Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification

Jason Gionta, William Enck
North Carolina State University

Per Larsen
UC Irvine
Kernel Attack Vectors

• Bypass access controls
  – SELinux

• Modify non-control data
  – Data integrity  [Kuzentsov et al. 2014]

• Code-reuse ← our focus
  – Return Oriented Rootkits
  – Enable arbitrary memory writes
Code-Reuse Attacks – Simple Example

What an attacker knows?
- Buffer overflow
- Static binary load location
- Static location of gadgets

Overwritten Return Address
Kernel Code Reuse Protections

• Requirements
  – Adoptability
  – Deployability

• Approaches
  – Policy/Access Control
    • Non-comprehensive
  – Control Flow Integrity
    • Large overhead
    • Practical challenges
  – **Software diversity**
    • Low overhead
    • Access to sources
Existing Approaches for Kernel Software Diversity

• Kernel ASLR
  – Already deployed
  – Too coarse-grained

• Function randomization
  – Limited protection

• Fine-grained instruction diversity
  – Not comprehensive

• Re-randomization
  – Micro-kernel modules only
  – Costly
Existing Approaches for Kernel Software Diversity

- Kernel ASLR
  - Already deployed

Fail to address Memory Disclosure

- Micro-kernel modules only
- Costly
Memory Disclosure

- Executable Data is Readable
- Execute permissions imply read permissions
  - Required for execution
- Protections rely on memory secrecy
  - Software diversity
    - Just-In-Time Code Reuse [Snow et al. 2013]
    - Relaxed control flow enforcement
      - Out-of-Control: Overcoming Control Flow Integrity [Gotkas et al. 2014]
- Memory disclosure vulnerabilities
  - Originate from memory corruption bugs
  - Leak raw memory
    - Information Leaks Without Memory Disclosure [Seibert et al. 2014]
Challenges

• Comprehensive
  – All code must be randomized
  – Leaked code pointers imply gadget offsets

• Protect kernel code against disclosure
  – Support reading of kernel code
  – Existing approaches rely on kernel
KHide Code Reuse Protections

- Comprehensive fine-grained instruction diversity of kernel code
  - Address limitations of existing approaches
  - Apply to Linux kernel
- Protect against disclosure of kernel code
  - Support existing code reading requirements
- Result:
  - No gadgets exist across diversified kernels
  - Adversaries must guess code reuse gadgets
KHide Architecture

• Comprehensive fine-grained instruction diversity of kernel code ← offline
• Protect against kernel code disclosure ← online
Challenges

• Comprehensive
  – All code must be randomized
  – Leaked code pointers imply gadget offsets

• Protect kernel code against disclosure
  – Support reading of kernel code
  – Existing approaches rely on kernel
Existing Fine-Grained Instruction Diversity

- Diversity of kernel sources
- NOP insertions [Homescu et al. 2013]
  - Low overhead
  - Native compiler support
    - No assembly support
  - Not enough

\begin{verbatim}
known a priori
0xF000: push rbp
      ...
0xFF10: call 0x5400 ; printk
0xFF13: pop rax
        pop rbx
        pop rbp
        ret

No Diversity
\end{verbatim}

\begin{verbatim}
0xF000: push rbp
      ...
0xFF14: call 0x5400 ; printk
0xFF17: nop
        pop rax
        nop
        pop rbx
        pop rbp
        ret

NOP Insertions
\end{verbatim}
Comprehensive Fine-Grained Instruction Diversity

- **Problem 1: diversify Linux kernel**
  - LLVM/Linux
  - Clang/LLVM - NOP insertions [Homescu et al. 2013]
- **Problem 2: assembly source files**
  - Micro-Architectural Optimizer (MAO) [Hundt et al. 2011]
    - Decompose post-processed assembly
    - _NOP insertions_
Problem 3: Leaked pointers can imply gadgets

0xF000: push rbp
... 0xFFFF: call 0x5400 ; printk
0xFFF10: pop rax
0xFFF13: pop rbx
0xFFF13: pop rbp
ret

stacks for printk

No Diversity

Leaked Return Address

0xFFF13
saved rbp
printk local

0xFFF17
saved rbp
printk local

0xFFF00: push rbp
... 0xFFFF: call 0x5400 ; printk
0xFFF14: nop
0xFFF17: nop
pop rax
pop rbx
pop rbp
ret

NOP Insertions

Shared Identical Gadget
Problem 3: Leaked pointers can imply gadgets

- Register randomization ← novel approach for LLVM

- Modify selection order

- Some register remain static

  - pop rbp; ret;

\[
\begin{align*}
0xF000: & \quad \text{push rbp} \\
       & \quad \text{nop} \\
       & \quad \text{...}
\end{align*}
\]
\[
\begin{align*}
0xFF14: & \quad \text{call 0x5400; printk} \\
0xFF17: & \quad \text{nop} \\
       & \quad \text{pop rax} \\
       & \quad \text{nop} \\
       & \quad \text{pop rbx} \\
       & \quad \text{pop rbp} \\
       & \quad \text{ret}
\end{align*}
\]

\[
\begin{align*}
0xF000: & \quad \text{push rbp} \\
       & \quad \text{nop} \\
       & \quad \text{...}
\end{align*}
\]
\[
\begin{align*}
0xFF14: & \quad \text{call 0x5400; printk} \\
0xFF17: & \quad \text{nop} \\
       & \quad \text{pop rsi} \\
       & \quad \text{nop} \\
       & \quad \text{pop rdx} \\
       & \quad \text{pop rbp} \\
       & \quad \text{ret}
\end{align*}
\]
Comprehensive Fine-Grained Instruction Diversity

• Problem 3: Leaked pointers can imply gadgets
  – Call Site Lifting ← novel contribution
  • Decouple return address
  • Cannot determine location of gadget

0xF000: push rbp
         nop
         …
         call 0x5400 ; printk
         nop
         pop rax
         nop
         pop rbx
         pop rbp
         ret

0xFF14:

0xFF17: NOP Insertions

+Register Randomization

0xF000: push rbp
         nop
         …
         call 0x5400 ; printk
         nop
         pop rsi
         nop
         pop rdx
         pop rbp
         ret

0xFF63:

0xF009: +Call Site Lifting

0xF004: call 0x5400 ; printk

0xF000: jmp 0xF100

0xF004: jmp 0xFF63

0xF000: nop

0xF000: push rbp

0xF000: nop
Challenges

• Comprehensive
  – All code must be randomized
  – Leaked code pointers imply gadget offsets

• Protect kernel code against disclosure
  – Support reading of kernel code
  – Existing approaches rely on kernel
Memory Disclosure Protection

• Existing approach limitations
  – XnR [Backes et al. 2014]
    • Limited protection model
    • Kernel enforces protection
  – HideM
    • Reliance on split-TLB
      – Limited support in new architectures
      – TLB pressure
    • Kernel enforces protection
• Apply and enforce protections outside kernel
Memory Disclosure Protection

• Virtualization Extensions
  – Hardware Assisted Paging
  – Execute-only support

• Enforce code reading policy
Memory Disclosure Protection

• Protect Executable Pages
  – Register executable-pages
  – On-boot and On-demand

• Pair with readable data pages
  – Data that may be read (code reading policy)
Build Code Reading Policy

• Find executable data to be read
  – Instrument compiler
    • Record locations of non-code is written to code section
    • Write locations to read-only data in kernel image
  – Function tracing support
    • kprobes/ftrace
      – Read first byte; write interrupt
    • Insert NOP instruction to beginning of all functions
Empirical Evaluation: Setup

- LLVM Linux kernel 3.18
  - 5 diversified kernels generated
    - Performance
    - Security
  - NOPs inserted at 50% probability [Homescu et al. 2013]
- KHide built on KVM
- Performance: three configurations
  - Native
  - Diversity
  - KHide
Empirical Evaluation: Performance

- Microbench
  - LMBench [McVoy et al. 1996]

<table>
<thead>
<tr>
<th>Test</th>
<th>Native</th>
<th>Diverse</th>
<th>KHide</th>
<th>ASLP</th>
<th>KCoFI</th>
<th>VGhost</th>
</tr>
</thead>
<tbody>
<tr>
<td>page fault</td>
<td>0.139</td>
<td>0.153</td>
<td>10%</td>
<td>0.151</td>
<td>8.9%</td>
<td>-</td>
</tr>
<tr>
<td>null syscall</td>
<td>0.0352</td>
<td>0.0388</td>
<td>10%</td>
<td>0.0384</td>
<td>9%</td>
<td>-</td>
</tr>
<tr>
<td>sig. handler install</td>
<td>0.101</td>
<td>0.117</td>
<td>15.2%</td>
<td>0.116</td>
<td>14.6%</td>
<td>-</td>
</tr>
<tr>
<td>sig. handler delivery</td>
<td>0.622</td>
<td>0.749</td>
<td>20.4%</td>
<td>0.743</td>
<td>19.6%</td>
<td>-</td>
</tr>
<tr>
<td>fork + exit</td>
<td>77.29</td>
<td>93.83</td>
<td>31.4%</td>
<td>92.8</td>
<td>20%</td>
<td>-</td>
</tr>
<tr>
<td>fork + exec</td>
<td>81.38</td>
<td>100.4</td>
<td>23.4%</td>
<td>99.38</td>
<td>22.1%</td>
<td>12.52%</td>
</tr>
<tr>
<td>select</td>
<td>4.7</td>
<td>6.87</td>
<td>46%</td>
<td>6.84</td>
<td>45.4%</td>
<td>-</td>
</tr>
<tr>
<td>open/close</td>
<td>0.777</td>
<td>1.191</td>
<td>53.3%</td>
<td>1.17</td>
<td>50.7%</td>
<td>-</td>
</tr>
</tbody>
</table>

File creations per second

<table>
<thead>
<tr>
<th>File Size</th>
<th>Native</th>
<th>Diversity</th>
<th>KHide</th>
<th>KCoFI</th>
<th>VGhost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 KB</td>
<td>192k</td>
<td>125k</td>
<td>35%</td>
<td>126k</td>
<td>34%</td>
</tr>
<tr>
<td>1 KB</td>
<td>125k</td>
<td>79.7k</td>
<td>36%</td>
<td>80.6k</td>
<td>35%</td>
</tr>
<tr>
<td>4 KB</td>
<td>123k</td>
<td>79.2k</td>
<td>35%</td>
<td>80k</td>
<td>33%</td>
</tr>
<tr>
<td>10 KB</td>
<td>92.5k</td>
<td>60k</td>
<td>35%</td>
<td>60.4k</td>
<td>35%</td>
</tr>
</tbody>
</table>
Empirical Evaluation: Performance

- **Macrobench: Network**
  - SSH File Transfer – files 1KB-1GB
    - Average: 1%
    - High: 10%
    - KCoFI: 14%
Empirical Evaluation: Memory Overhead

• Kernel image size
  – bzImage
    • 5.1MB (No diversity) → 6.3MB (23%)
  – In memory code
    • 7.9MB (No diversity) → 10MB (27%)

• Code reading policy
  – One shared shadow read page
    • No embedded jump-tables
  – 4KB
Empirical Evaluation

• Security
  – Attacker Model:
    • Attackers guess gadgets
    • Find common gadgets across diversified versions
  – Calculate gadget survivability [Homescu et al. 2013]
    • Find identical gadgets at same offsets
      – Accounting for NOPs
      – Across diversified versions

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td># Survived</td>
<td>6258</td>
<td>116</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Conclusion

- **KHide** provides comprehensive code-reuse protection for kernels
  - All code diversified
  - Addresses limitations of previous diversity techniques
  - Provides runtime protection against code disclosure

- **Adoptable design**
  - Applied to Linux kernel
  - Low performance impact
  - No gadgets survive diversification
Thanks

• Questions?

✉️ jason@gionta.org
🖥️ gionta.org
Empirical Evaluation: Performance

- **Macrobench: Network**
  - ApacheBench – 32 clients 10,000 requests
    - Apache webserver
    - Average: 4%
    - High: 9%
    - ASLP: 14%
      - 100 Clients
    - KCoFI: 0%
      - thttpd
Empirical Evaluation: Performance

- Macrobench: Disk
  - Postmark – simulate email server
  - 500,000 transactions

<table>
<thead>
<tr>
<th>Config</th>
<th>Time(s)</th>
<th>Std. Dev</th>
<th>95% Conf.</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native</td>
<td>13.57</td>
<td>0.188</td>
<td>±0.052</td>
<td></td>
</tr>
<tr>
<td>Diversify</td>
<td>17.3</td>
<td>0.974</td>
<td>±0.12</td>
<td>28%</td>
</tr>
<tr>
<td>KHide</td>
<td>16.63</td>
<td>1.31</td>
<td>±0.16</td>
<td>23%</td>
</tr>
<tr>
<td>KCoFI</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>96%</td>
</tr>
<tr>
<td>VGhost</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>372%</td>
</tr>
</tbody>
</table>