Defeating Return-Oriented Rootkits with "Return-less" Kernels

Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, Sina Bahram

Department of Computer Science
North Carolina State University
Outline

1. Motivation
2. Background: Return-Oriented Programming
3. Our Approach
4. Implementation & Evaluation
5. Conclusions
Motivation: Rootkit Threats

Number Of New Rootkits

<table>
<thead>
<tr>
<th>Year</th>
<th>Number of Rootkits</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>639</td>
</tr>
<tr>
<td>2007</td>
<td>1381</td>
</tr>
<tr>
<td>2008 Jan-Jun</td>
<td>2471</td>
</tr>
</tbody>
</table>

Kernel Rootkits

- Rootkit goals:
  - Maintain undetectable presence
  - Keep privileges, hide malicious activities, etc.

- Need to execute code at kernel privilege
Rootkit Examples

**adore-ng**
- Linux 2.4/2.6
- Kernel module
- Adds “custom” functions
- Hooks VFS

**SucKIT**
- Linux 2.4
- /dev/kmem
- Adds “custom” functions
- Hooks system calls
Existing Defenses

- **Attestation**: Is the code to be executed expected?
  - e.g., driver signing

- **Kernel Code Integrity (W⊕X)**: Make kernel code read-only and kernel data non-executable
  - **SecVisor** [Seshadri et al., SOSP 2007], **NICKLE** [Riley et al., RAID 2008]

- **Control Flow Integrity (CFI)**: Detect unexpected execution paths
  - **SBCFI** [Petroni et al., CCS 2007]
Arms Race

1. $W \oplus X$ prevents injected code from executing
2. Return-to-libc defeats $W \oplus X$ by returning to an existing function
   - e.g., system (char *command)
3. Return-Oriented Programming (ROP) generalizes return-to-libc by misusing legitimate code fragments or gadgets
   - [Shacham et al., CCS ’07], [Buchanan et al., CCS ’08], [Hund et al., USENIX Sec ’09]
4. Our technique de-generalizes ROP back to return-to-libc
Return-Oriented Programming

Gadget

- A sequence of instructions ending with a `ret` to complete an operation

Stack

- A manipulated stack to chain gadgets together
Return-Oriented Programming

- **Turing-complete**: allowing for performing arbitrary computation based on gadgets
  - [Shacham et al., CCS ’07], [Buchanan et al., CCS ’08], [Hund et al., USENIX Sec ’09]
  - Return-to-libc attacks are not Turing-complete

- Example gadgets
  - Load/store, arithmetic, logic, control flow, system calls, ...
Our Defense Against ROP

- Observation: Each gadget ends with a `ret`

**Can we build a kernel without using ret?**
Finding `ret`

- FreeBSD 8.0/x86-amd64:
  - 8,329 ret instr. / 675,763 total instr. = 1/81

- Additional challenge in x86: variable-length instructions
  - +10,001 additional ret opcodes
  - 18,330 ret opcodes / 675,763 total instr. = 1/37
Finding `ret`

- Example: `AcpiUtDeleteRwLock` (FreeBSD 8.0/x86-amd64 function)

```
48 8b 3b         mov (%rbx), %rdi
  e8 c3 e0 00 00  callq <AcpiOsDeleteMutex>

3b e8           cmp %eax, %ebp
  c3             retq
```
Our Defense Against ROP

Can we build a kernel without return opcodes?
Our Approach

- **Return Opcode**
  - (18,330)
- **Instruction Operands**
  - (8,337)
  - Subdivisions:
    - **Return Instructions**
      - (8,329)
    - **Non-Return Instructions**
      - (8)
    - **Immediate Operands**
      - (2,923)
    - **Register Operands**
      - (7,070)

Connections:
- **Return Indirection**
- **Peephole Optimization**
- **Register Allocation**
Return Indirection: Removing return instructions

- Naïve approach: ret → pop + jmp
  - This still can be used to make gadgets!

- Our approach: return address → return index
  - call dst → push $index; jmp dst
  - ret → pop %reg; jmp *RetAddrTable(%reg)
Return Indirection: Removing return instructions

- Return indirection de-generalizes ROP back to return-to-libc
- Additional control-flow integrity checks can be enforced to defeat return-to-libc
  - e.g., function type-based validation
Register Allocation

- Adjust register operands when being allocated to avoid introducing `ret` opcodes

- Example:
  
  48 89 c3  mov  %rax, %rbx  (rbx introduces `ret` opcode)

  49 89 c5  mov  %rax, %r13  (replace rbx with r13)
Peephole Optimization

- Non-return instructions
  - `0f c3 ...`
  - `movnti mem32, reg32`
  - `89 ...`
  - `mov mem32/64, reg32/64`

- Immediate operands
  - `cmp $0xc3, %ecx`
  - `mov $0xc4, %reg`
  - `dec %reg`
  - `cmp %reg, %ecx`

- Relative offset
  - `e9 c3 00 00 00`
  - `jmpq 0xffffffff801a4fc9`
  - `e9 c4 00 00 00 90`
  - `jmpq 0xffffffff801a4fca`
  - `nop`
Implementation & Evaluation

- Our prototype, based on LLVM, is used to compile a FreeBSD 8.0 kernel
  - FreeBSD OS image file size increased by 9.4%
    - 4,370,704 bytes → 4,782,952 bytes
  - Instruction count increased by 10.9%
    - 675,763 → 749,227

No return opcodes!
Implementation & Evaluation

- LMbench microbenchmark

![Graph showing performance overhead comparison between Return-less Kernel without Type-based Validation and Return-less Kernel with Type-based Validation.](graph.png)
Limitations

- Kernel source code access
- Return-oriented attacks ➔ return-to-libc
- Jump-oriented programming?
Related Work

- **Return-Oriented Rootkit Detection**
  - e.g. DROP [Chen et al., ICISS ’09], DynIMA [Davi et al., STC ’09]...

- **Kernel Rootkit Prevention**
  - e.g. SecVisor [Seshadri et al., SOSP ’07], Lares [Payne et al., Oakland ’08], NICKLE [Riley et al., RAID ’08]...
Conclusions

- Compiler-based approach to defeat ROP
  - Return indirection
  - Peephole optimization
  - Register allocation

- Result: Return-less Kernel
  - No ret opcode ➔ no return-oriented gadgets
Thank you!
Return Opcodes

- x86 has four ret opcodes
  - c2, c3, ca, cb
  - All handled
CFI vs. Return-less Kernels

- Major CFI approaches:
  - Control transfer instruction instrumentation
  - Data Flow Integrity
  - Memory-safe programming languages
  - Software bounds checking

- Challenging to apply in kernel space

- Return-less approach complements CFI