems of coarse-grained CFI
  1. Too many call sites available
     * Turing-complete gadget set possible on small code base
       → Restrict returns to valid/hot call sites (shadow stack)
  2. Heuristics are ad-hoc and ineffective
     * Our generic Long NOP evades heuristics and handles all side effects
       → Adjusted sequence length leads to high false positive
  3. Too many indirect jump and call targets
     * Resolving indirect jumps and calls is non-trivial
     * Original CFI: Vulcan framework remains blackbox
       → Compromise: Compiler support
Summary and Future Work

• Summary
  • We systematically analyze the security of all recently proposed “practical” defenses against code reuse attacks
  • We show Turing-completeness of ROP even over less than 1MB code base
  • And, we provide real-world exploits

• Our ongoing work: fine-grained CFI
  • Hardware/Software Co-design [DAC 2014]; collaboration with Intel
  • Software Solutions; collaboration with Microsoft

• Future work
  • CFI for dynamic code or JIT-enabled applications
  • Improved code randomization schemes
Thank you!

For more information see
www.trust.cased.de