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

Hovav Shacham, CCS07

Presented by Jaehwang Jung

Return-Oriented Programming

Computer Systems: A Programmer's Perspective

Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University

Title

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

Background: Attacks & Defenses

Attackers

1. Hijack the control flow
    - Stack smashing (stack buffer overflow): overwrite the return address → return to somewhere else
2. Run something interesting
    - Code injection
        - "shellcode": `execve("/bin/sh", …, …)`

Defenders

1. Prevent hijacking the control flow
    - buffer bound checking → performance overhead
    - stack canary → limited
2. Prevent running something interesting
    - Non-executable memory (e.g. W^X: Write xor eXecute)

Return-into-libc Attack

W^X only works for newly injected code.

→ Make use of already existing code! (e.g. libc), which must be executable.

→ Carefully smash the stack to call libc functions

Limitations of Return-into-libc: Less freedom

- Only able to call some functions sequentially
- Some functions may not be available
  - If the program doesn't use a function, compiler can simply remove it from the binary

→ Relying on **calling existing functions** is inflexible.

Innocent Flesh on the Bone

Key Idea 1: x86 is weird

- extremely dense ISA
- variable length

It's likely that a random byte stream can be interpreted as a sequence of valid instructions!

Re-interpretation of binary stream

f7 c7 07 00 00 00	test $0x00000007, %edi
0f 95 45 c3	setnzb -61(%ebp)

c7 07 00 00 00 0f	movl $0x0f000000, (%edi)
95	xchg %ebp, %eax
45	inc %ebp
c3	ret

Now what?

We generated a pool of instruction sequences that does something.

But each individual piece of sequences isn't that meaningful.

To utilized them, we need a way to combine them.

(or, *program* with them)

Key Idea 2: `ret` as glue for instruction sequences

- 1 byte (0xc3): very abundant
- programmable: set the return address stored in the stack (pointed to by %esp)

Approach

1. Find valid instruction sequences that end with `ret` (called *useful* sequences)
2. make chain of such sequences that does something interesting.

Quick Recap of x86 stack

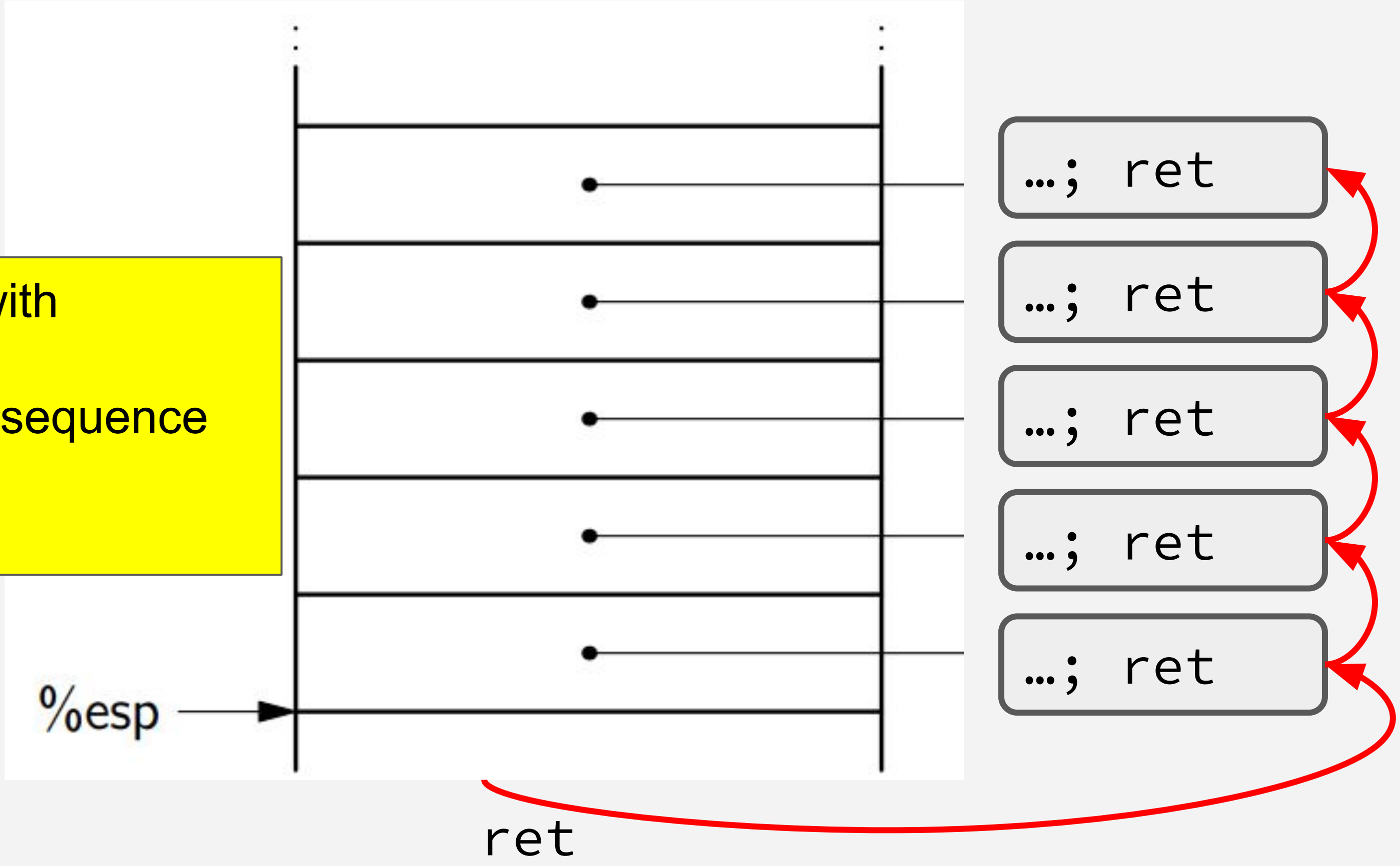
`ret`:
jump to the return address;
increment %esp (up)

Return-Oriented Programming

Return-Oriented Programming Framework

Overwrite the stack with sequence of *gadgets*

- pointer to useful sequence
- input data
- …



Loading a Constant

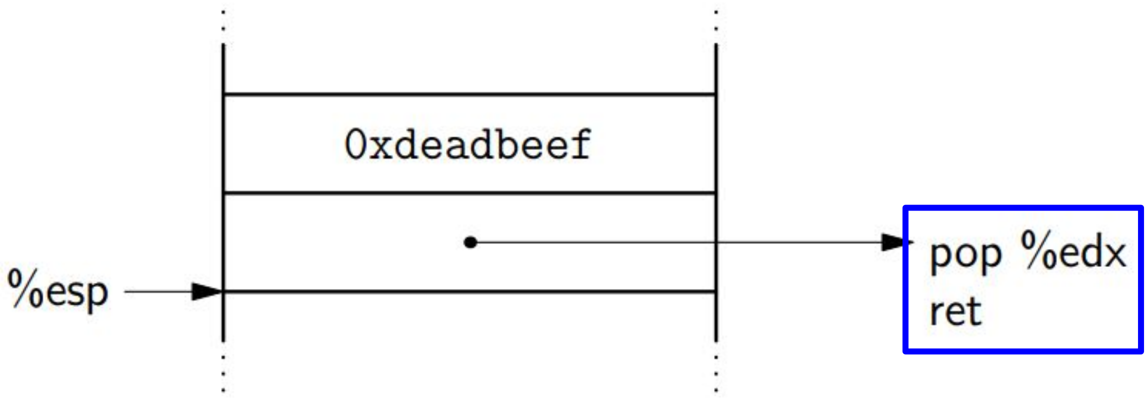


Figure 2: Load the constant 0xdeadbeef into %edx.

Variant: Infinite Loop

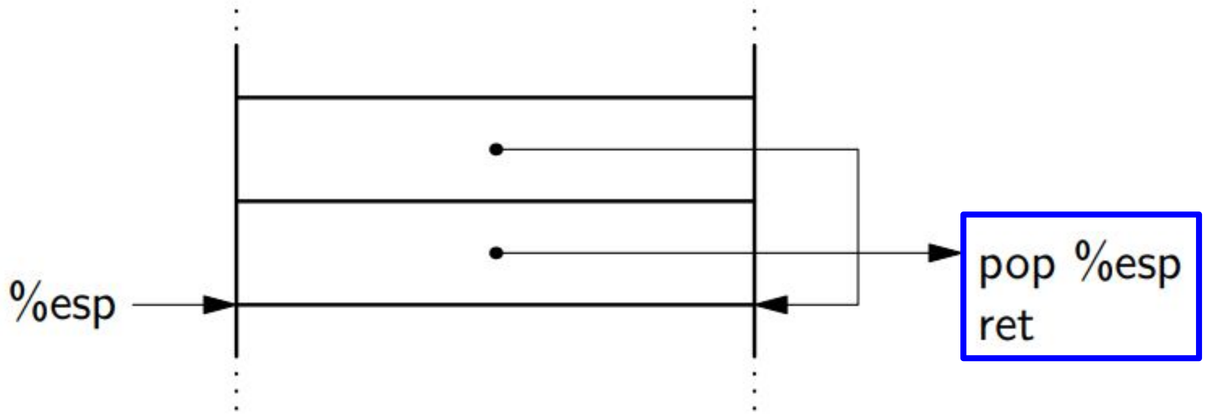


Figure 10: An infinite loop by means of an unconditional jump.

Loading from Memory

Need to undo the effect of +64

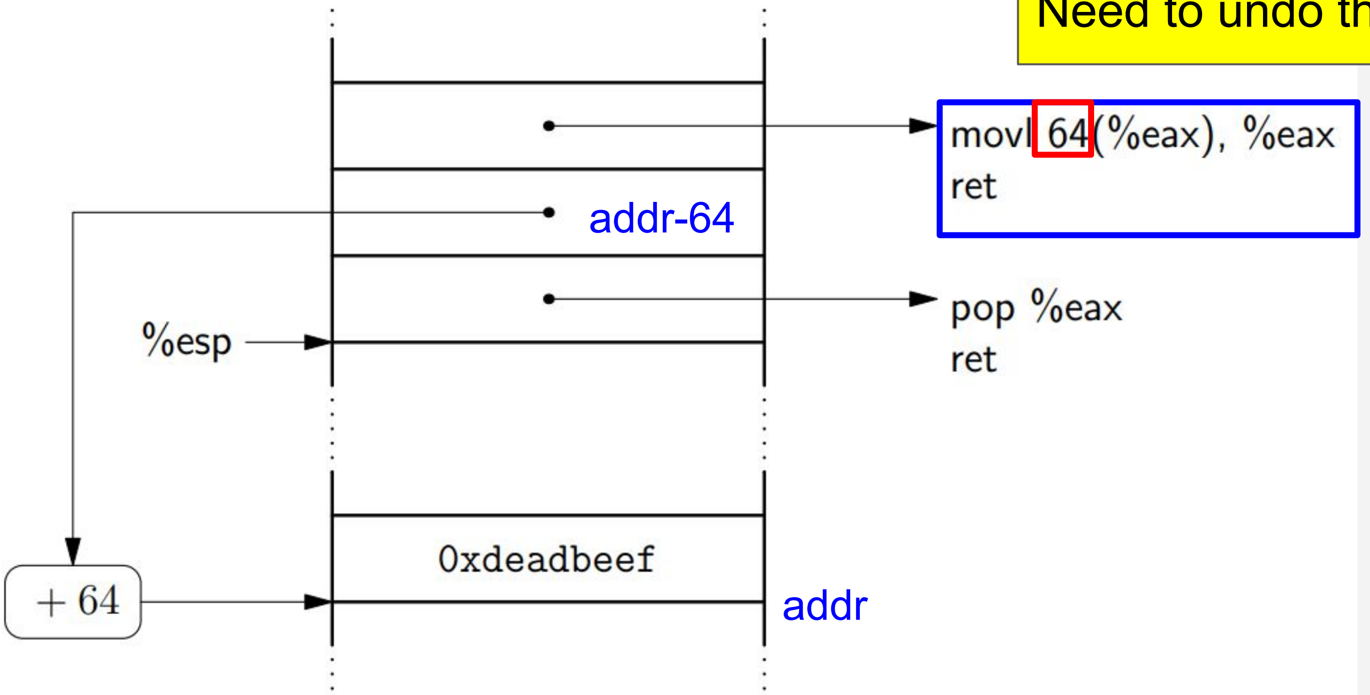


Figure 3: Load a word in memory into %eax.

Addition

tricky: pushed value is used for ret

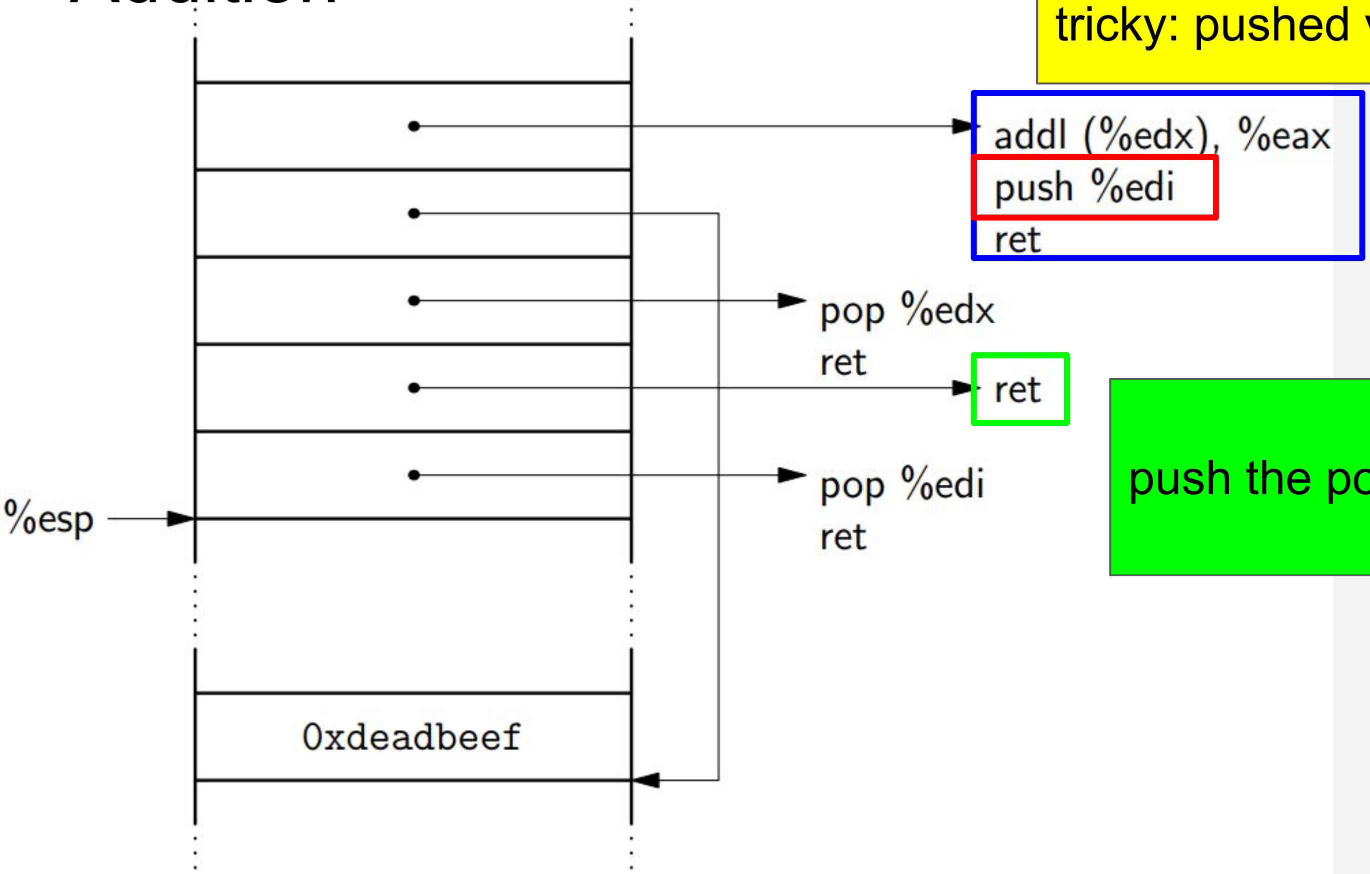


Figure 5: Simple add into %eax.

push the pointer to noop (= ret)

Conditional Jump: Difficult

- Conditional jump instructions of x86 are not viable!
  - ROP "program" is written in stack.
  - Overall control flow is governed by %esp, not %eip.
- Approach: Conditionally modify %esp.

Conditional Jump

1. Set/Unset flag
   - e.g. Carry flag (CF)
2. Transfer the flag to general purpose register and isolate the flag
   - If CF=1, then put 1 in %esi (vice versa)
   - then, apply `neg`
     - 0 → neg → 000...00000
     - 1 → neg → 111...111111
3. Perturb %esp by the desired jump mount, if the flag is set
   - apply `and` to the value from 2 and jump amount
   - `add` this result to %esp

System call → Shellcode

It's very likely that libc's syscall wrappers are available.

(Most programs do syscalls.)

1. Set the syscall index and arguments
   - 0xb for syscall index (`execve`)
   - `"/bin/sh"` as argument
2. Return into the middle of a syscall wrapper
   - `lcall %gs:0x10(,0); ret`
3. Do something interesting with the shell
   - ~~seize the means of production~~

Defense

Avoiding spurious `rets`

Modify the compiler to avoid generating unintended useful sequences.

Limitations:

- Generated code might be inefficient.
- It's effectively impossible to eliminate them completely.
- It's possible to use other instructions for glue.
  - e.g. `jmp %edx` where `%edx` points to (intended) `ret`
- It's possible to find some useful sequences in the other parts of the binary
  - e.g. ELF header

Wrap-Up

Summary

ROP is a powerful exploit that bypasses various countermeasures by utilizing the existing code in an unexpected way.

Stuff I didn't cover in this presentation

- bunch of other gadgets
- coping with nul byte while smashing the stack
- algorithm for finding useful sequences
- statistics of useful sequences

Follow-up works

- Applying ROP to other architectures
  - When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC, E. Buchanan el al. 2008
- Defense
  - Apply address space layout randomization (ASLR) globally
  - (Advanced compiler-based method) G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries, Kaan Onarlioglu et al., 2010
- Variants
  - Jump-Oriented Programming: A New Class of Code-Reuse Attack, T. Bletsch et al, 2011
  - (Blind ROP) Hacking Blind, Andrea Bittau et al, 2014
- Automation: basically a program synthesis problem

References

- All figures that are not mine are from the full version of the paper https://hovav.net/ucsd/dist/geometry.pdf
- http://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf
    - Real world example CVE-2011-1938