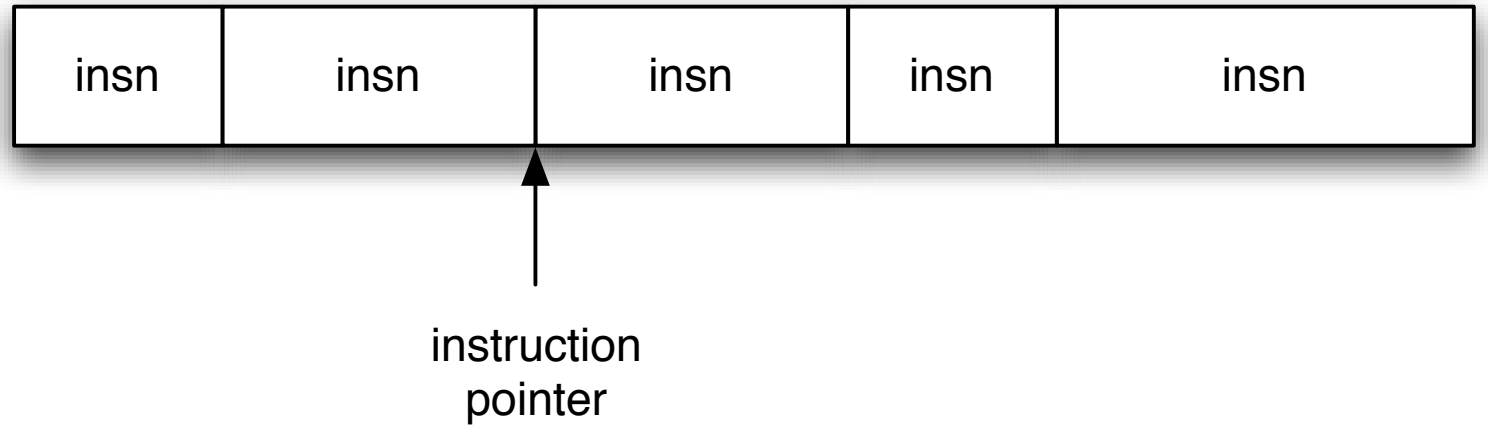
Original at:
http://cseweb.ucsd.edu/~hovav/talks/blackhat08.html
Modified (slides cut) for CMSC 498L, October 4, 2012

Return-oriented Programming: Exploitation without Code Injection

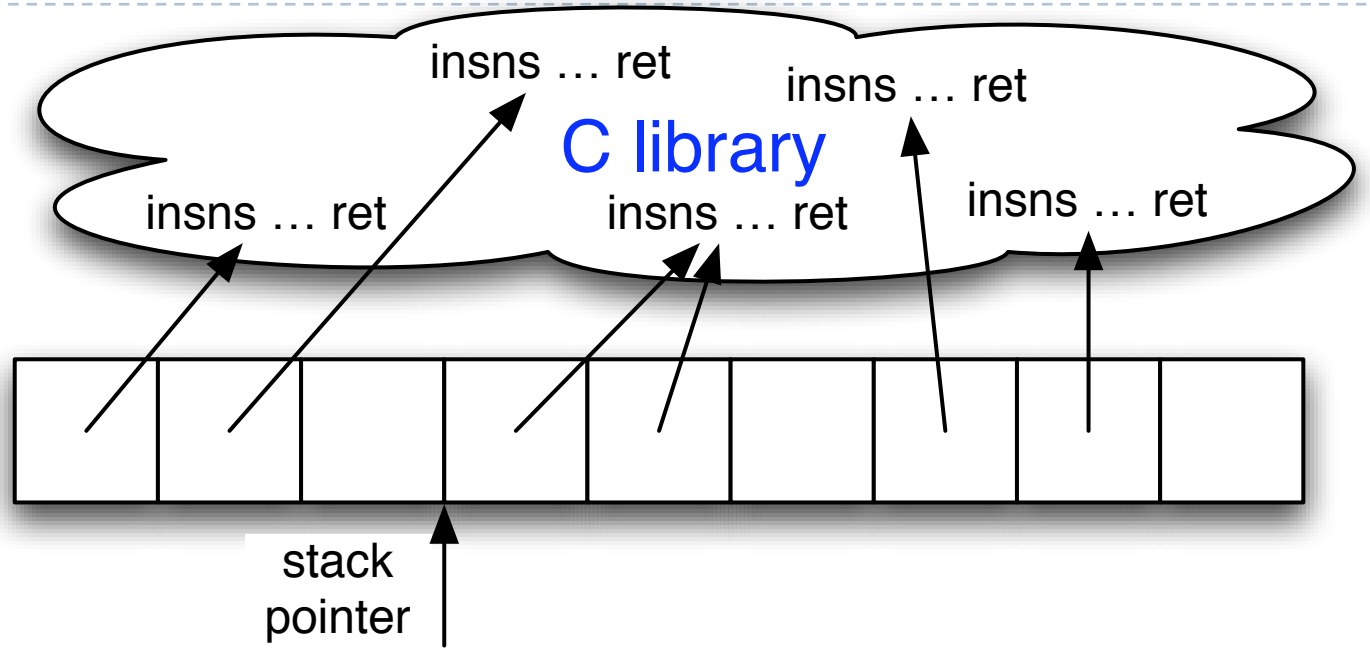
Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham
University of California, San Diego

Ordinary programming: the machine level



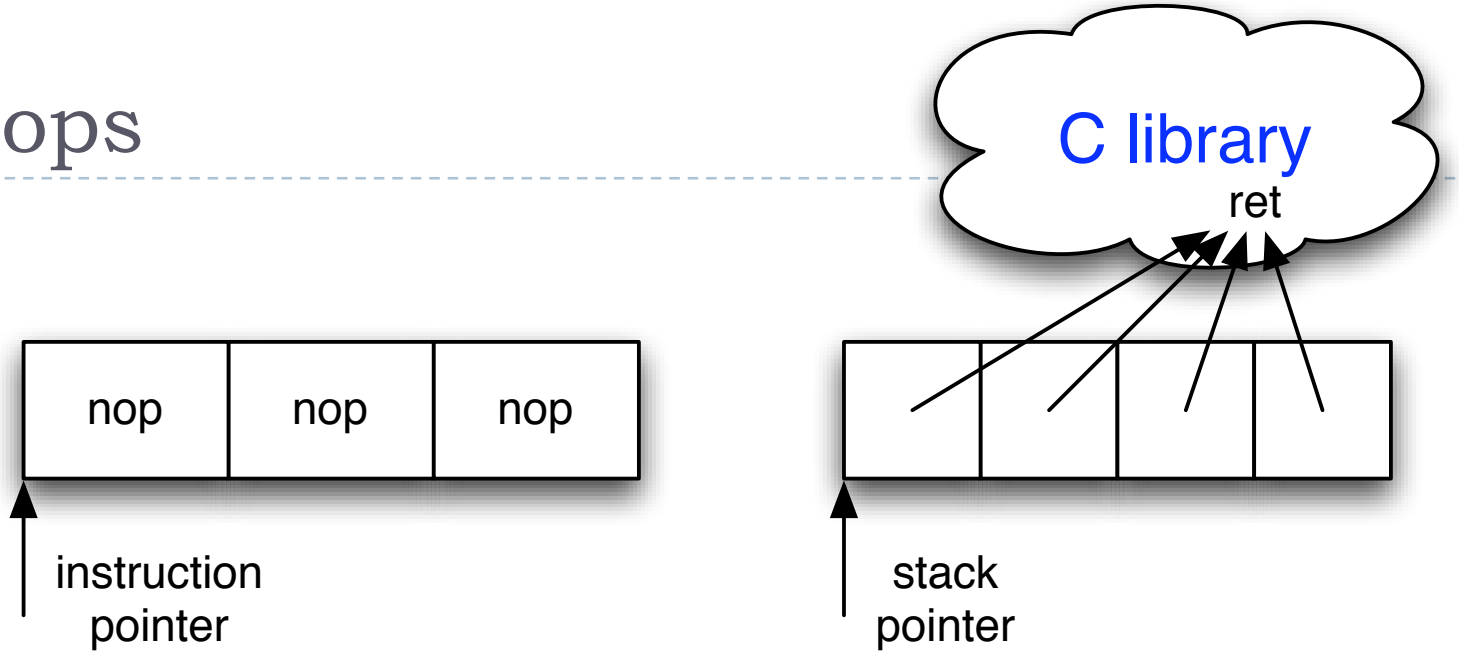
- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip

Return-oriented programming: the machine level



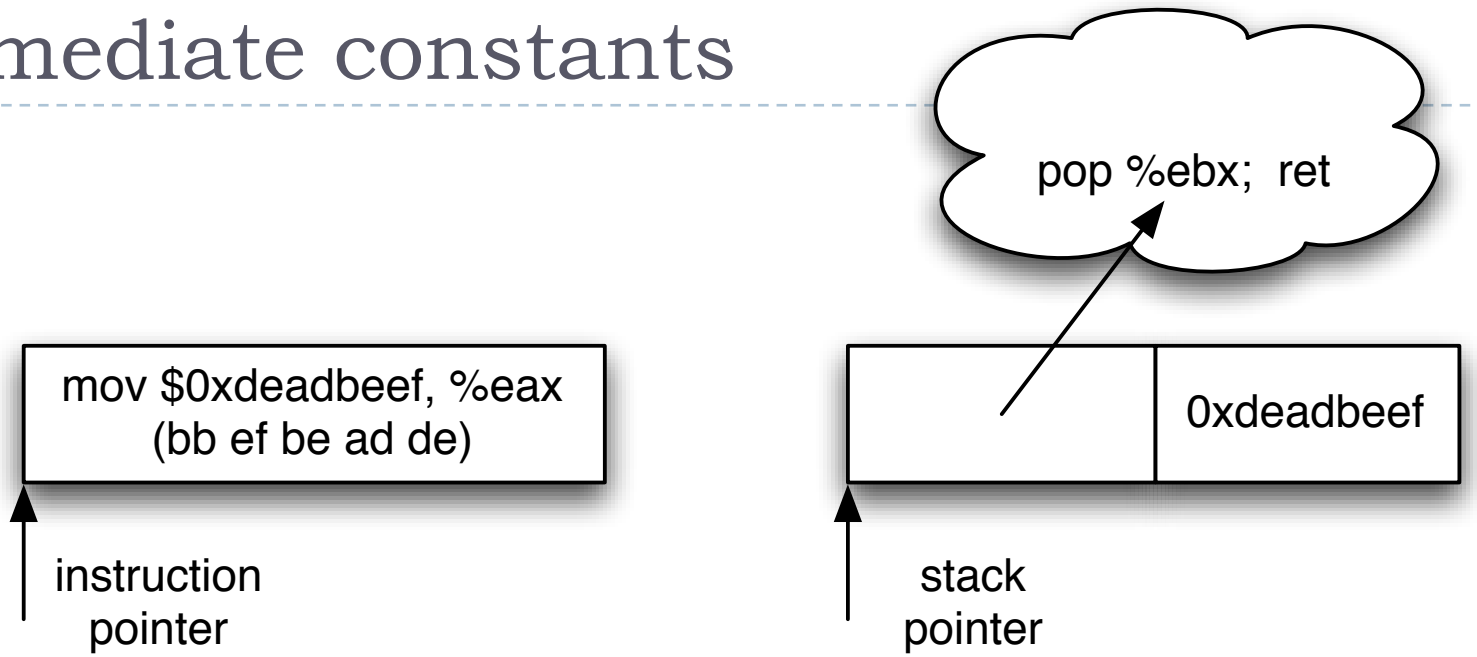
- *Stack pointer* (%esp) determines which instruction sequence to fetch & execute
- Processor doesn't automatically increment %esp; — but the "ret" at end of each instruction sequence does

No-ops



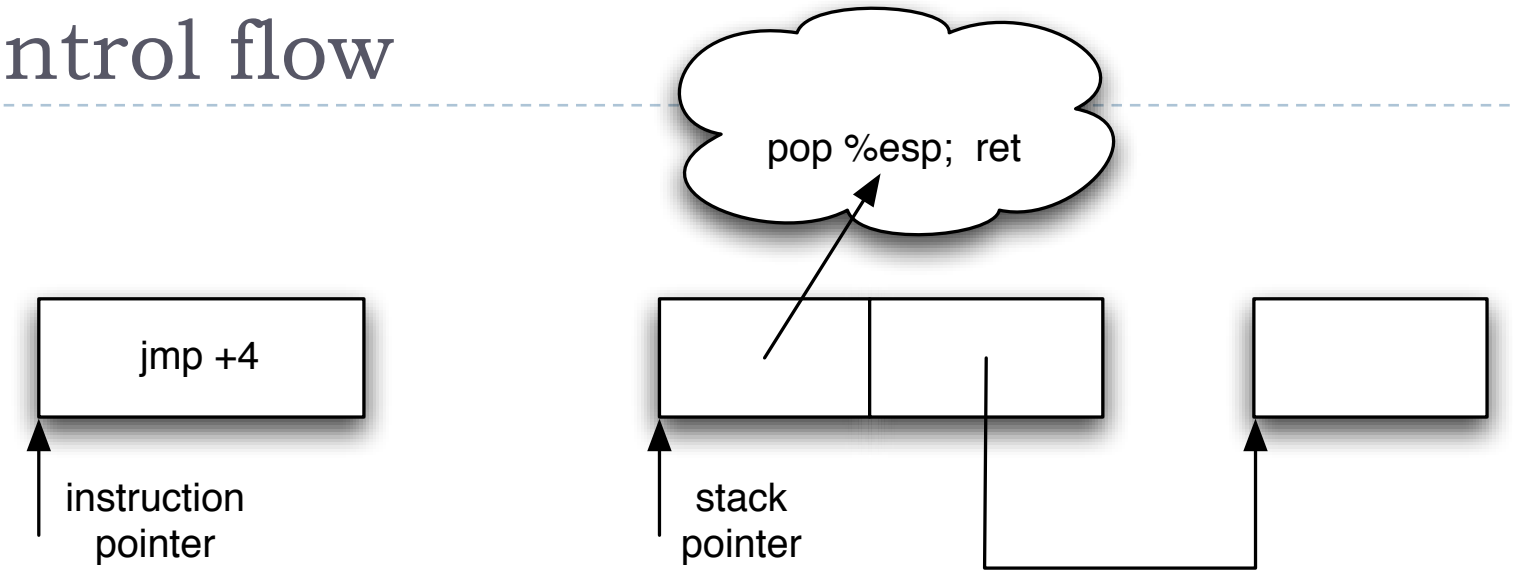
- No-op instruction does nothing but advance %eip
- Return-oriented equivalent:
  - point to return instruction
  - advances %esp
- Useful in nop sled

Immediate constants



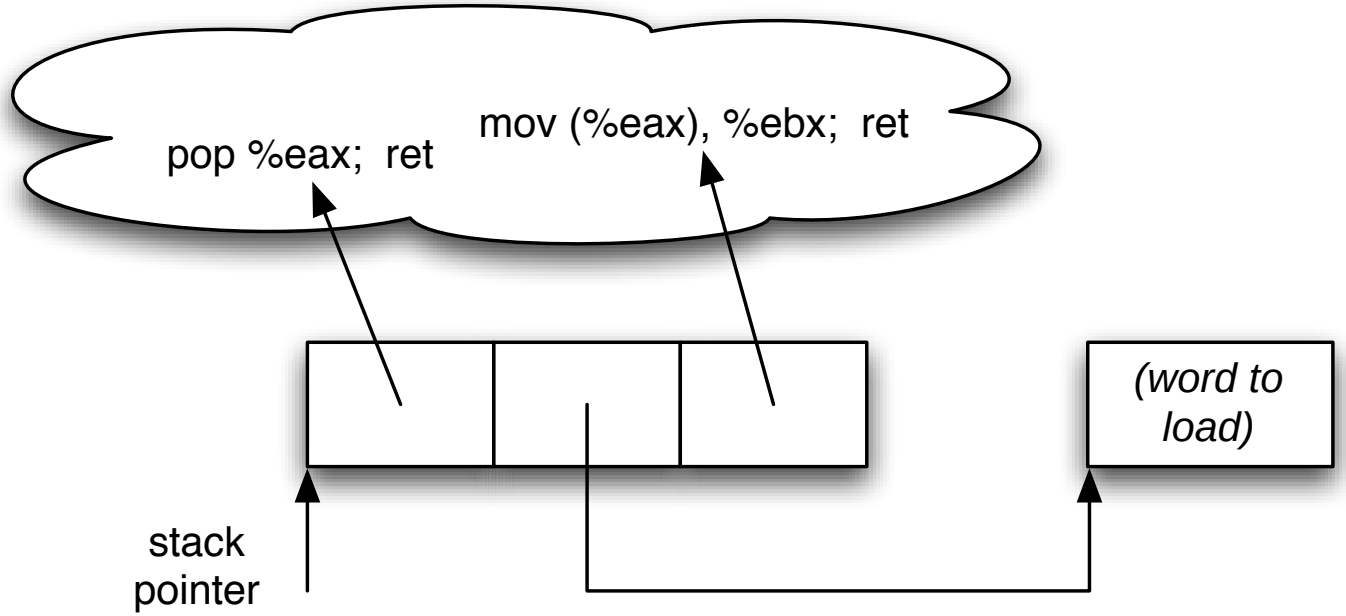
- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use

Control flow



- Ordinary programming:
  - (Conditionally) set %eip to new value
- Return-oriented equivalent:
  - (Conditionally) set %esp to new value

Gadgets: multiple 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

Gadget design

- Testbed: libc-2.3.5.so, Fedora Core 4
- Gadgets built from found code sequences:
  - load-store
  - arithmetic & logic
  - control flow
  - system calls
- Challenges:
  - 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 (e.g., lahf)

Finding instruction sequences

- Any instruction sequence ending in "ret" is useful — could be part of a gadget

- **Algorithmic problem**: recover all sequences of valid instructions from libc that end in a "ret" insn
- Idea: at each ret (c3 byte) look back:
  - are preceding *i* bytes a valid length-*i* insn?
  - recurse from found instructions
- Collect instruction sequences in a trie

Unintended instructions — ecb_crypt()

Return-oriented programming on SPARC

- Use Solaris 10 libc: 1.3 MB
- New techniques:
  - Use instruction sequences that are *suffixes* of real functions
  - Dataflow within a gadget:
    - Use structured dataflow to dovetail with calling convention
  - Dataflow between gadgets:
    - Each gadget is memory-memory
- Turing-complete computation!

- **Conjecture**: Return-oriented programming likely possible on *every* architecture.

Conclusions

- Code injection is not necessary for arbitrary exploitation
- Defenses that distinguish "good code" from "bad code" are useless
- Return-oriented programming likely possible on *every* architecture, not just x86
- Compilers make sophisticated return-oriented exploits easy to write