Ensuring the Minimality of Included Kernel Components

Dannie Stanley

November 18th 2013
Outline

Code Minimization

Code Verification

Code Diversification

Code Minimization

Introduction

Problem

Approach

Evaluation

Summary
Outline

Code Minimization

Code Verification

Code Diversification

Code Minimization

Introduction

Problem

Approach

Evaluation

Summary
NICKLE Guarantee:
“No unauthorized code can be executed at the kernel level.”

✓ Kernel code is authenticated prior to execution
✗ Authorized code may be executed in malicious combinations
Return-Oriented Programming (ROP)
Return-oriented programming reuses library code for malicious purposes

- Attacks work against $W \oplus KX$ protected systems
  - Only authorized code is re-used
- Kernels contain sufficient authorized code to perform arbitrary computations
Monolithic Kernels are Large
Proponents of microkernel designs criticize monolithic designs used in many current commodity operating systems as having too much code running in kernel mode.

- Commodity kernels have extraneous code to support many potential applications
  - Linux has 15 million lines of code
  - Windows has 50 million lines of code
**Poly² Framework**

One component of the Poly² framework is to apply the *economy of mechanism* to the operating system kernel

- ✓ Specialize kernel for specific application
  - • Remove extraneous code
  - • Remove extraneous system calls
- ✓ Specialization happens at compile-time
Problem:

- Return-oriented programming can be used to attack systems that are protected by \( W \oplus KX \) mechanisms
- Operating system kernels are large and have lots of “flesh on the bone”
We can improve the security of a general-purpose operating system kernel by specializing it for specific applications at run-time.
Remove "flesh on the bone" at run-time.
Code Minimization Approach

Kernel Space

Unprotected

Activated Code (flesh on the bone)

Deactivated but Potentially Activated Code

Protected

Deactivated Code (or data)

---

[Diagram showing a visual representation of activated and deactivated code spaces in a kernel context.]
How do we specialize a kernel at run-time?

✓ **Deactivate** as much code as possible.
  • Replace instructions in memory (function-level)
    - OR -
  • Make code non-executable (page-level)

✓ **Reactivate** code in an "on-demand" fashion
  • If code is accessed from a valid start address
### Disassembled Kernel Function

c102da10 <module_attr_show>:
c102da10:  83 ec 08  sub $0x8,%esp
        c102da13:  89 74 24 04  mov %esi,0x4(%esp)
c102da17:  89 d6  mov %edx,%esi
        c102da19:  89 1c 24  mov %ebx,(%esp)
c102da1c:  8b 5a 08  mov 0x8(%edx),%ebx
        c102da1f:  85 db  test %ebx,%ebx

### Bytes in Kernel Memory

<table>
<thead>
<tr>
<th>i</th>
<th>ec</th>
<th>08</th>
<th>89</th>
<th>74</th>
<th>24</th>
<th>04</th>
<th>89</th>
<th>d6</th>
<th>89</th>
<th>1c</th>
<th>24</th>
<th>8b</th>
<th>5a</th>
<th>08</th>
<th>85</th>
<th>db</th>
</tr>
</thead>
</table>

### Executed Functions

Security Monitor Storage

Security Monitor

Approach
Disassembled Kernel Function

c102da10 <module_attr_show>:
c102da10:  83  ec  08          sub    $0x8,%esp
          89  74  24  04      mov    %esi,0x4(%esp)
c102da17:  89  d6                mov    %edx,%esi
          89  1c  24            mov    %ebx,(%esp)
c102da1c:  8b  5a  08            mov    0x8(%edx),%ebx
          85  db                test   %ebx,%ebx
          ...

Bytes in Kernel Memory

Security Monitor Storage

Security Monitor Storage
Code Minimization

Approach

Start → Deactivate Kernel Functions Not On Whitelist → Whitelist

Run → Activate Function

Interrupt

On Blacklist?

Y → Blacklist

N → Add to Live Functions → Live Functions

Addr Function Start?

N →

Ancestor Functions Live?

N →

Check Ancestor Liveness → CFG

Y →
Code Minimization Approach

1. Start
2. Deactivate Kernel Functions Not On Whitelist
3. Whitelist
4. Run
5. Activate Function
6. Interrupt
7. On Blacklist?
   a. Y
      i. Blacklist
      ii. Add to Live Functions
   b. N
      i. N
         a. Addr Function Start?
         b. N
            i. CFG
            ii. Ancestor Functions Live?
               a. N
                  i. CFG
               b. Y
                  i. Check Ancestor Liveness
         ii. Y
            a. CFG
            b. Live Functions
9. N
   Y
      a. CFG
      b. Live Functions
List of functions that cannot be deactivated

For in-kernel security monitors this list may be large.
List of functions that are not needed by the applications of the system

This list is determined ahead of time as in Poly²
Call graph of kernel functions

A function cannot be activated unless ancestors are activated
List of activated functions

Functions start deactivated and are activated in a Just-In-Time fashion
Code Minimization

Approach

- Start
- Deactivate Kernel Functions Not On Whitelist
- Whitelist

- Run
- Activate Function

- Interrupt
- On Blacklist?
  - Y
    - Blacklist
  - N
    - Addr Function Start?
      - N
        - Add to Live Functions
      - Y
        - Ancestor Functions Live?
          - N
            - Check Ancestor Liveness
          - Y
            - CFG

- Y
  - Live Functions
Outline

Code Minimization

Code Verification

Code Diversification

Code Minimization

Introduction

Problem

Approach

Evaluation

Summary
Implemented VMM-based security monitor named KIS based on KVM

• Code deactivated at function-level
• Protect unmodified Linux guest (guest transparent)
Common Web Server Workload

- LAMP
  - Linux, Apache, MySQL and PHP
- Wordpress
• 78.12% of the 28,828 kernel functions were unused for our workload
Web Server Workload

- Within one minute 94% of reactivated functions were activated
ROPgadget
- Found 92 reusable chunks of code in Linux kernel
- 64/92 would fail to run for our workload

Evaluation

Code Minimization
Introduction
Problem
Approach
Evaluation
Summary
Summary

• Kernels have excessive code that can be reused for ROP-style attacks

• We can reduce this vulnerability by minimizing the kernel at run-time
Thank You