Return-oriented Programming

or

The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)

Hovav Shacham, CCS '07
Arms Race: Buffer overflows

- Vulnerabilities
  - Classic: buffer overflow (stack/heap)
  - Integer overflows, format strings, double free, ...

- Countermeasures
  - Canaries on the stack
  - Copy or encrypt jump targets (ret addr, function ptr)
  - Strict Harvard architecture: separate code and data (Non-executable stacks, heaps → W⊕X)
  - Hardware: bit in page table entries preventing code execution from a page (AMD: NX, Intel: XD)
Arms Race II - ret2libc

• “Indirect” code injection
  – (mis)use existing code (libc.so.6, msvcrtd.dll)
  – Return-into-libc: Overwrite stack frame (ret addr)
  – Attacker can call any function with arbitrary arguments (even chaining is possible)
  – But: cannot execute arbitrary code

• Defense
  – Remove all unnecessary functions, e.g. system()
  – ASLR to add noise to jump target addresses
Arms Race III – ROP

- Use short instruction sequences ending in a ret
- Combine them to achieve complexer tasks

- Turing complete
- No function calls, control flow via returns

'In any sufficiently large code base are enough short sequences allowing an attacker to perform arbitrary computations.'
Short Sequences – Gadgets

- Where to get these instruction sequences
  → dense x86 instruction encoding

\[
\begin{align*}
\text{f7 c7 07 00 00 00} & \quad \text{test} \; \$0x00000007, \; \%edi \\
\text{0f 95 45 c3} & \quad \text{setnz} \; \text{-61(}%\text{ebp}) \\
\text{c7 07 00 00 00 0f} & \quad \text{movl} \; \$0x0f000000, \; (\%edi) \\
\text{95} & \quad \text{xchg} \; \%ebp, \; \%eax \\
\text{45} & \quad \text{inc} \; \%ebp \\
\text{c3} & \quad \text{ret}
\end{align*}
\]
Discovery of usable Gadgets

- Search executable segments of libc for ret opcode (c3)
- Scan backwards, disassemble, gather valid entry points into a tree

```
test $0x7,%edi
setnzb -61(%ebp)
```

```
f7 c7 07 00 00 00 0f 95 45 c3
f7 c7 07 00 00 00 0f 95 45 c3
f7 c7 07 00 00 00 0f 95 45 c3
```

```
xchg %ebp, %eax
```

```
inc %ebp
```

```
ret
```

```
c3
```
Recap: Ordinary Execution

• Current instruction is decoded and executes
• eip points after the current instruction to the next one to be executed
• control flow changes value of eip
Recap: Ordinary Execution

- Current instruction is decoded and executes
- eip points after the current instruction to the next one to be executed
- control flow changes value of eip
Recap: Ordinary Execution

- Current instruction is decoded and executes
- eip points after the current instruction to the next one to be executed
- control flow changes value of eip
Recap: Ordinary Execution

- Current instruction is decoded and executes
- eip points after the current instruction to the next one to be executed
- control flow changes value of eip
Recap: Ordinary Execution

- Current instruction is decoded and executes
- eip points after the current instruction to the next one to be executed
- control flow changes value of eip
Return-Oriented Programming

- Assume the next instruction is a `ret`
- The return addresses on the stack are our new “instruction pointer”
- Stack: pointers to code snippets to call + data
- Control transfer via return operations

```
pop %edx; ret  
mov %eax, (%edx); ret  
more code; ret
```

```
data

%esp
```
Return-Oriented Programming

- Assume the next instruction is a `ret`
- The return addresses on the stack are our new "instruction pointer"
- Stack: pointers to code snippets to call + data
- Control transfer via return operations

```
data

%esp

pop %edx; ret
mov %eax, (%edx); ret
more code; ret
```
Return-Oriented Programming

- Assume the next instruction is a `ret`
- The return addresses on the stack are our new “instruction pointer”
- Stack: pointers to code snippets to call + data
- Control transfer via return operations

```assembly
pop %edx; ret
mov %eax, (%edx); ret
more code; ret
```
Return-Oriented Programming

- Assume the next instruction is a `ret`
- The return addresses on the stack are our new “instruction pointer”
- Stack: pointers to code snippets to call + data
- Control transfer via return operations

```
pop %edx; ret
mov %eax, (%edx); ret
more code; ret
```
Return-Oriented Programming

- Assume the next instruction is a `ret`
- The return addresses on the stack are our new “instruction pointer”
- Stack: pointers to code snippets to call + data
- Control transfer via return operations
Return-Oriented Programming

- Assume the next instruction is a ret
- The return addresses on the stack are our new "instruction pointer"
- Stack: pointers to code snippets to call + data
- Control transfer via return operations
Gadgets – Building Blocks

- Load/Store operations (reg/mem)
- ALU (arithm./logic)
- Conditional jumps
- System calls
- Shellcode

q.e.d.
Load a Constant

%esp

0xDEADBEEF

pop %edx
ret
Load from Memory

movl 64(%eax), %eax
ret
pop %eax
ret
Store to Memory

- `movl %eax, 24(%edx)`
- `ret`
- `pop %edx`
- `ret`
Add Value from Memory to eax

- push %edi
- ret
- pop %edx, ret
- ret
- addl (%edx), %eax
- ret

%esp

0xdeadbeef
XOR

- Ideally: xorl (%edx), %eax (but not available)
- Code snippet found:

```assembly
xorl %al, 0x48908c0(%ebx)
and $0xff, %al
push %ebp
or $0xc9, %al
ret
```

- xorl: rotate 4 times and xor byte wise
- push %ebp; ret → let %ebp point to a ret
Control Flow – Jumps

- Unconditional jumps → pop %esp, ret
- Conditional jumps
  - Test condition → %eflags hold result
  - Jumps change instruction pointer, we need to conditionally change the stack pointer
  - Three steps
    - Test condition
    - Copy flags to general purpose register (add-with-carry)
    - Mask desired bit, change esp accordingly
Control Flow – Jumps

- Test condition → flags change
  - Copy flags to general purpose register
    → clear ecx; adc %cl, %cl → 0 or 1
- Modify esp depending on ecx
  - negl ecx → 0 or 0xFFFFFFFF
  - andl with relative jump distance
  - add to esp
Conditional Jumps: Step 2

```
0x00000000
movl %ecx, (%edx)
adc %cl, %cl
ret
```

```
(CF goes here)
```

```
%(esp)
popl %ecx
popp %edx
ret
```
System Calls

- Typical syscall wrapper:
  - Load arguments to registers
  - Invoke kernel (sysenter/int 80)
  - Check for error, cleanup

```
89 da          mov %ebx, %edx
8b 5c 24 04    movl 4(%esp), %ebx
b8 3c 00 00 00 mov $0x0000003c, %eax
65 ff 10 00 00 00 lcall %gs:0x10(,%0)
89 d3          mov %edx, %ebx
83 c3          ret
```

wrapper for sysenter or int 0x80
No-argument Syscall

```
%esp
(call index)
pop %esp
ret
pop %eax
ret
lcall %gs:0x10(0)
ret
```
Shellcode

- Set %eax (syscall number) to 0xb
- Let %ebx point to the string (“/bin/sh”)
- Load %ecx to point to argv vector
- Clear %edx (pointer to environment)
- Trap into kernel

Game over
c3 / ret Statistics

- 5,483 out of 975,626 bytes are 0xc3
- 3,429 are actually ret instructions
- 1,692 are `add imm32, %ebx` (81 c3 imm32)
- 290 are displacements (relative calls and jumps or data offsets like `movb %al, -61(%ebp)` encoded as 88 45 c3)
- Constants, SIB byte, floating point operation, ...
Avoid Spurious ret?

• Get rid of unintended c3 bytes (compiler)
  – No moves from %eax to %ebx, no add %reg, %ebx
  – Avoid offsets by c3 (rearrange instructions & data)

• But: c3 is only one type of return (near return)
  – There are four: Far ret, near ret with stack unwinding, far ret with stack unwinding
  – Much harder to use (far returns also pop the code segment selector, but possible)

• Register spring: jmp %reg if reg points to a ret

• Even better: jmp *%reg, indirect jump
Address Space Layout Randomization

- Add random to placement of movable sections
  - Stack
  - Heap
  - Libraries
- Attacker has to “guess” addresses of return opcodes (c3)
- Most probably program crashes if guess was wrong
- Amount of randomness influences search space
Address Space Layout Randomization

First run

- Offsets still need to be page aligned
- Text segment cannot be moved
- Addresses must not leak in any way

Second run

- libc
- ld
- stack
- heap
- text
ASLR – Efficiency

- On The Effectiveness of Address-Space Randomization (CSS'04)
- 32bit architectures → number of bits to randomize is limited
- PaX
  - Executable: Code, Data, BSS → 16 bit random
  - Mapped: heap, libs, thread stack, … → 16 bit
  - Stack: normal user stack → 24 bit random offset
ASLR – Efficiency

• Fork vs. Exec()
  – Fork: the address space layout does not change
  – Exec: new layout every time the program runs

• Example: Web server spawns (forks) new processes for every client request
  – A return-to-libc attack needs to guess 16 bit, the offset of the libc text segment
  – Mount a return-2-libc attack to call usleep()
    • Guessed offset wrong → child crashed, connection resets
    • Right offset → connection hangs for some seconds
    • Run second stage attack with de-randomized addresses
ASLR – Efficiency

- Apache web server with added buffer overflow
- Infer delta_mmap through brute force (usleep)
- After 216 seconds on average successful → start second stage attack, circumventing ALSR
- Conclusion: on 32bit architectures and only 16 bit random for mapped area serious problems
- But: 64bit is there, at least 40 bit usable for address space randomization, practically brute forcing no more thread
“Return-less” Kernels

- Defeating Return-Oriented Rootkits With “Return-Less” Kernels (EuroSys 2010)
- Get rid of all return instructions in the kernel → no returns, no return-oriented exploits
- Idea: return indirection
  - Caller puts return address into a global table and pushes the index onto the stack
  - Callee on return does a table lookup and jumps to the intended instruction after the call
  - No more “normal” call/ret instructions
“Return-less” Kernels

- Remove all remaining c3 bytes from the code
- Relative offsets in jumps → add nops
  `jmp ffc3` → `nop; jmp ffc2`
- Relative offsets to data → adapt register
  `mov %eax, 0xc3(%ebx)` →
  `dec %ebx; mov %eax, 0xc4(%ebx)`
- Peephole optimization to replace instructions by equivalent ones without c3 (ret)
Normal Call-Return Pairing

- **C1:** push
- **C2:** control flow transfer
- **R2:** control flow transfer
- **R1:** pop

Diagram shows the interaction between caller and callee, with stack and return address being pushed and popped during the call and return processes.
Return Indirection

C1: push return index

R1: pop

C2: control flow transfer

R2: locate return address

R3: control flow transfer
Summary

- Buffer overflows, format string attacks, double frees, ... many ways to penetrate a system
- Counter code injection with W⊕X: never execute writable code segments
- Return-oriented programming: use existing code snippets (gadgets)
- Sufficiently large code bases to take gadgets from → turing completeness
- ALSR with 64bit seems to solve this (for now)
In A Nutshell

- Code fractions from sufficiently large code base yield Turing completeness
- Avoiding return opcodes won't help
- No code injection at all, thus circumventing DEP

Questions:
- Ret2libc with ASLR: guess one address
- ROP with ASLR: guess many return targets
- ASLR with 48/64 bit → problem postponed?