University of Mannheim, Germany
Laboratory for Dependable Distributed Systems

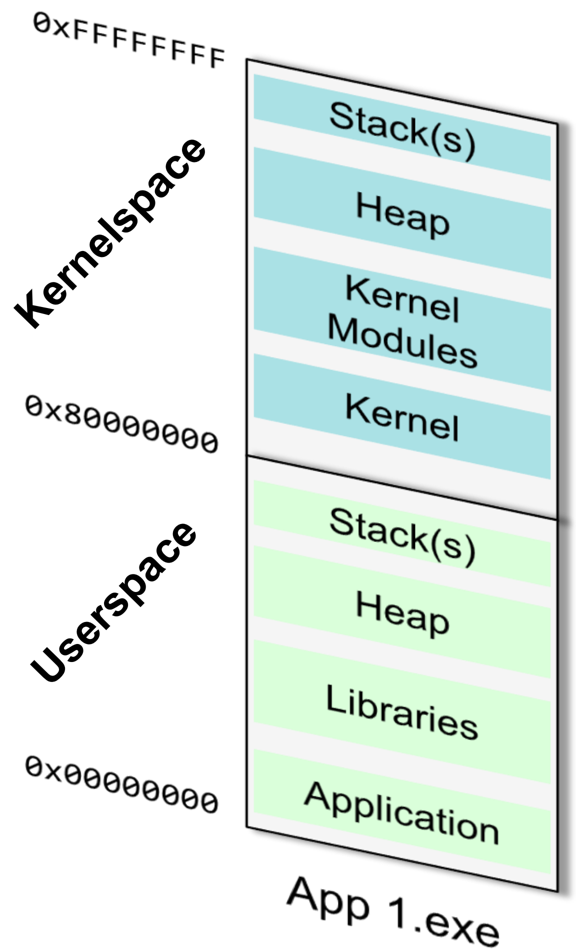
Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms

Ralf Hund
University of Mannheim

SPRING
14. September 2009

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

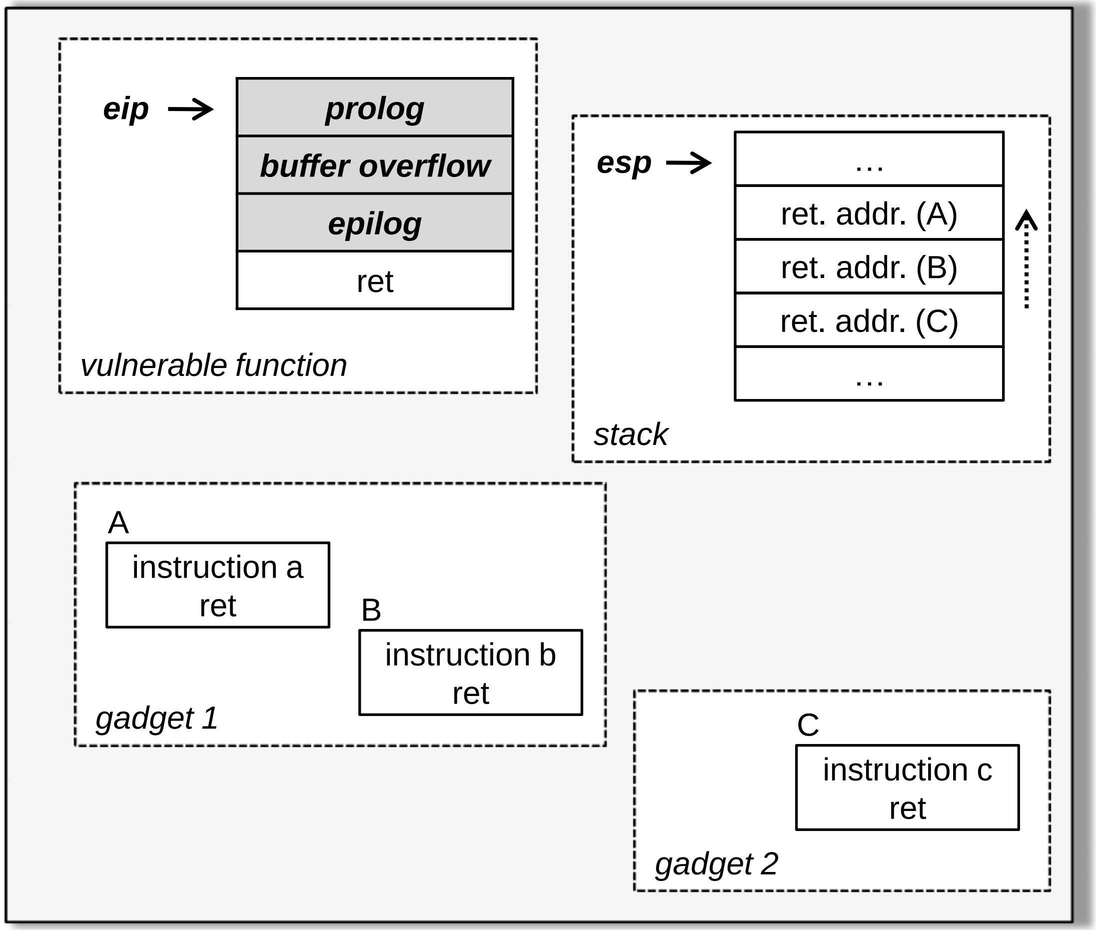


Motivation (2)

- 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 **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**



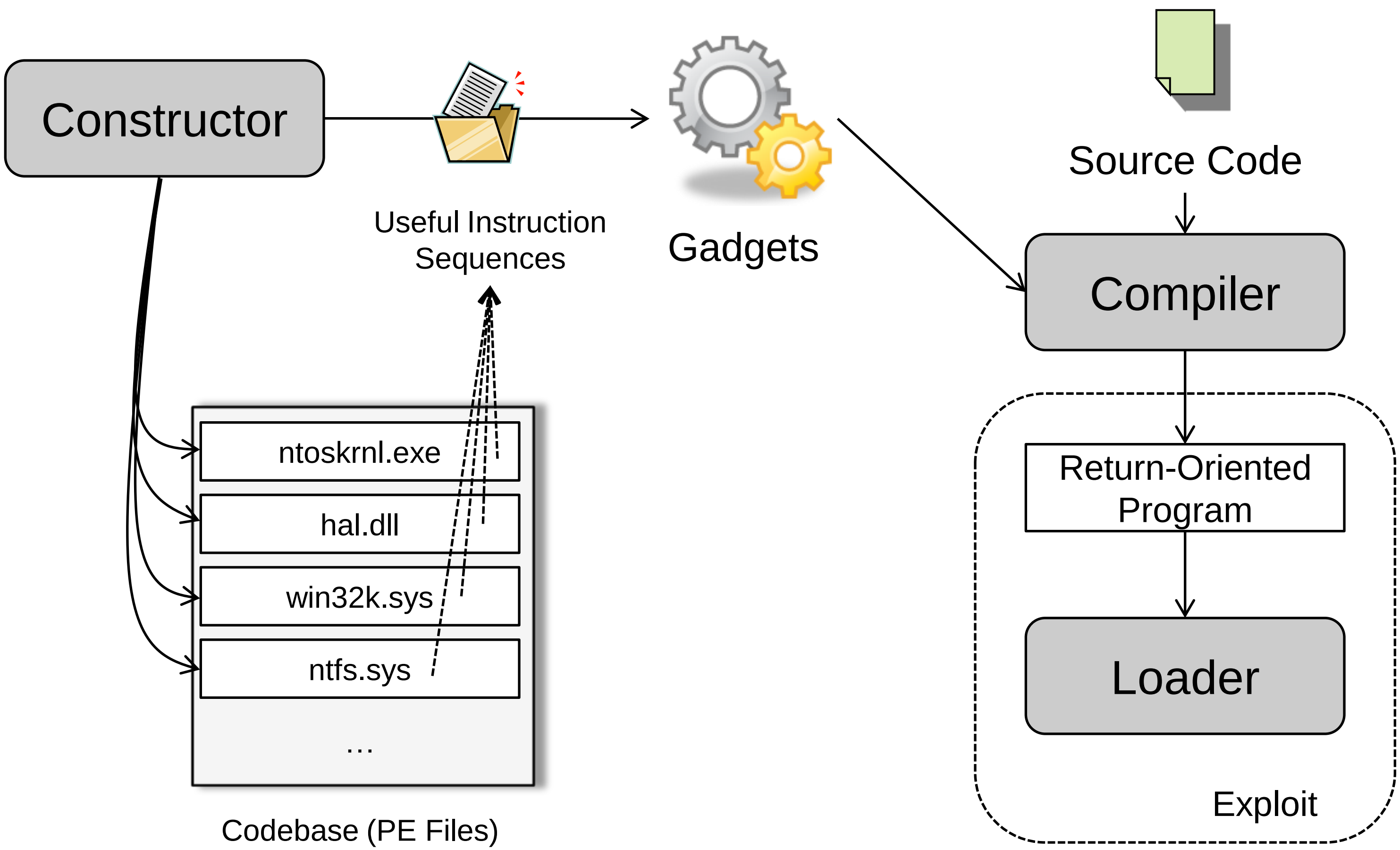
Overview

- 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



Useful Instruction Sequences

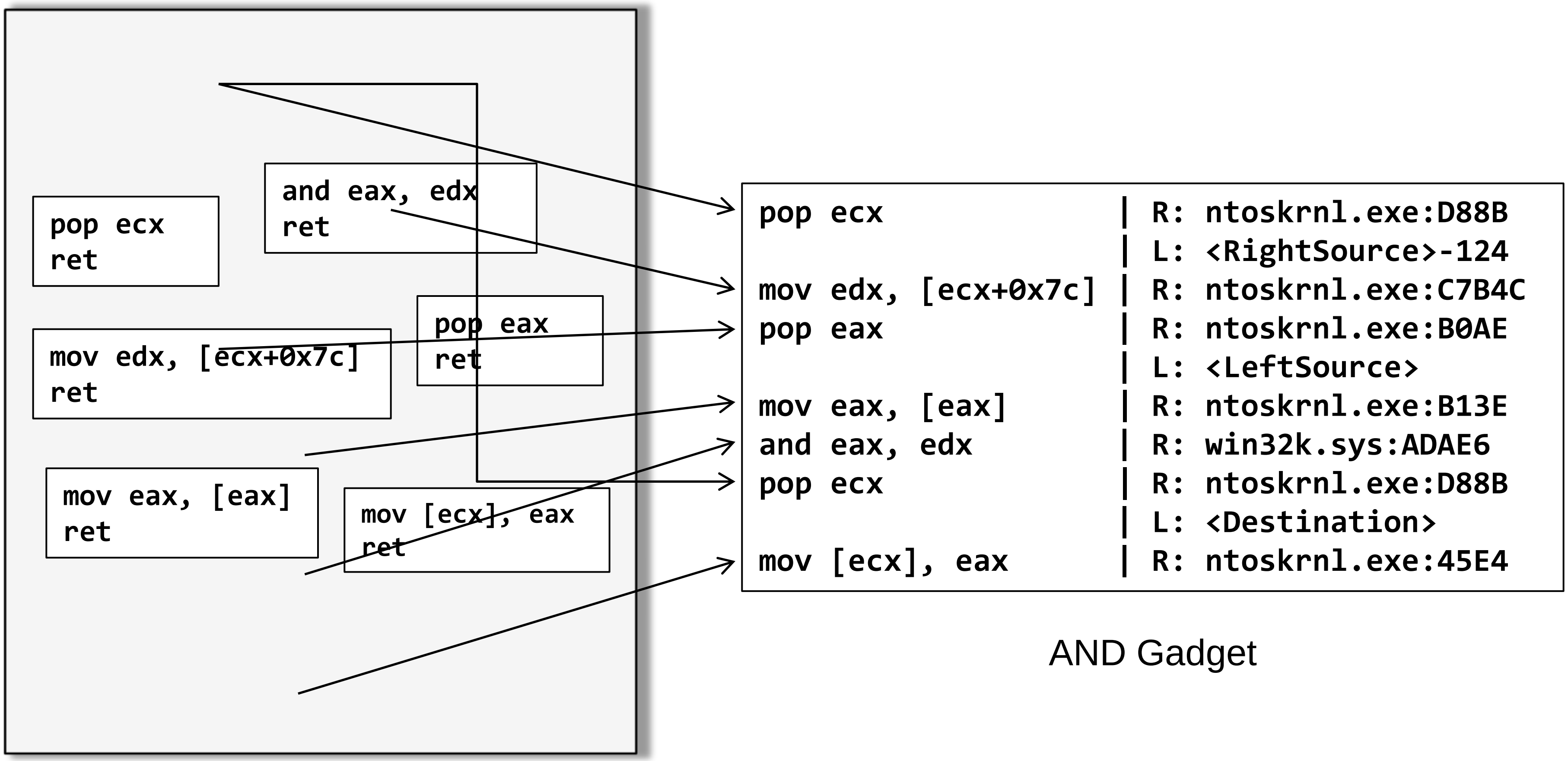
- **Definition**: instruction sequence that ends with a return
- How many instructions preceding a return should be considered?
  - ➡ Must take **side-effects** into account
  - ➡ Simplifying assumption: only consider **one** preceding instruction
- Which registers may be altered?
  - ➡ Only **eax**, **ecx**, and **edx**
- Not turned out to be problematic (see evaluation)

```
<instruction 1>
…
<instruction n>
ret
```

Example:

```
mov eax, [ecx]
add eax, edx
ret
```

Gadget Example (AND)



AND Gadget

Codebase

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

Machine configuration	# ret instr.	# ret instr. (res)
Native / XP SP2	118,154	22,398
Native / XP SP3	95,809	22,076
VMware / XP SP3	58,933	22,076
VMware / 2003 Server SP2	61,080	23,181
Native / Vista SP1	181,138	30,922
Bootcamp / Vista SP1	177,778	30,922

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

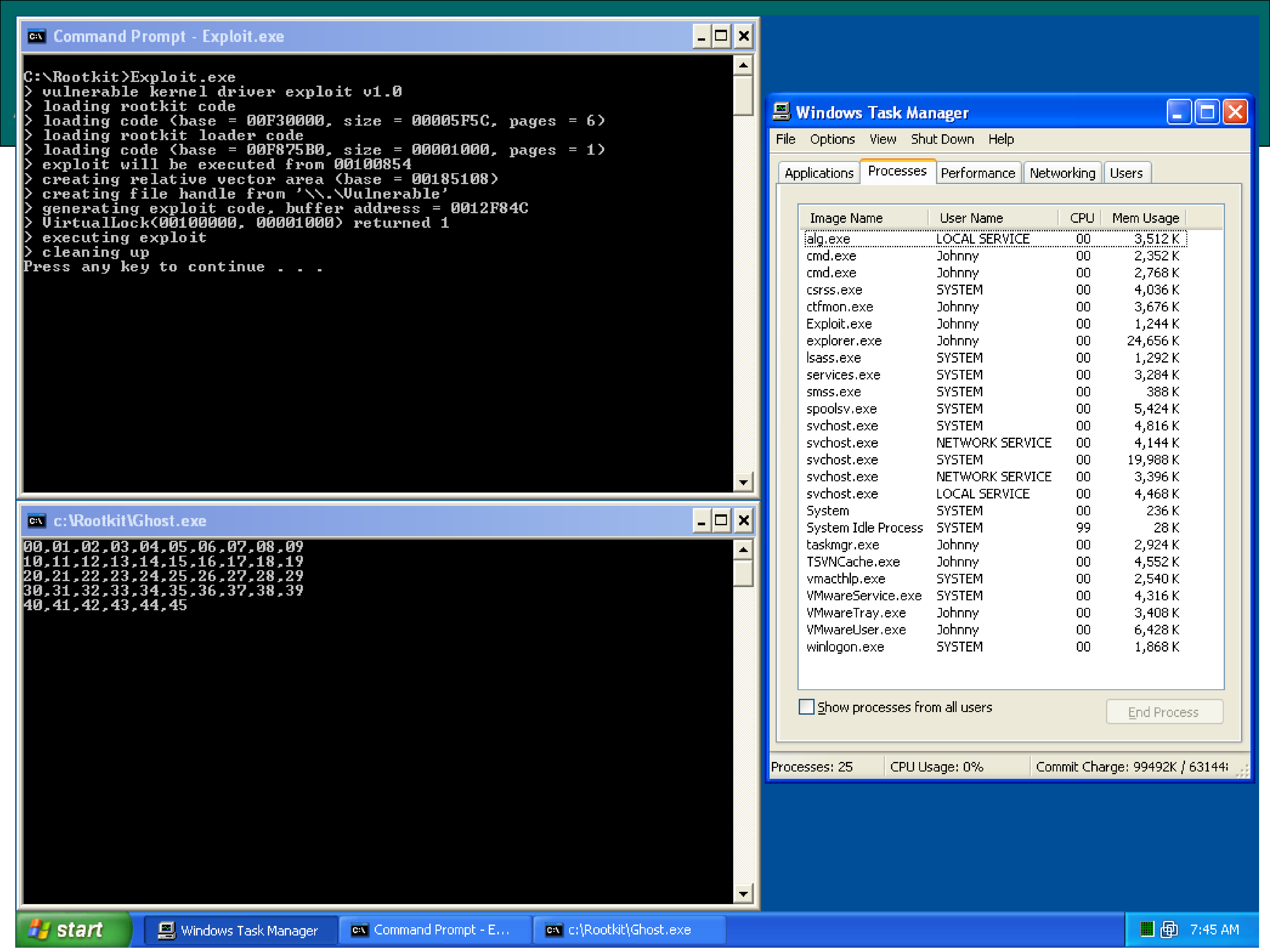
- Motivation
- Automating Return-Oriented Programming
- Evaluation
- **Rootkit Example**
- Conclusion

Rootkit Implementation

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

```
int ProcessName;
int ListStartOffset = &CurrentProcess->process_list.Flink - CurrentProcess;
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 = 00001000, pages = 1)
> exploit will be executed from 00100854
> 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 . . .
c:\Rootkit\Ghost.exe
00,01,02,03,04,05,06,07,08,09
10,11,12,13,14,15,16,17,18,19
20,21,22,23,24,25,26,27,28,29
30,31,32,33,34,35,36,37,38,39
40,41,42,43,44,45
Windows Task Manager
File Options View Shut Down Help
Applications Processes Performance Networking Users
Image Name User Name CPU Mem Usage
alg.exe LOCAL SERVICE 00 3,512 K
cmd.exe Johnny 00 2,352 K
cmd.exe Johnny 00 2,768 K
csrss.exe SYSTEM 00 4,036 K
ctfmon.exe Johnny 00 3,676 K
Exploit.exe Johnny 00 1,244 K
explorer.exe Johnny 00 24,656 K
lsass.exe SYSTEM 00 1,292 K
services.exe SYSTEM 00 3,284 K
smss.exe SYSTEM 00 388 K
spoolsv.exe SYSTEM 00 5,424 K
svchost.exe SYSTEM 00 4,816 K
svchost.exe NETWORK SERVICE 00 4,144 K
svchost.exe SYSTEM 00 19,988 K
svchost.exe NETWORK SERVICE 00 3,396 K
svchost.exe LOCAL SERVICE 00 4,468 K
System SYSTEM 00 236 K
System Idle Process SYSTEM 99 28 K
taskmgr.exe Johnny 00 2,924 K
TSVNCache.exe Johnny 00 4,552 K
vmacthlp.exe SYSTEM 00 2,540 K
VMwareService.exe SYSTEM 00 4,316 K
VMwareTray.exe Johnny 00 3,408 K
VMwareUser.exe Johnny 00 6,428 K
winlogon.exe SYSTEM 00 1,868 K
Show processes from all users
End Process
Processes: 25
CPU Usage: 0%
Commit Charge: 99492K / 63144
start
Windows Task Manager
Command Prompt - E...
c:\Rootkit\Ghost.exe
7:45 AM

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

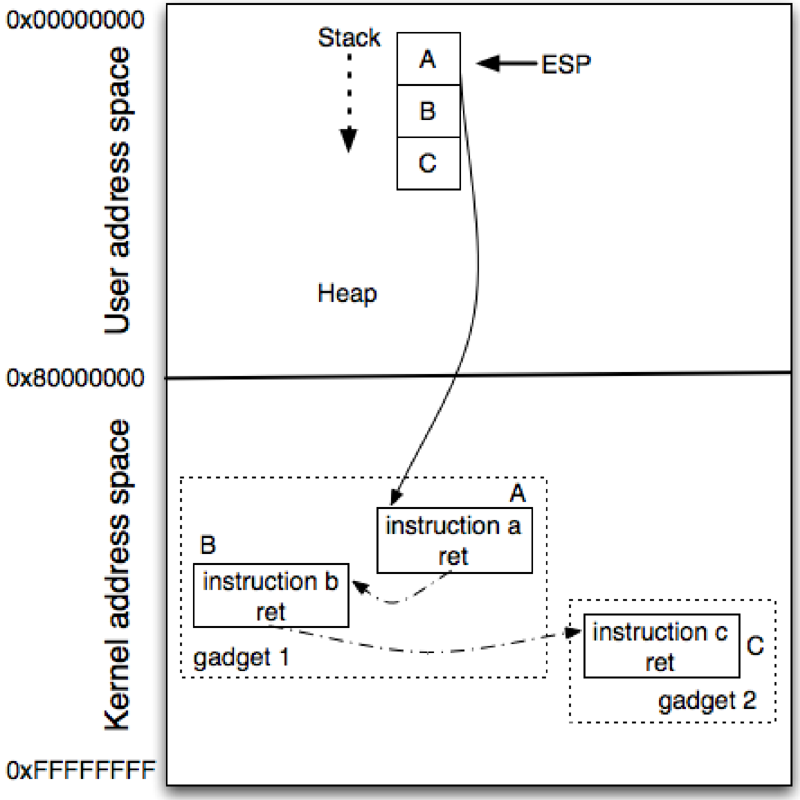
- [RAID08] Riley et al.: Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing
- [ACM07] Seshadri et al.: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes
- [CCS07] Shacham: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function 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

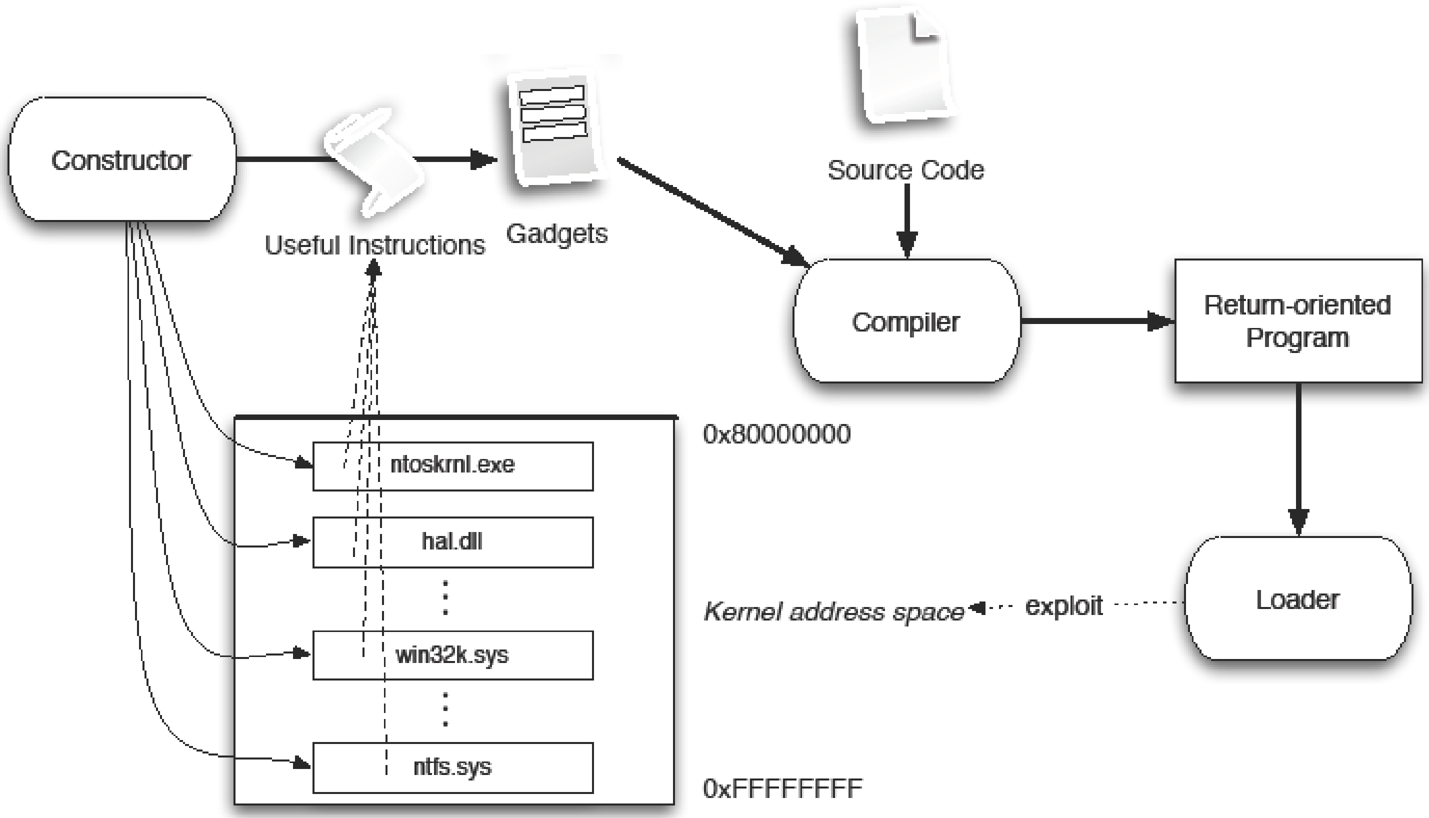
- Allows **hiding** of arbitrary **network socket** connections
- **Hooks** into tcpip.sys **control flow**
- **Concurrency** is the natural **enemy** of return-oriented programming
  - Overcome **synchronization** issues

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

Load constant into register	`pop eax`
Load memory variable	`mov eax, [ecx]`
Store memory variable	`mov [edx], eax`
Perform addition	`add eax, ecx` `add eax, [edx+1337h]`

- **Gradually** construct further lists by combining previous gadgets