Opaque Control-Flow Integrity

Vishwath Mohan\textsuperscript{\$}, Per Larsen\textsuperscript{\$}, Stefan Brunthaler\textsuperscript{\$}, Kevin W. Hamlen\textsuperscript{\$}, Michael Franz\textsuperscript{\$}

The University of Texas at Dallas\textsuperscript{\$} University of California, Irvine\textsuperscript{\$}
Code Reuse Attacks

• Needs
  – Location of code
  – Hijack control-flow

• Defensive options
  – Randomization
  – Control Flow Integrity
Where does that leave us?

Randomization

CFI

Fine-grained CFI
Opaque Control-Flow Integrity

Randomization  O-CFI  CFI
Traditional CFI

\[ \Theta_1 \leq \beta \leq \Theta_3 \]
Randomization I: Shuffle clusters
Randomization II: Shuffle basic blocks
Accelerated Bounds Checks

- MPX mode on supported chipsets

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bndmov bnd, m64</td>
<td>Move upper and lower bound from m64 to bound register bnd.</td>
</tr>
<tr>
<td>bndcl bnd, r/m32</td>
<td>Generate a #BR if r/m32 is less than the lower bound in bnd.</td>
</tr>
<tr>
<td>bndcu bnd, r/m32</td>
<td>Generate a #BR is r/m32 is higher than the upper bound in bnd.</td>
</tr>
</tbody>
</table>

- Legacy mode as fallback
Performance

Legacy mode overhead 4.7%

MPX mode overhead ≈ 4.2%
Security

• When the CFI policy is opaque

<table>
<thead>
<tr>
<th>Gadget Chain Size</th>
<th>Chance(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>3</td>
<td>0.8</td>
</tr>
<tr>
<td>4</td>
<td>0.01</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
</tr>
</tbody>
</table>
Security

• When the CFI policy is not opaque
  – Expressed as CSP
  – Attempted constructing `VirtualAlloc` payload
  – Across Mona/custom tool, no payload found
Conclusion

• Coarse-grained CFI with randomization
  – Advantages of both
• Effective against state-of-the-art exploits
  – JIT-ROP, BROP, Gadget stitching
• Efficient
  – 4.7% overhead in legacy mode
Thank you
Extra Resources
Optimizing Guards

• Actual guard implementation
  – PittSFIeld inspired guards
  – Want minimal chunk size
  – Comparison instructions rather large (~ 7 bytes)

• How efficient can we be?
## Optimizing Guards

<table>
<thead>
<tr>
<th>Description</th>
<th>Original Code</th>
<th>Rewritten Code (MPX-mode)</th>
<th>Rewritten Code (Legacy-mode)</th>
</tr>
</thead>
</table>
| Indirect Branches | `call/jmp r/[m]` | 1: `mov [esp-4], eax`  
2: `mov eax, r/[m]`  
3: `cmp byte ptr [eax], 0xF4`  
4: `cmovz eax, [eax+1]`  
— chunk boundary —  
5: `bndmov bndl, gs:branch_id`  
6: `bndcu bndl, eax`  
7: `jmp` | 1: `push ecx`  
2: `push eax`  
3: `mov eax, r/[m]`  
4: `cmp byte ptr [eax], 0xF4`  
5: `cmovz eax, [eax+1]`  
— chunk boundary —  
6: `mov ecx, branch_id`  
7: `cmp eax, gs:[ecx]`  
8: `jb`  
9: `cmp gs:[ecx+4], eax`  
— chunk boundary —  
10: `jbe abort`  
11: `and al, align_mask`  
12: `xchg eax, [esp]`  
13: `pop ecx`  
14: `pop ecx`  
15: `call/jmp [esp-8]` | 2: `cmp eax, gs:branch_id`  
3: `jbe abort` |
| Returns           | `ret (n)`     | — chunk boundary —  
1: `xchg eax, [esp]`  
2: `and al, align_mask`  
3: `bndmov bndl, gs:branch_id`  
4: `jmp`  
— chunk boundary —  
5: `xor eax, eax`  
6: `bndcu bndl, eax`  
7: `bndcl bndl, eax`  
8: `xchg eax, [esp]`  
9: `ret (n)` | — chunk boundary —  
1: `xchg eax, [esp]`  
2: `cmp eax, gs:branch_id`  
3: `zb`  
4: `and al, align_mask`  
— chunk —  
5: `cmp eax, gs:branch_id + 4`  
6: `jae`  
7: `xchg eax, [esp]`  
8: `ret (n)`  
— chunk boundary —  
9: `jmp abort` | 3: `and al, align_mask`  
4: `cmp eax, gs:branch_id + 4`  
5: `jae`  
6: `xchg eax, [esp]`  
7: `ret (n)`  
8: `jmp abort` |
Coarse Grained Insecurity

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DeMott [12]</td>
<td>Feb 2014</td>
<td>☹</td>
<td>☹</td>
<td>☹</td>
<td></td>
<td></td>
<td>☹</td>
</tr>
<tr>
<td>Göktas et al. [18]</td>
<td>May 2014</td>
<td>☹</td>
<td>☹</td>
<td>☹</td>
<td></td>
<td></td>
<td>☹</td>
</tr>
</tbody>
</table>