0x1A Great Papers in Computer Security

Vitaly Shmatikov

http://www.cs.utexas.edu/~shmat/courses/cs380s/

H. Shacham

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

(CCS 2007)

Buffer Overflow: Causes and Cures

- Typical memory exploit involves code injection
  - Put malicious code in a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it
    - Overwrite saved EIP, function callback pointer, etc.
- Defense: prevent execution of untrusted code
  - Make stack and other data areas non-executable
    - Note: messes up useful functionality (e.g., ActionScript)
  - Digitally sign all code
  - Ensure that all control transfers are into a trusted, approved code image

W⊕X / DEP

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's DEP - Data Execution Prevention
  - This blocks most (not all) code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - OS can make a memory page non-executable
- Widely deployed
  - Windows (since XP SP2), Linux (via PaX patches), OpenBSD, OS X (since 10.5)

What Does W⊕X Not Prevent?

- Can still corrupt stack …
  - … or function pointers or critical data on the heap, but that's not important right now
- As long as "saved EIP" points into existing code, W⊕X protection will not block control transfer
- This is the basis of return-to-libc exploits
  - Overwrite saved EIP with address of any library routine, arrange memory to look like arguments
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code
  - … especially if system() is not available

return-to-libc on Steroids

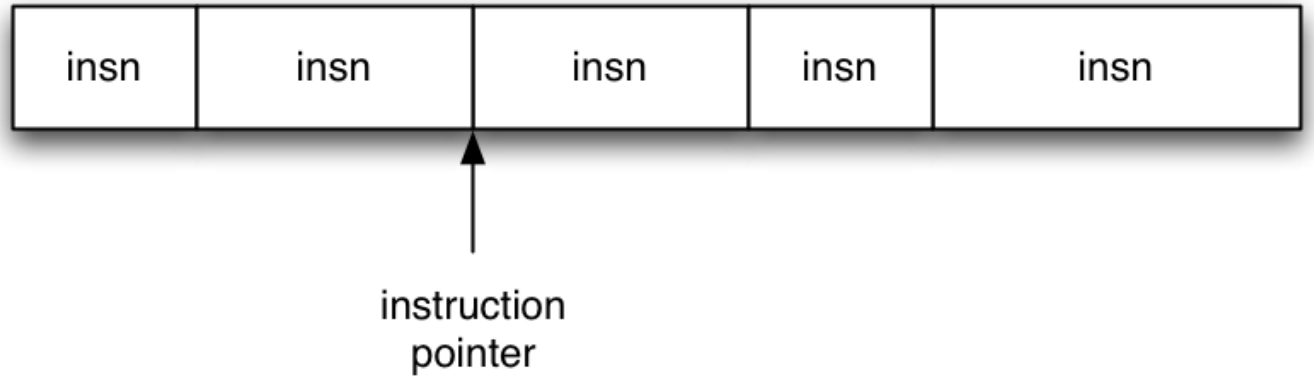
- Overwritten saved EIP need not point to the beginning of a library routine
- Any existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read the word pointed to by stack pointer (ESP)
    - Guess what? Its value is under attacker's control! (why?)
  - Use it as the new value for EIP
    - Now control is transferred to an address of attacker's choice!
  - Increment ESP to point to the next word on the stack

Chaining RETs for Fun and Profit

[Shacham et al]

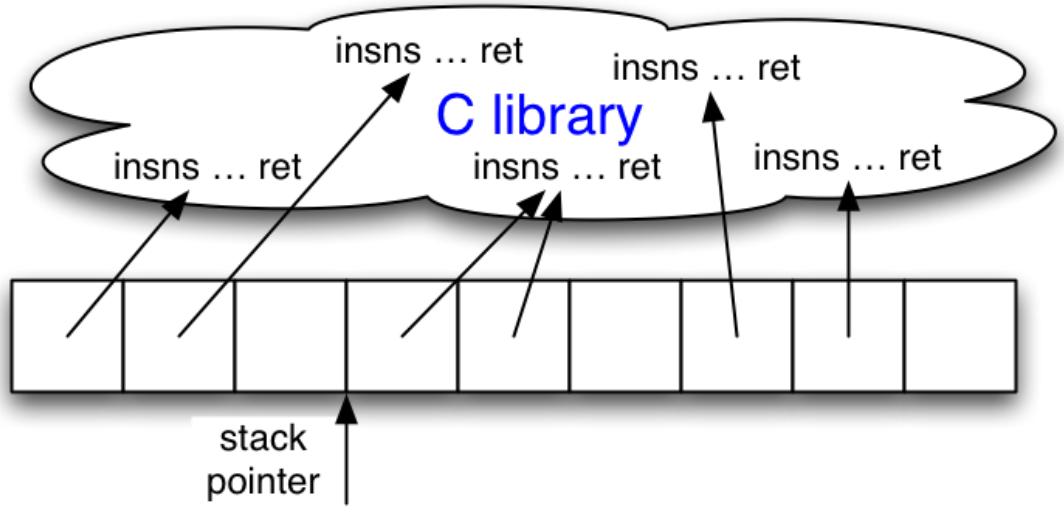
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation using no injected code at all!

Ordinary Programming



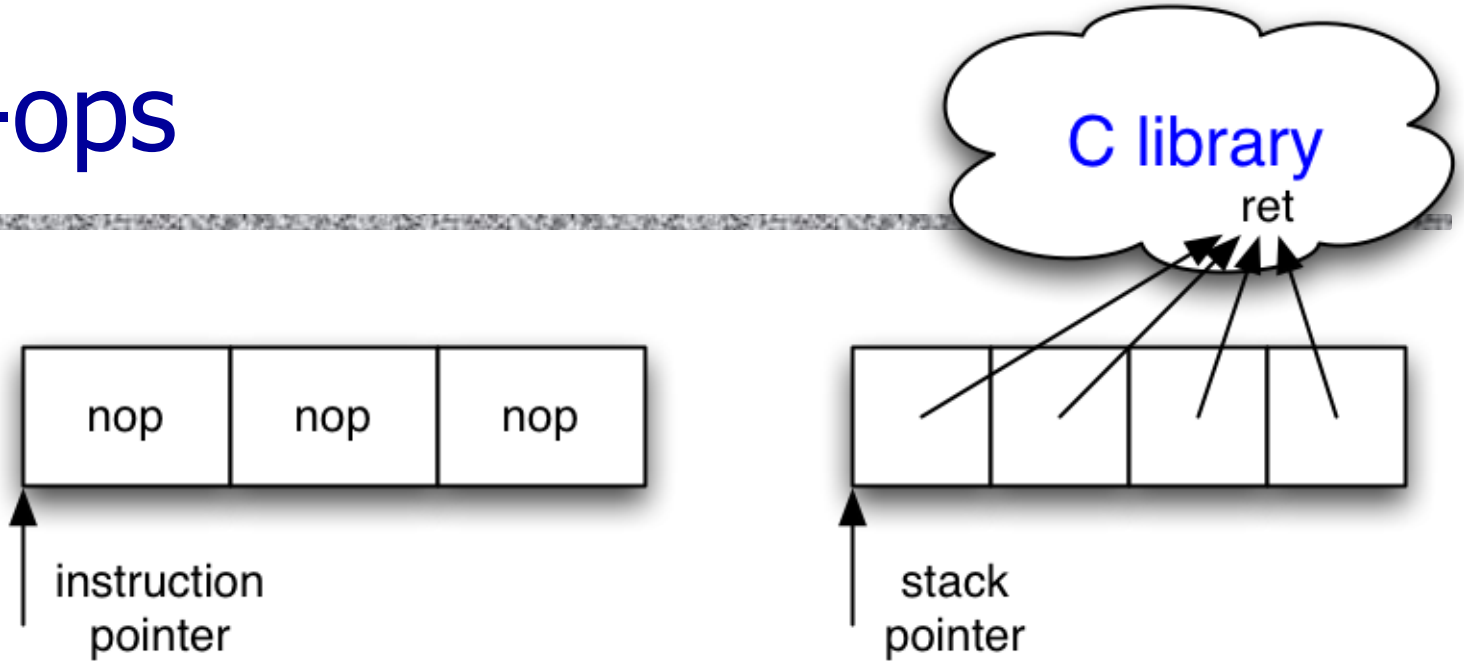
- Instruction pointer (EIP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

Return-Oriented Programming



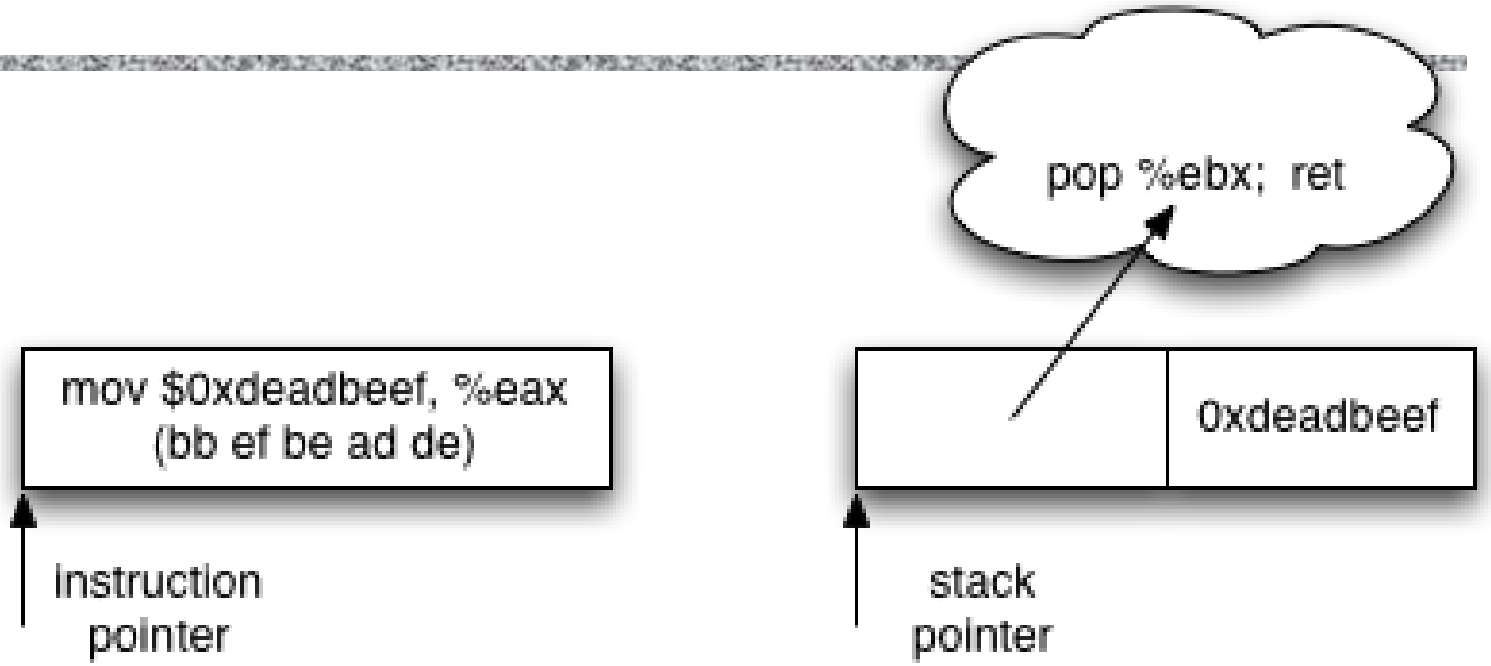
- Stack pointer (ESP) determines which instruction sequence to fetch and execute
- Processor doesn't automatically increment ESP
  - But the RET at end of each instruction sequence does

No-ops



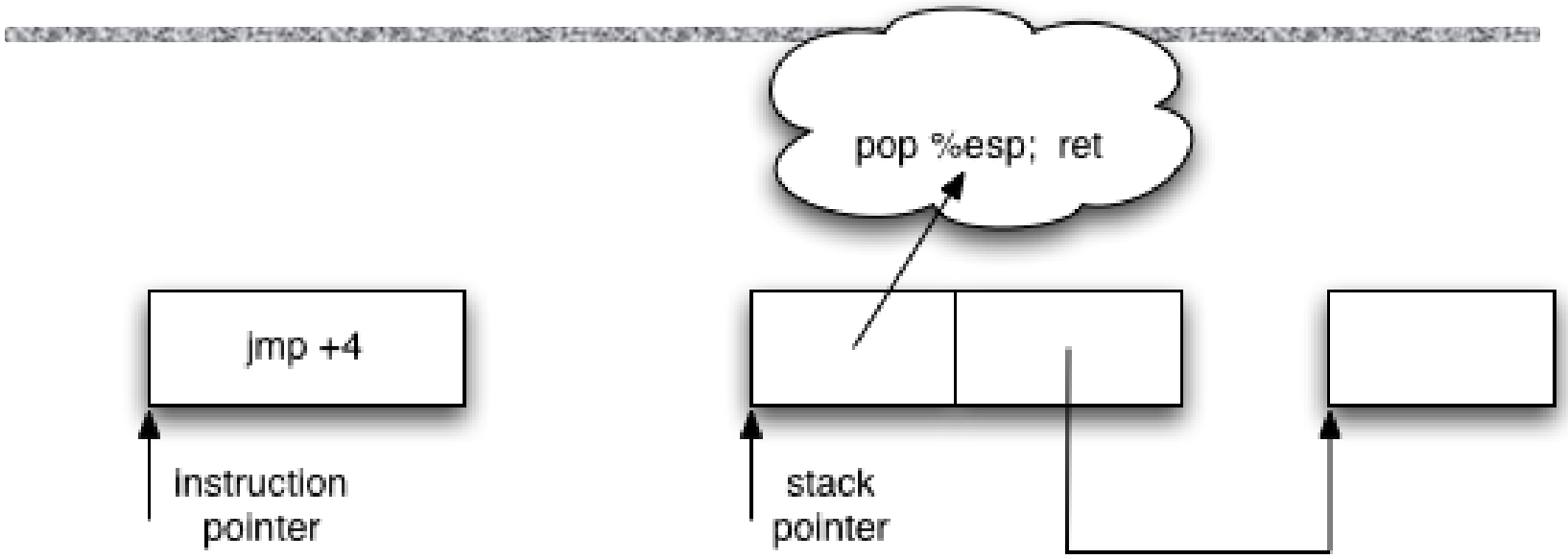
- No-op instruction does nothing but advance EIP
- Return-oriented equivalent
  - Point to return instruction
  - Advances ESP
- Useful in a NOP sled (what's that?)

Immediate Constants



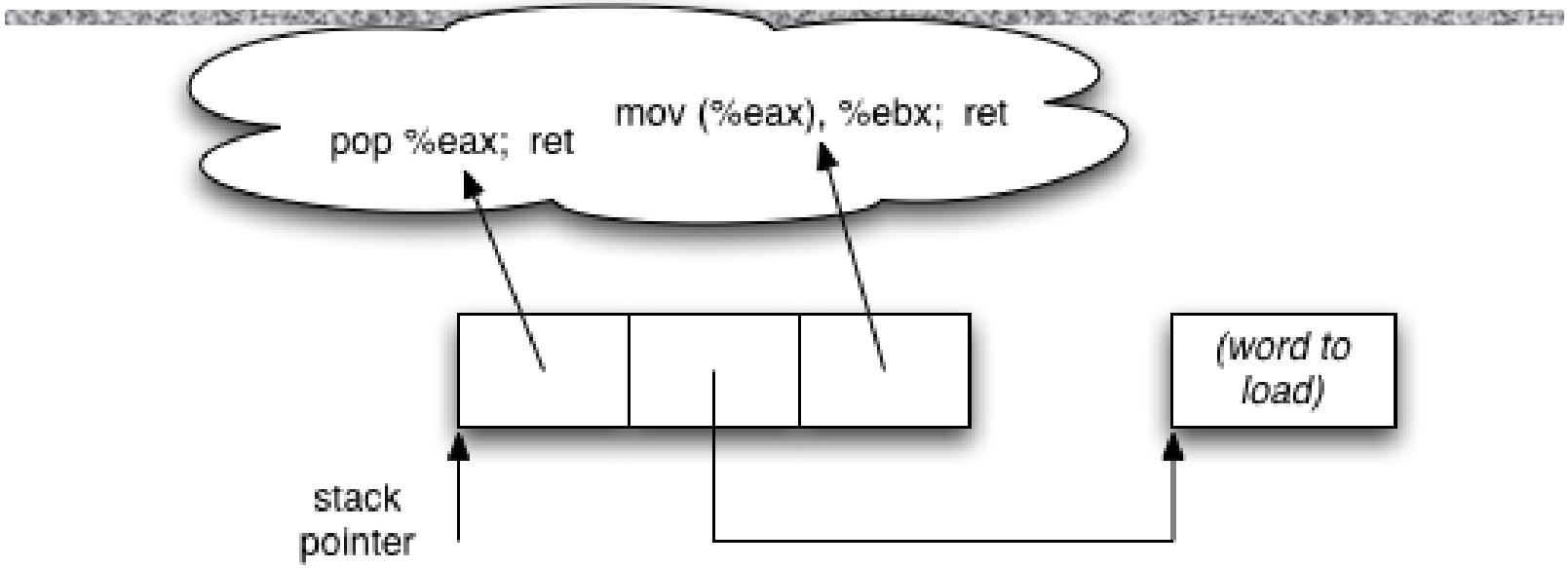
- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

Control Flow



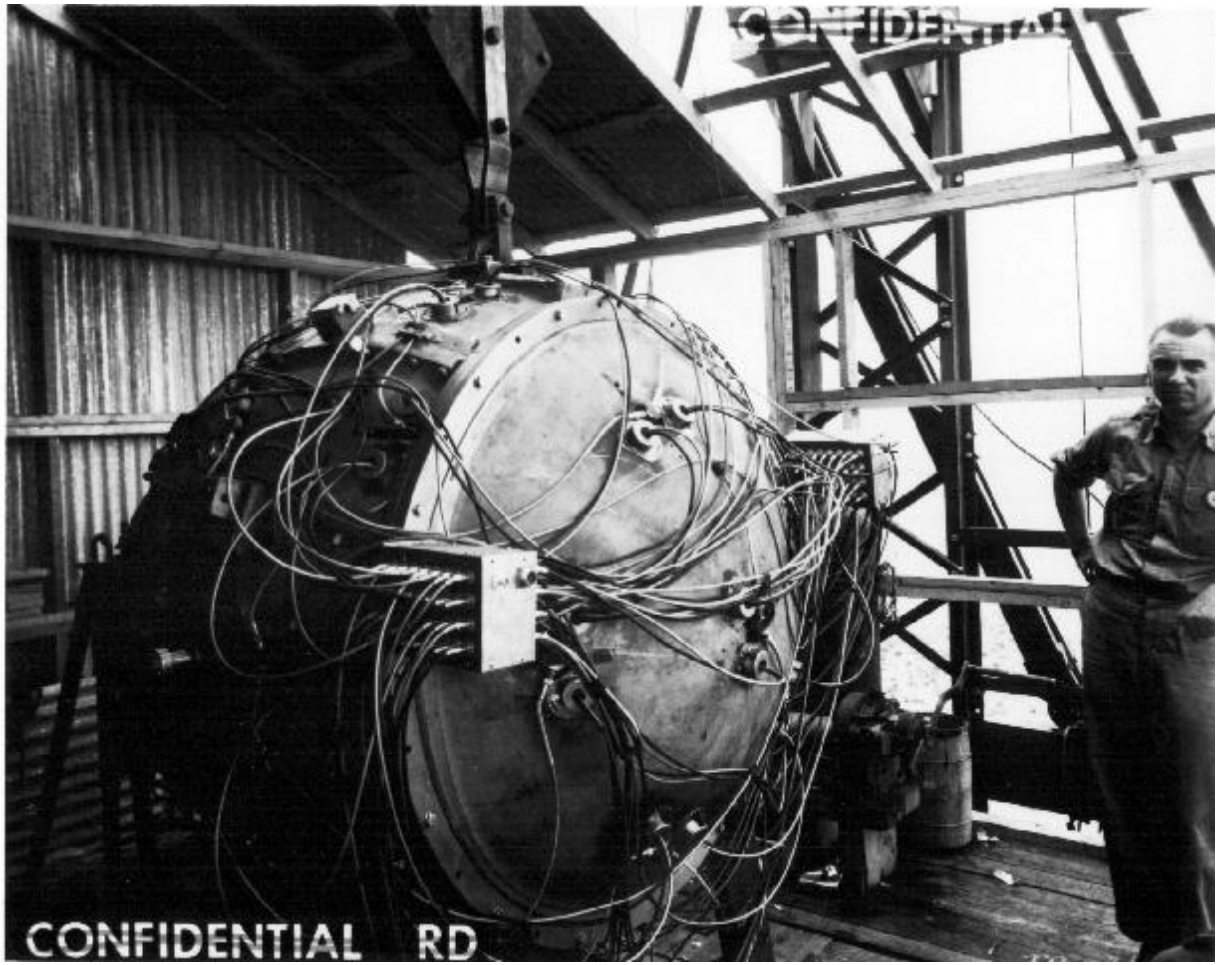
- Ordinary programming
  - (Conditionally) set EIP to new value
- Return-oriented equivalent
  - (Conditionally) set ESP to new value

Gadgets: Multi-instruction Sequences



- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register
  - Load address of source word into EAX
  - Load memory at (EAX) into EBX

“The Gadget”: July 1945



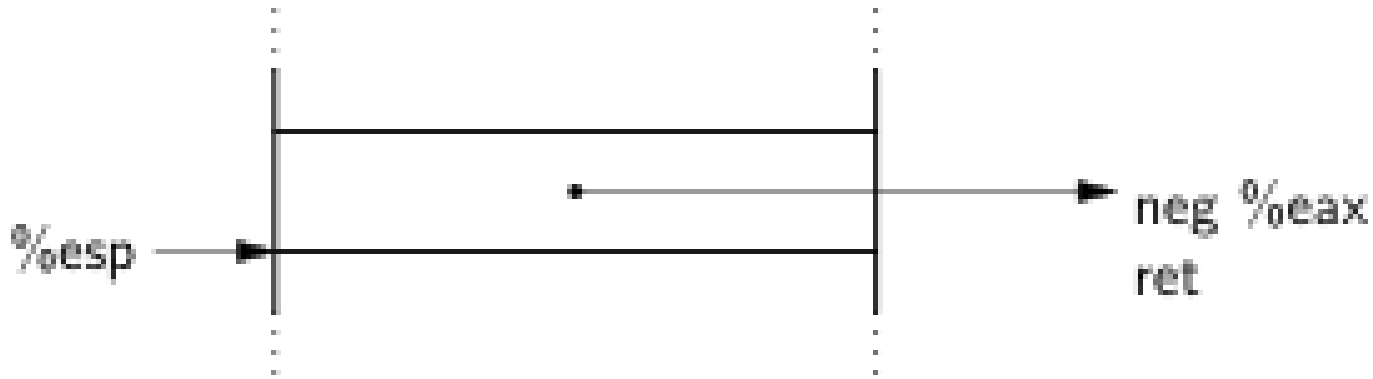
Gadget Design

- Testbed: libc-2.3.5.so, Fedora Core 4
- Gadgets built from found code sequences:
  - Load-store, arithmetic & logic, control flow, syscalls
- Found code sequences are challenging to use!
  - Short; perform a small unit of work
  - No standard function prologue/epilogue
  - Haphazard interface, not an ABI
  - Some convenient instructions not always available

Conditional Jumps

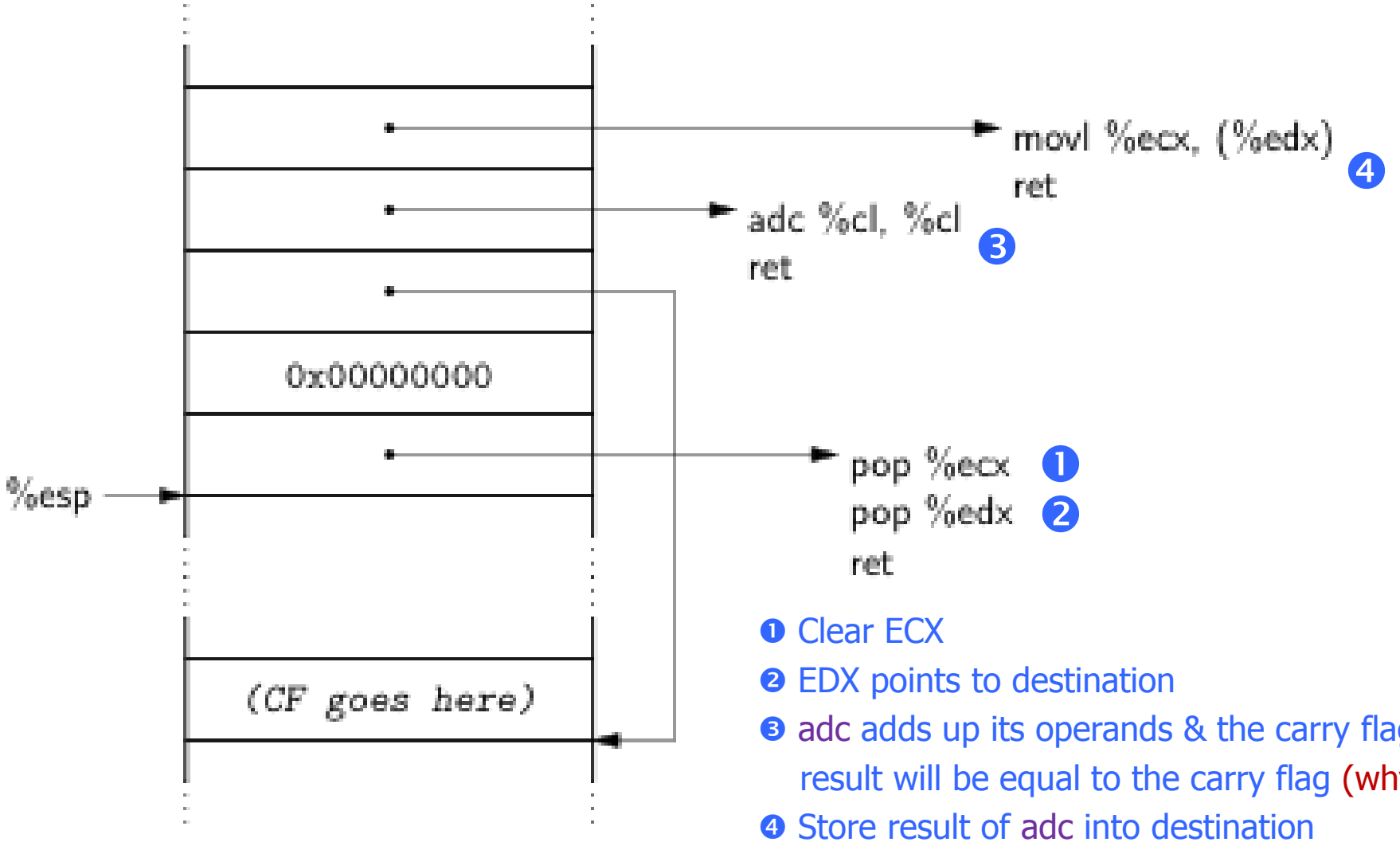
- cmp compares operands and sets a number of flags in the EFLAGS register
  - Luckily, many other ops set EFLAGS as a side effect
- jcc jumps when flags satisfy certain conditions
  - But this causes a change in EIP… not useful (why?)
- Need conditional change in stack pointer (ESP)
- Strategy:
  - Move flags to general-purpose register
  - Compute either delta (if flag is 1) or 0 (if flag is 0)
  - Perturb ESP by the computed delta

Phase 1: Perform Comparison



- neg calculates two's complement
  - As a side effect, sets carry flag (CF) if the argument is nonzero
- Use this to test for equality
- sub is similar, use to test if one number is greater than another

Phase 2: Store 1-or-0 to Memory



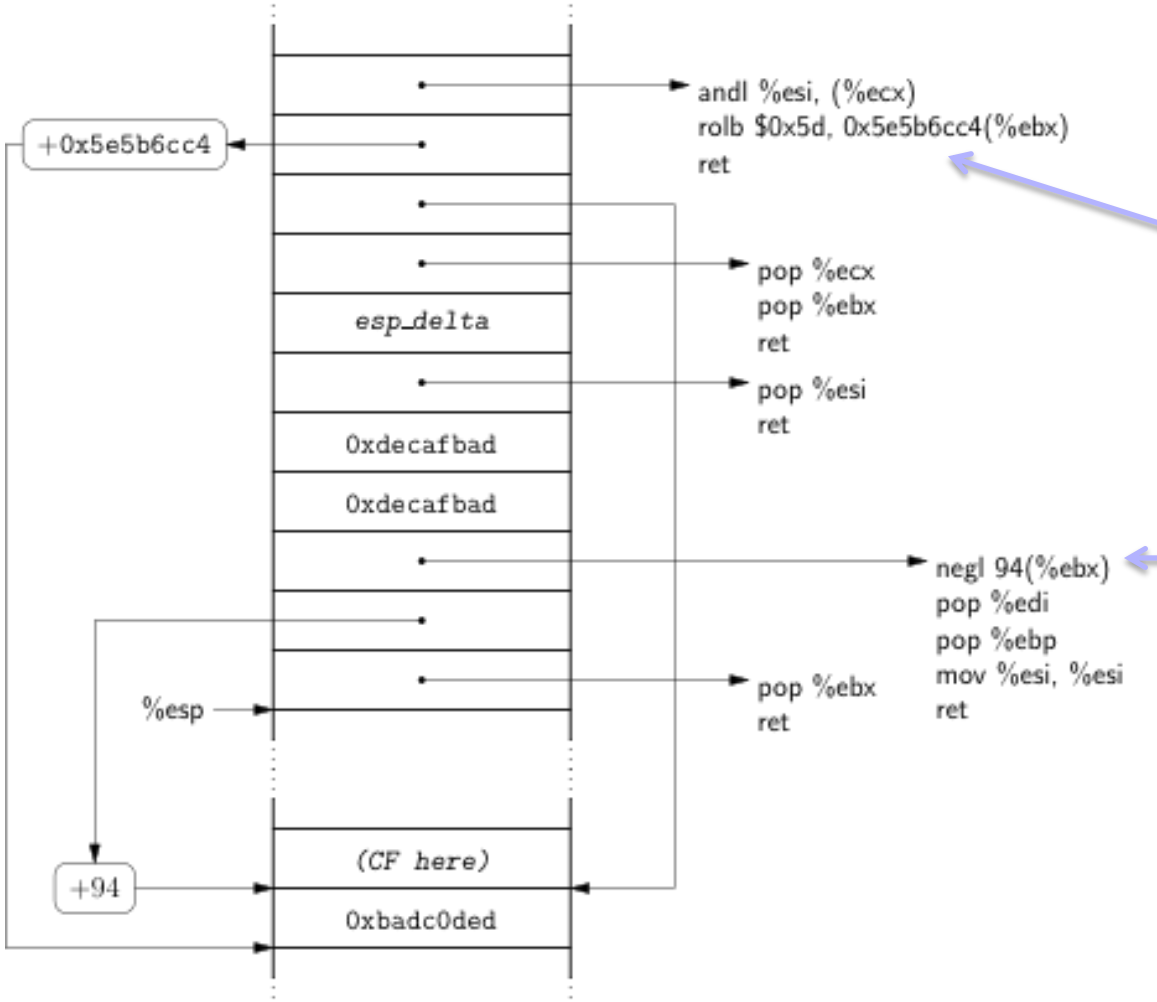
❶ Clear ECX

❷ EDX points to destination

❸ adc adds up its operands & the carry flag; result will be equal to the carry flag (why?)

❹ Store result of adc into destination

Phase 3: Compute Delta-or-Zero



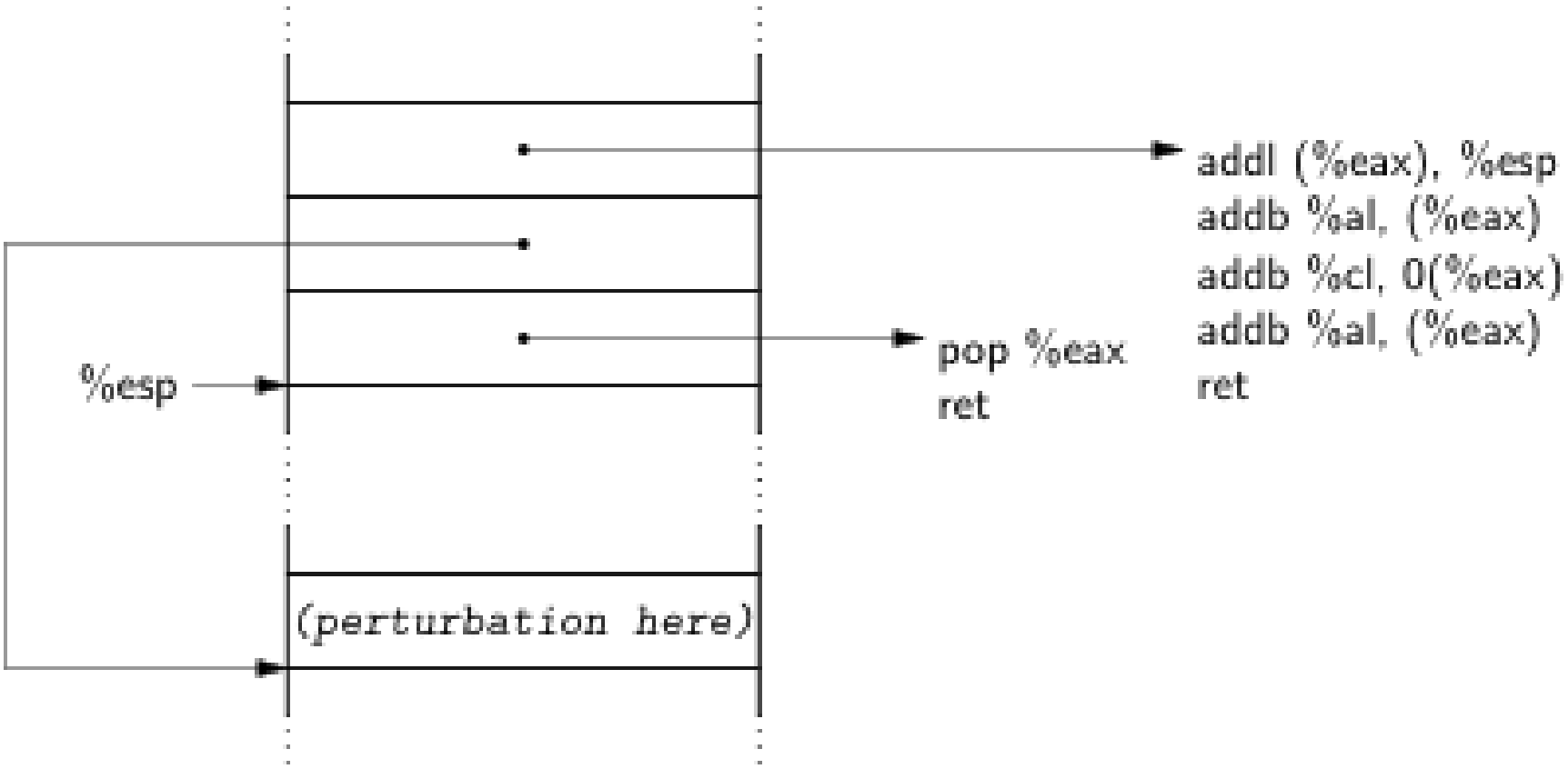
Bitwise AND with delta (in ESI)

Two's-complement negation:

0 becomes 0…0;

1 becomes 1…1

Phase 4: Perturb ESP by Delta

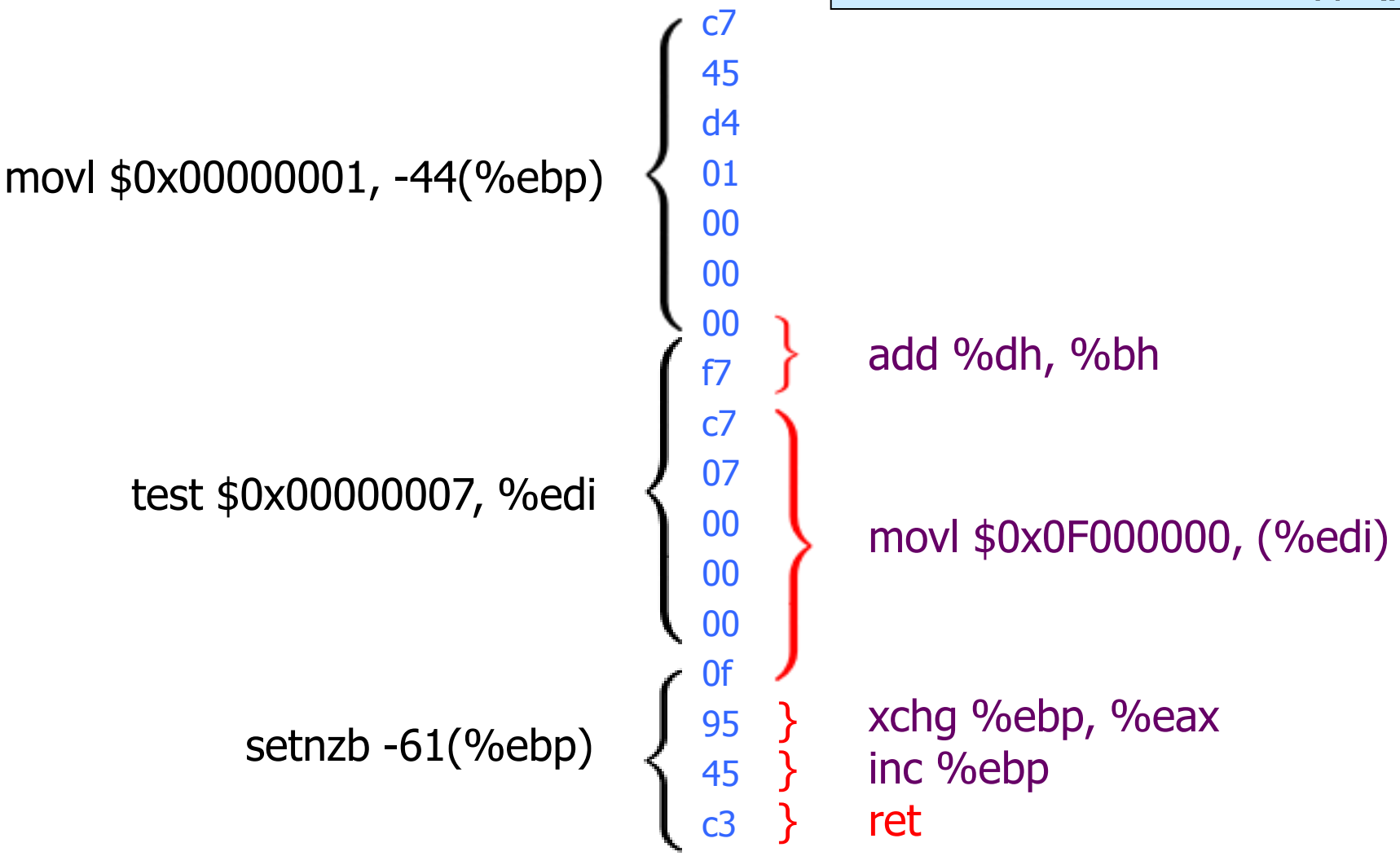


Finding Instruction Sequences

- Any instruction sequence ending in RET is useful
- Algorithmic problem: recover all sequences of valid instructions from libc that end in a RET
- At each RET (C3 byte), look back:
  - Are preceding i bytes a valid instruction?
  - Recur from found instructions
- Collect found instruction sequences in a trie

Unintended Instructions

Actual code from ecb_crypt()



x86 Architecture Helps

- Register-memory machine
  - Plentiful opportunities for accessing memory
- Register-starved
  - Multiple sequences likely to operate on same register
- Instructions are variable-length, unaligned
  - More instruction sequences exist in libc
  - Instruction types not issued by compiler may be available
- Unstructured call/ret ABI
  - Any sequence ending in a return is useful

SPARC: The Un-x86

- Load-store RISC machine
  - Only a few special instructions access memory
- Register-rich
  - 128 registers; 32 available to any given function
- All instructions 32 bits long; alignment enforced
  - No unintended instructions
- Highly structured calling convention
  - Register windows
  - Stack frames have specific format

ROP on SPARC

- Use instruction sequences that are suffixes of real functions
- Dataflow within a gadget
  - Structured dataflow to dovetail with calling convention
- Dataflow between gadgets
  - Each gadget is memory-memory
- Turing-complete computation!
  - "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC" (CCS 2008)

More ROP

- Harvard architecture: code separate from data ⇒ code injection is impossible, but ROP works fine
  - Z80 CPU – Sequoia AVC Advantage voting machines
  - Some ARM CPUs – iPhone
- No returns = no problems
  - (Lame) defense against ROP: eliminate sequences with RET and/or look for violations of LIFO call-return order
  - Use update-load-branch sequences in lieu of returns + a trampoline sequence to chain them together
  - Read "Return-oriented programming without returns" (CCS 2010)

Other Issues with W⊕X / DEP

- Some applications require executable stack
  - Example: Lisp interpreters
- Some applications are not linked with /NXcompat
  - DEP disabled (e.g., popular browsers)
- JVM makes all its memory RWX – readable, writable, executable (why?)
  - Spray attack code over memory containing Java objects (how?), pass control to them
- Return into a memory mapping routine, make page containing attack code writeable