Return-oriented Programming

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University

Buffer Overflow

EIP

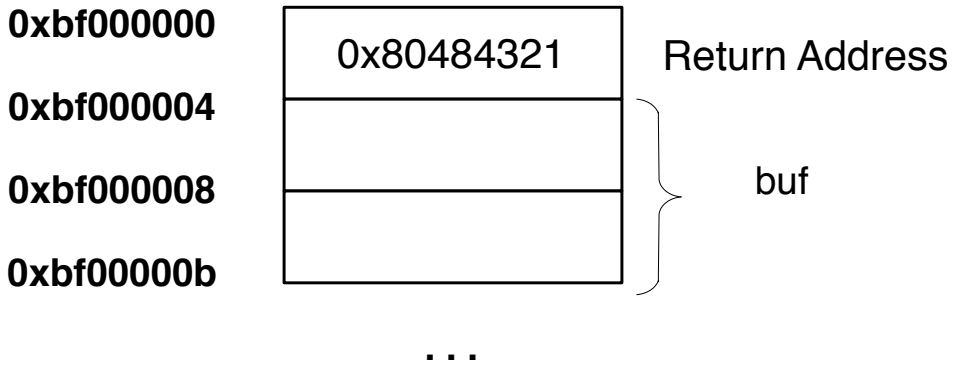
```
                _start:
0x0804321         call main

                int main() {
                  char buf[8];
0x0804480         gets(buf);
                  printf("You typed: %s", buf);
0x0804484       }
```

Buffer Overflow

EIP

ESP

```
                _start:
0x0804321         call main

                int main() {
                  char buf[8];
0x0804480         gets(buf);
                  printf("You typed: %s", buf);
0x0804484       }
```



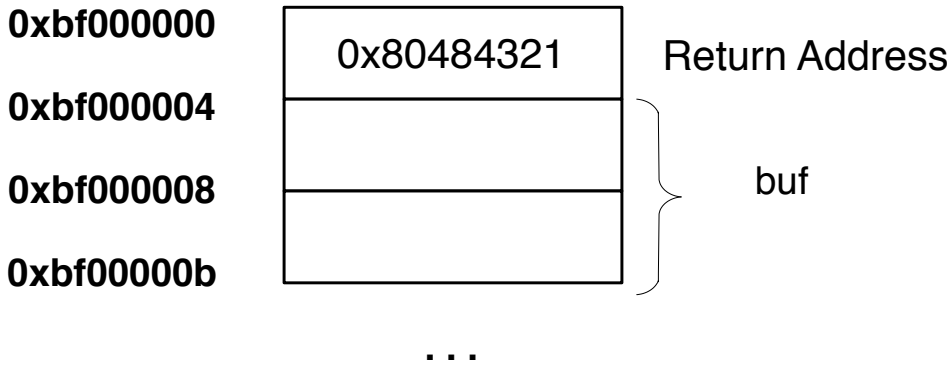
Buffer Overflow

```
0x0804321   _start:
              call main

            int main() {
              char buf[8];
0x0804480     gets(buf);
              printf("You typed: %s", buf);
0x0804484   }
```

EIP

0x12345678 0x90abcdef 0xbf000004

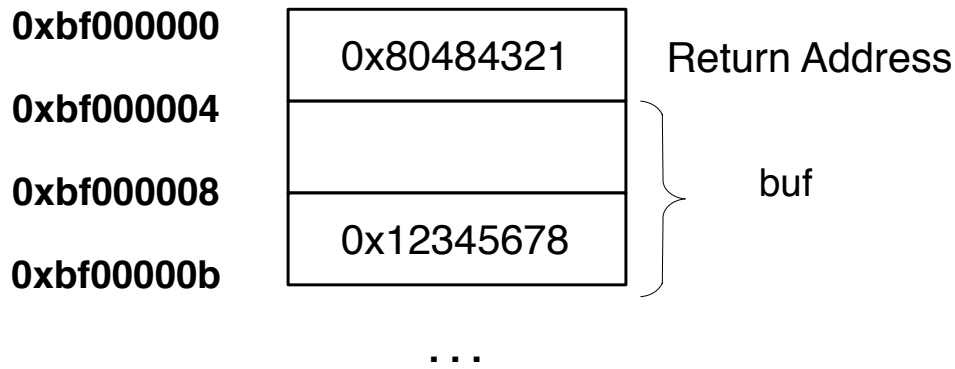
ESP



Buffer Overflow

PENNSTATE

```
                _start:
0x0804321         call main

                int main() {
                  char buf[8];
EIP
0x0804480         gets(buf);        0x12345678 0x90abcdef 0xbf000004
                  printf("You typed: %s", buf);
0x0804484       }
```



ESP

Buffer Overflow

PENNSTATE

EIP

```
0x0804321   _start:
              call main

            int main() {
              char buf[8];
0x0804480     gets(buf);      0x12345678 0x90abcdef 0xbf000004
              printf("You typed: %s", buf);
0x0804484   }
```

ESP

Buffer Overflow

PENNSTATE 1855

EIP

```
0x0804321   _start:
              call main

            int main() {
              char buf[8];
0x0804480     gets(buf);      0x12345678 0x90abcdef 0xbf000004
              printf("You typed: %s", buf);
0x0804484   }
```

ESP

Buffer Overflow

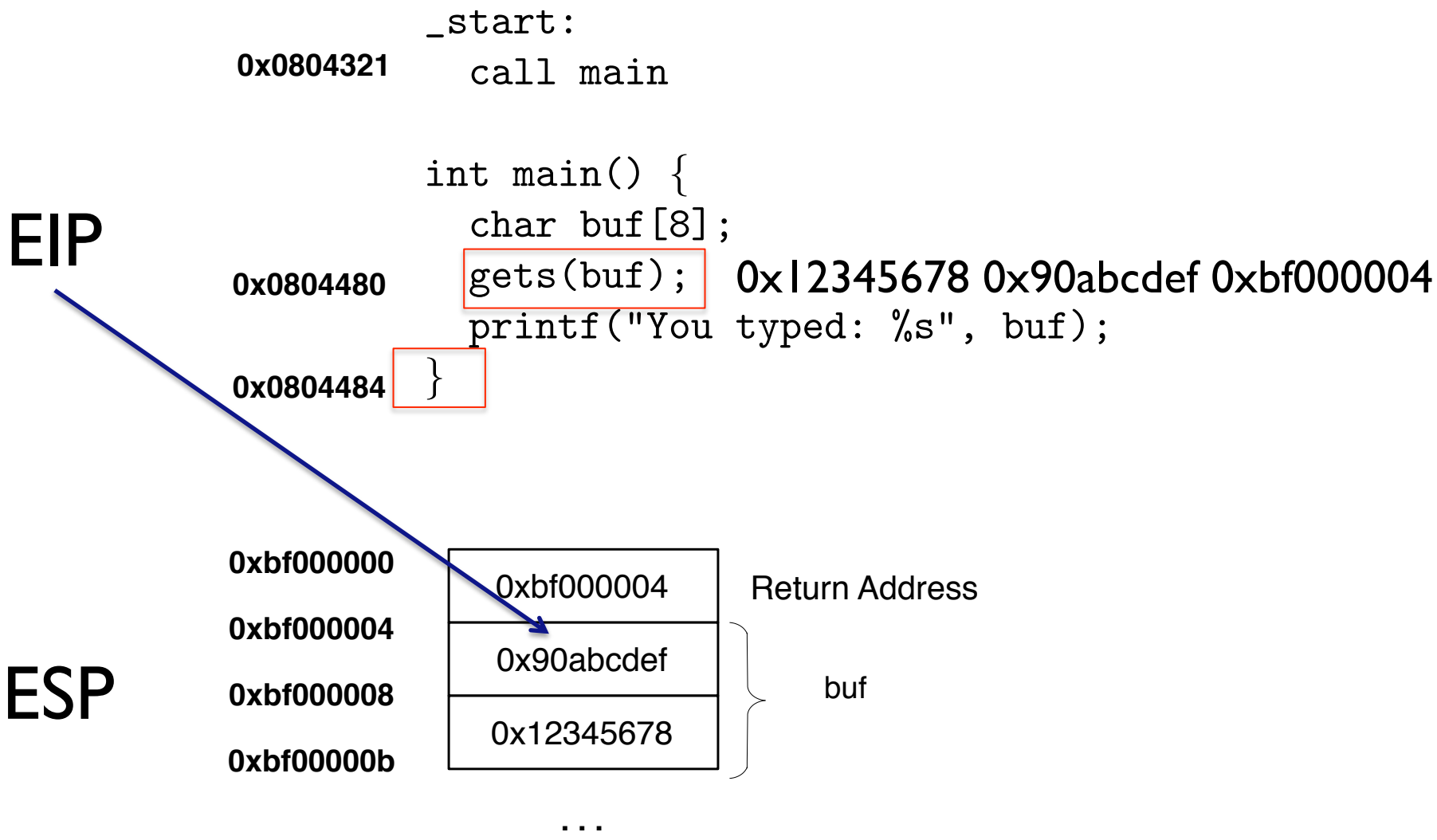
PENNSTATE 1855

EIP

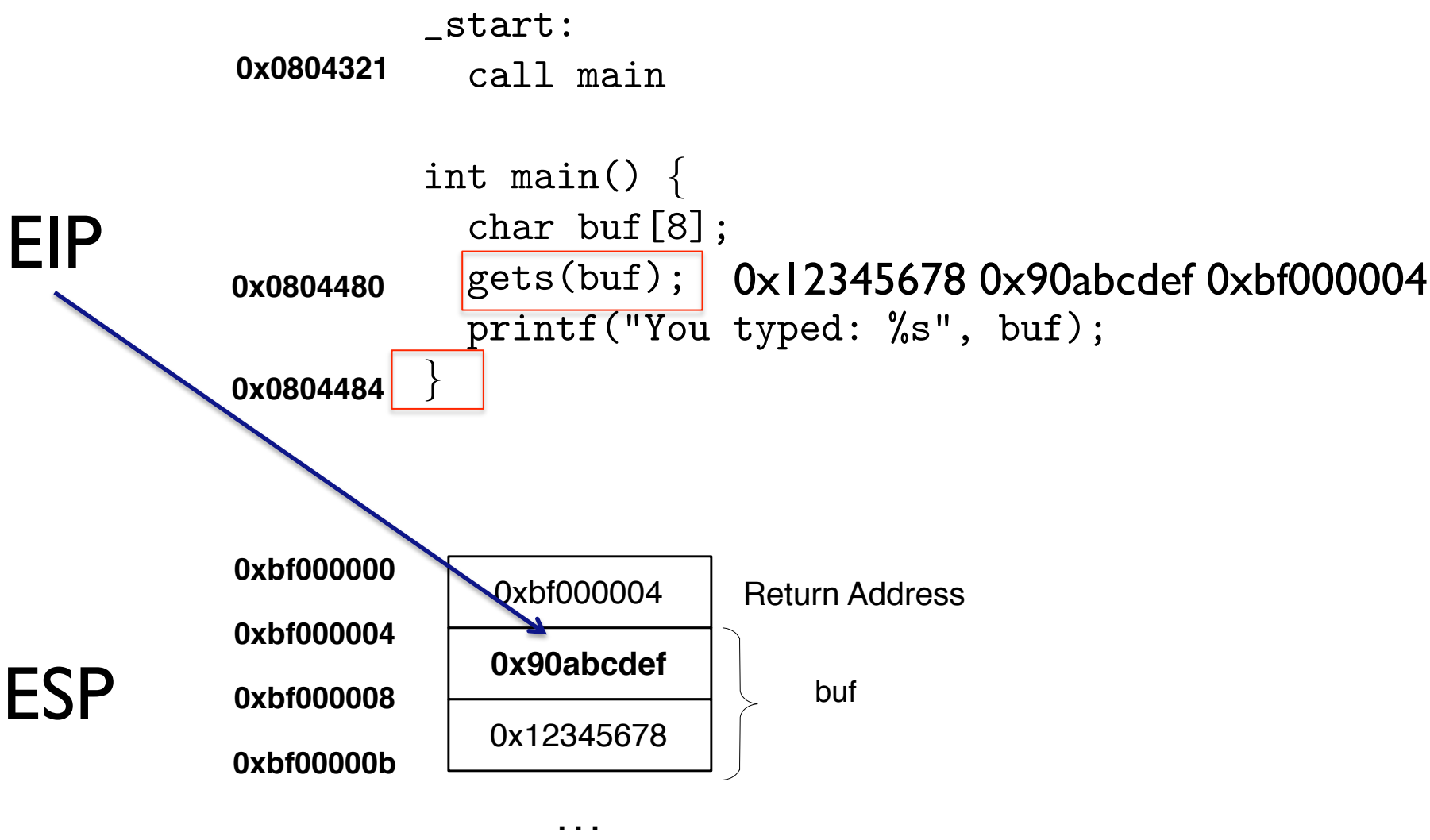
```
                _start:
0x0804321         call main

                int main() {
                  char buf[8];
0x0804480         gets(buf);      0x12345678 0x90abcdef 0xbf000004
                  printf("You typed: %s", buf);
0x0804484       }
```

ESP

Buffer Overflow

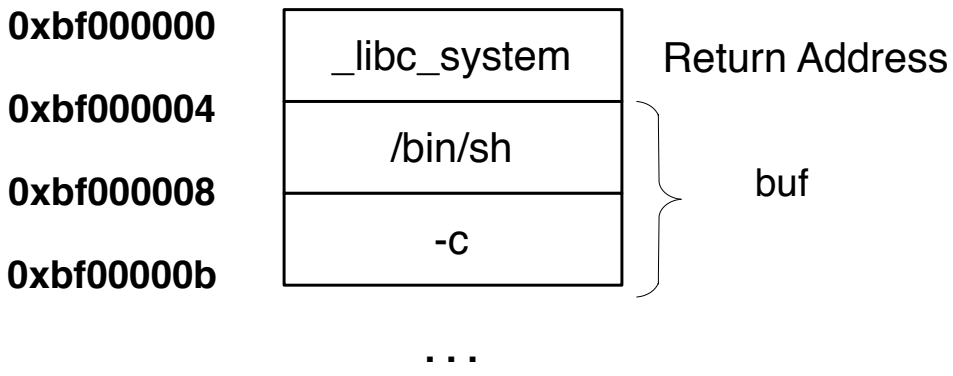


Buffer Overflow



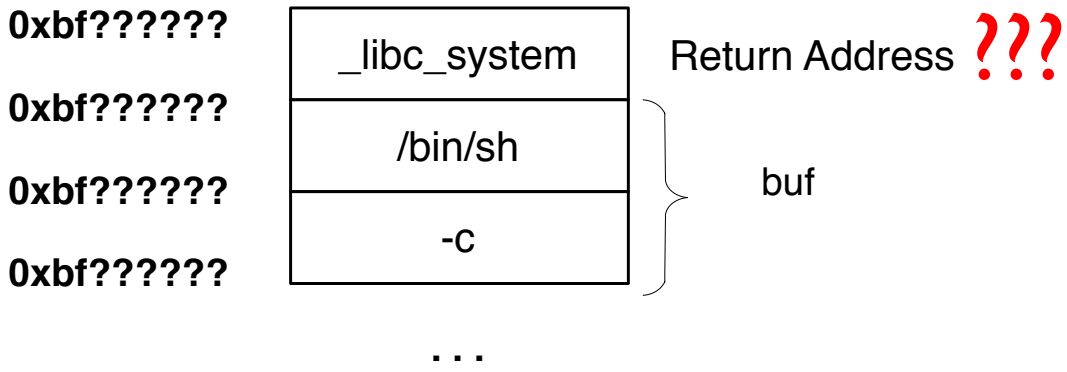
Buffer Overflow Defense

- W xor X
  - Pages marked write can't be executed
- Return-to-libc



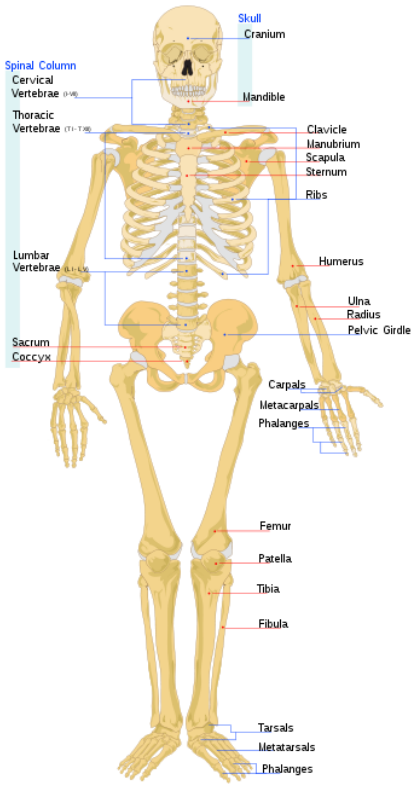
ASLR

- Randomize bases of memory regions
  - Stack (Thwarts traditional stack overflow)
  - Mmap (Thwarts return-to-libc)
  - Brk (Heap – Thwarts traditional heap overflow)
  - Exec (Program binary)
    - Not enabled by default



Anatomy of Control Flow Attacks

- Two steps
- First, the attacker changes the control flow of the program
  - In buffer overflow, overwrite the return address on the stack
  - What are the ways that this can be done?
- Second, the attacker uses this change to run code of their choice
  - In buffer overflow, inject code on stack
  - What are the ways that this can be done?



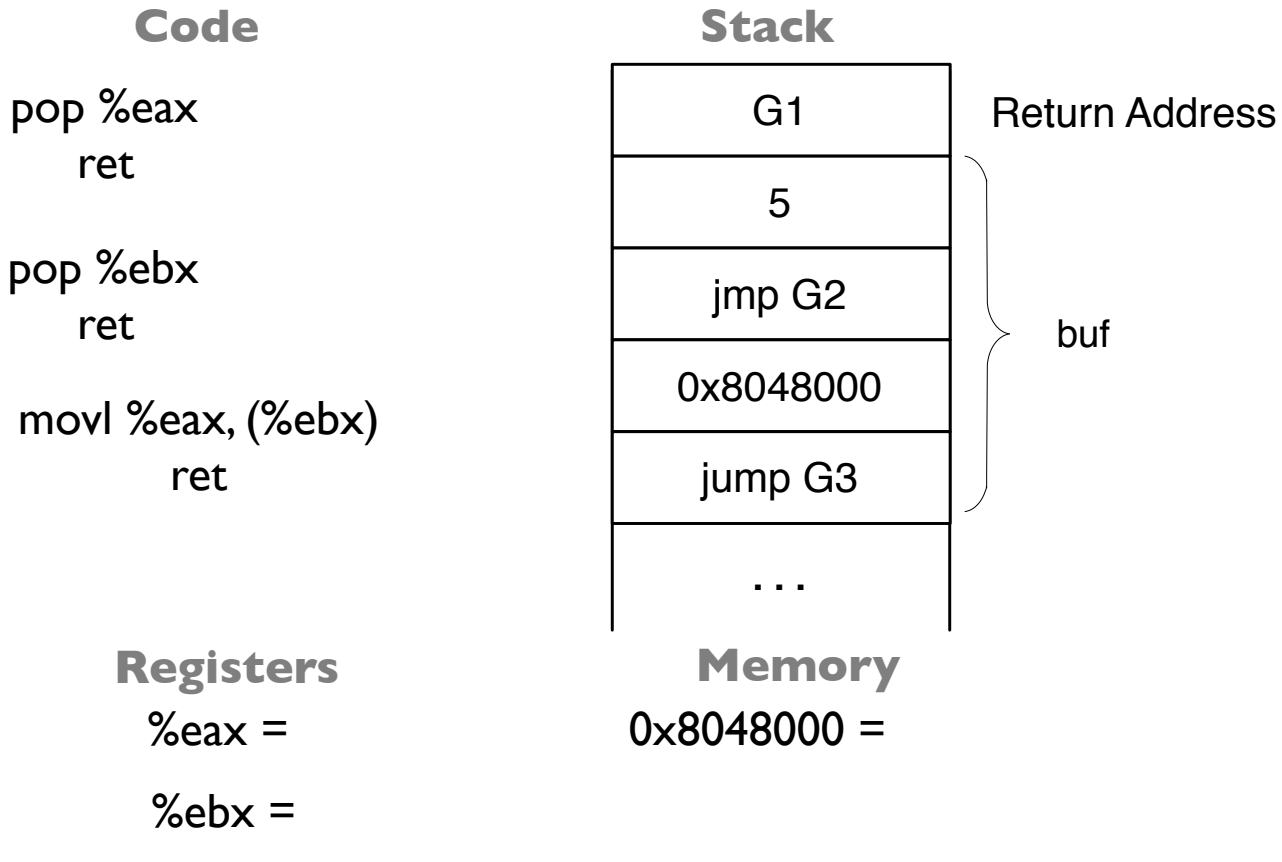
Return-oriented Programming

PENNSTATE 1855

- General approach to control flow attacks
- Demonstrates how general the two steps of a control flow attack can be
- First, change program control flow
  - In any way
- Then, run any code of attackers' choosing, including the code in the existing program

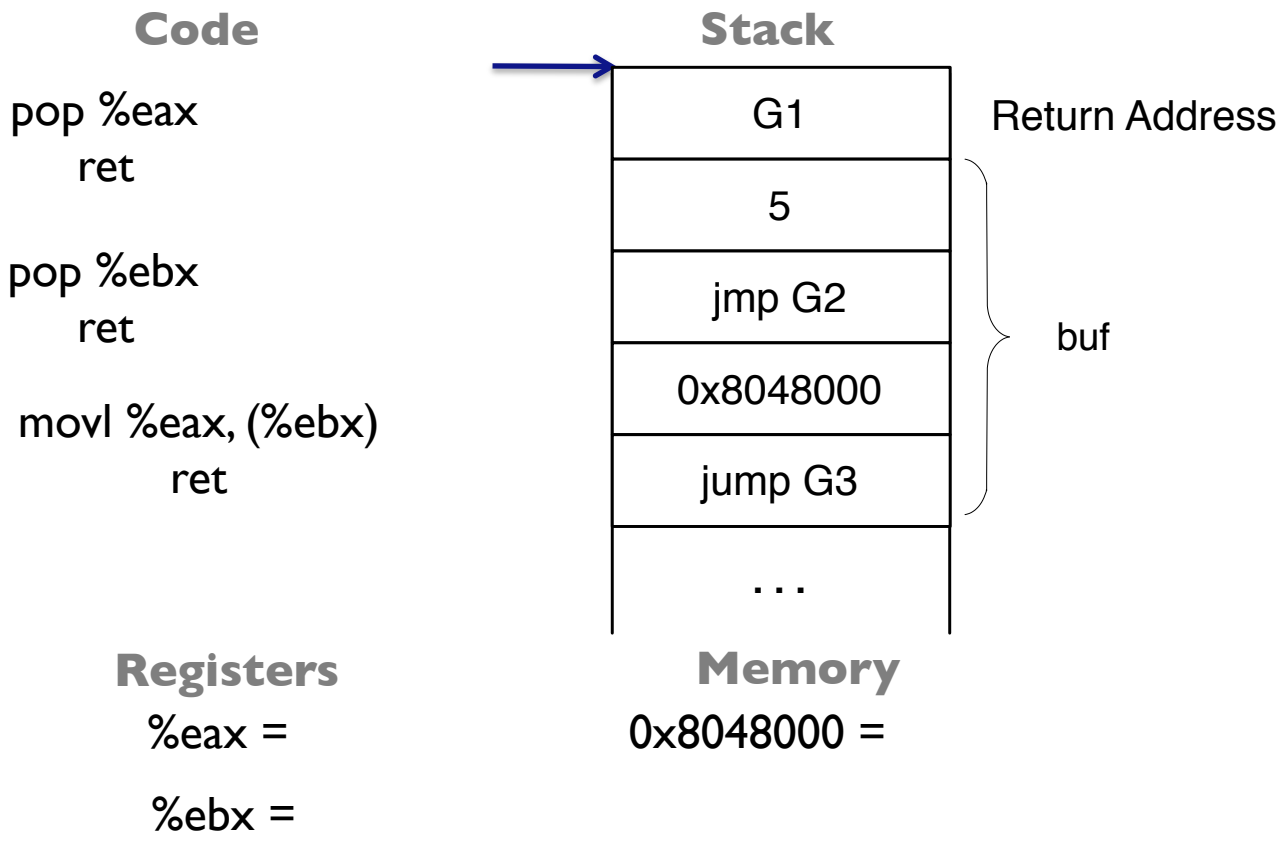
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



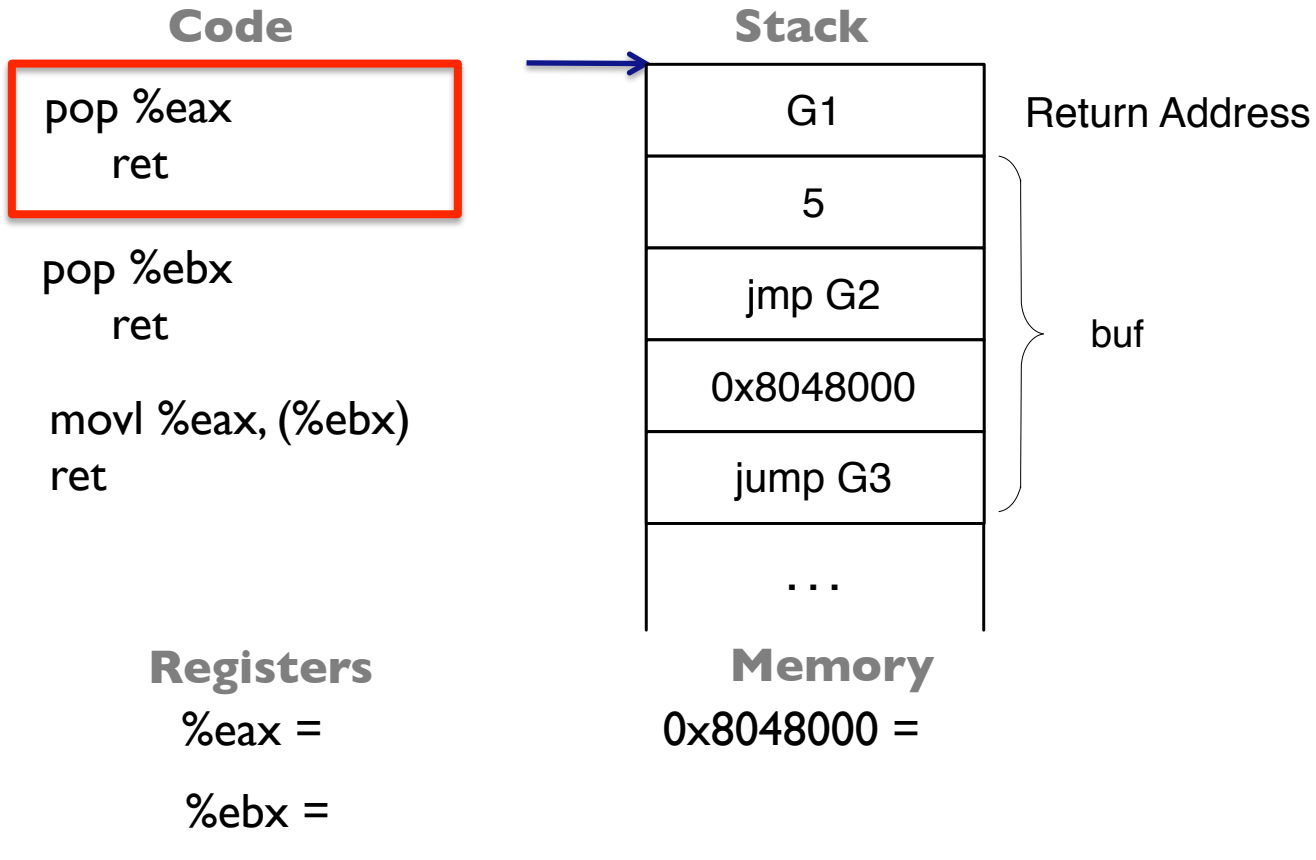
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



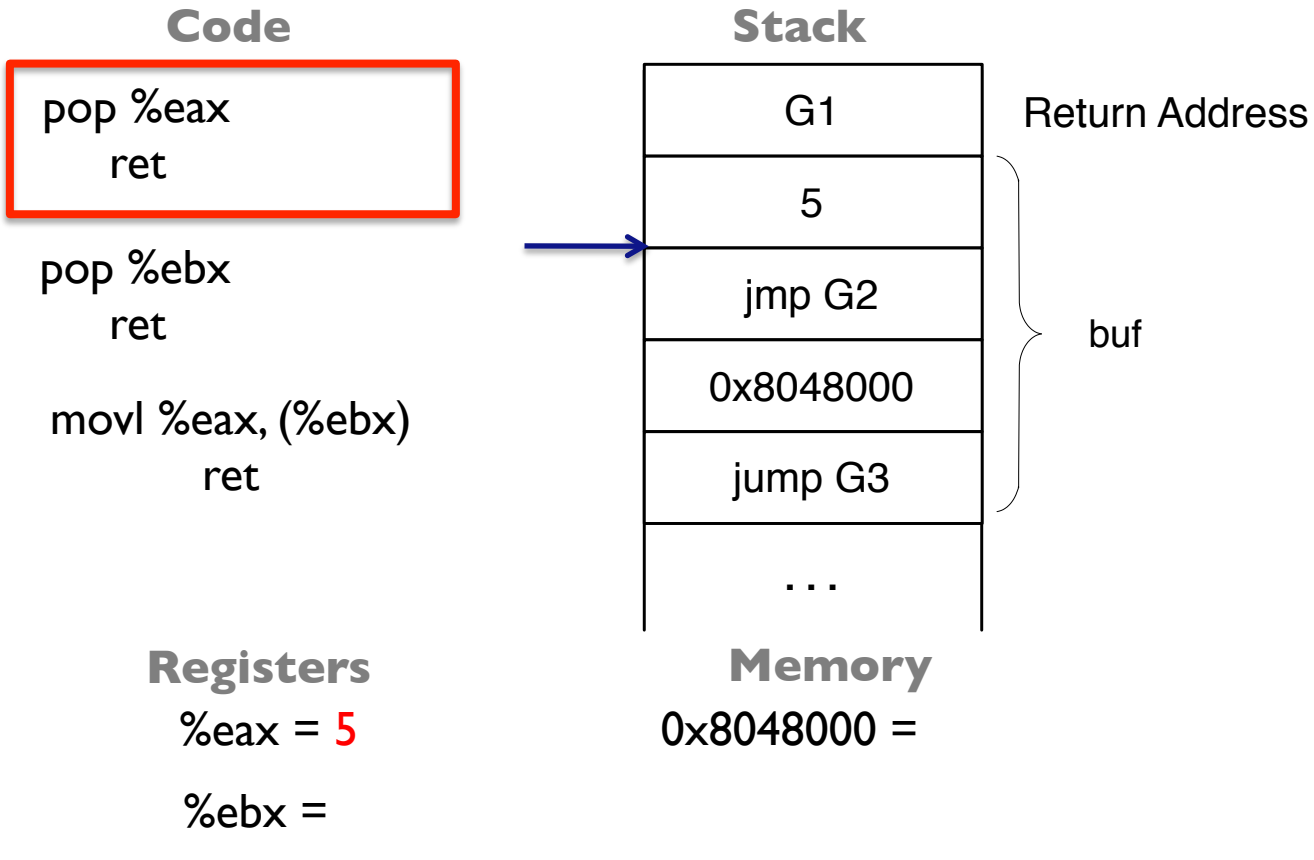
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



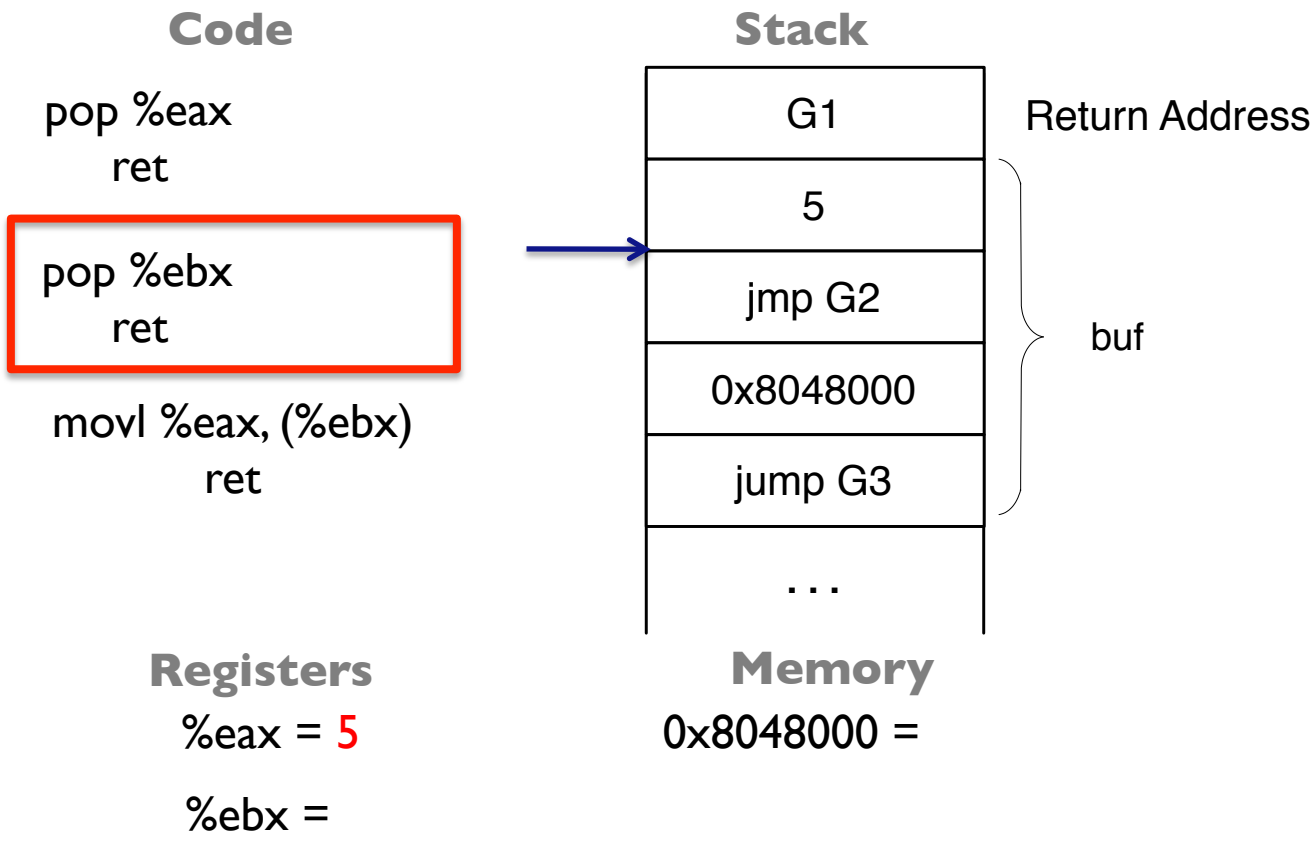
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



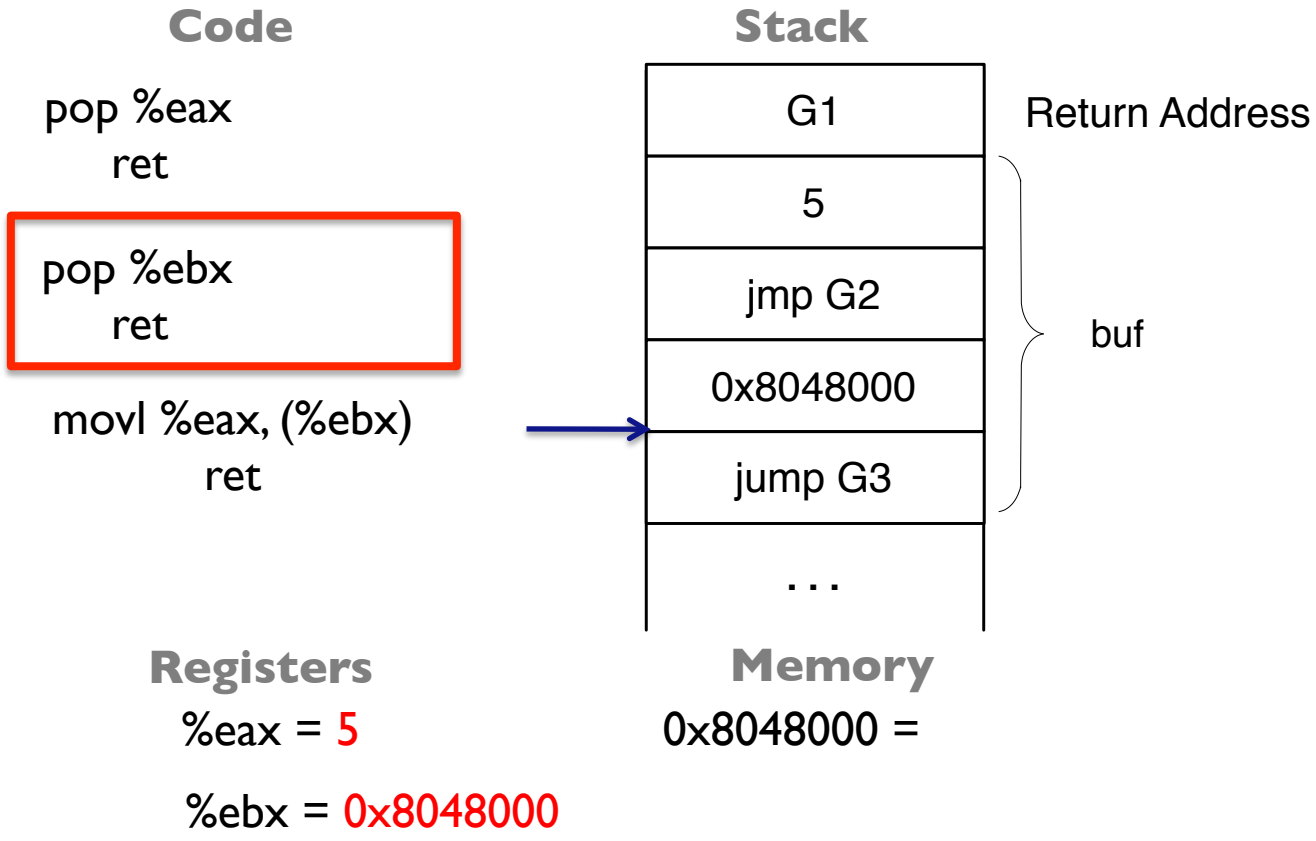
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



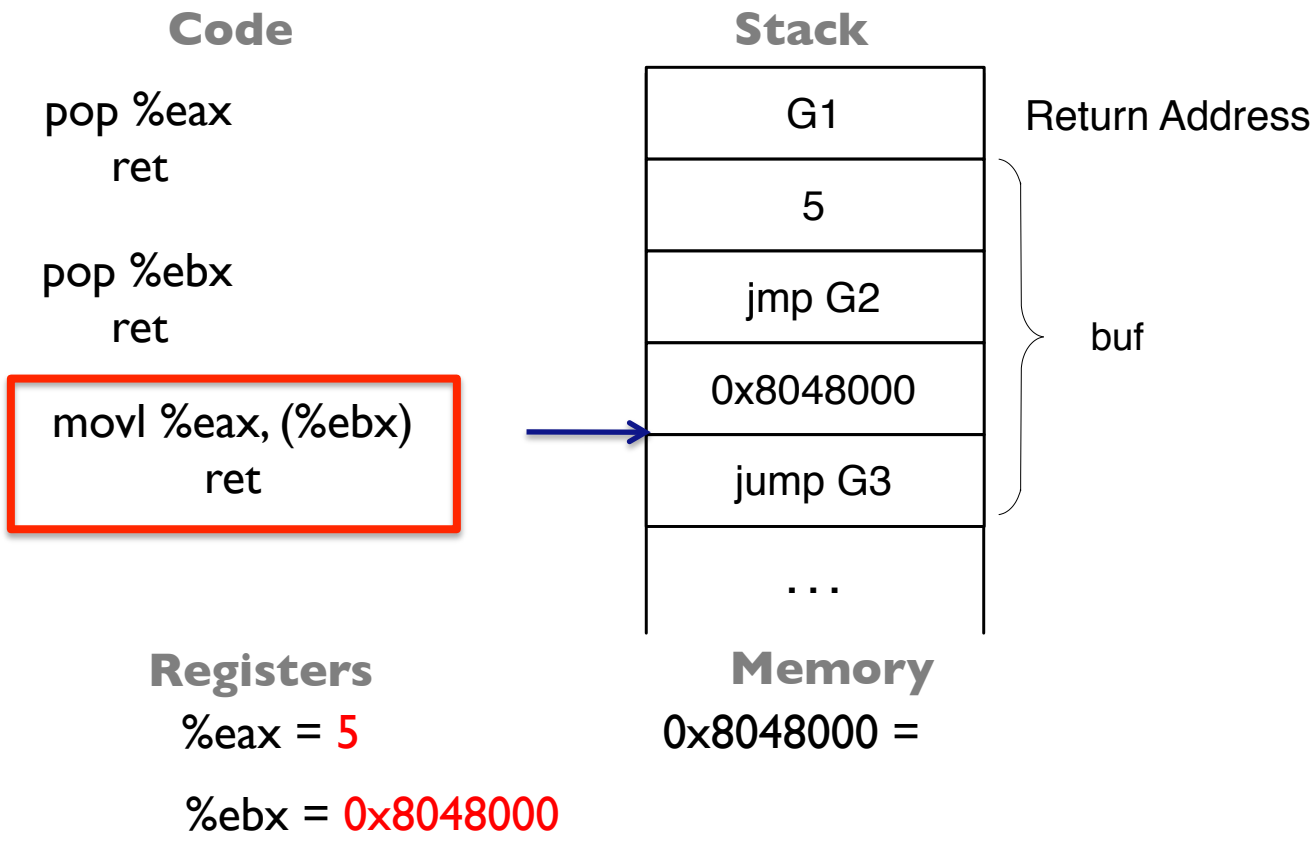
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



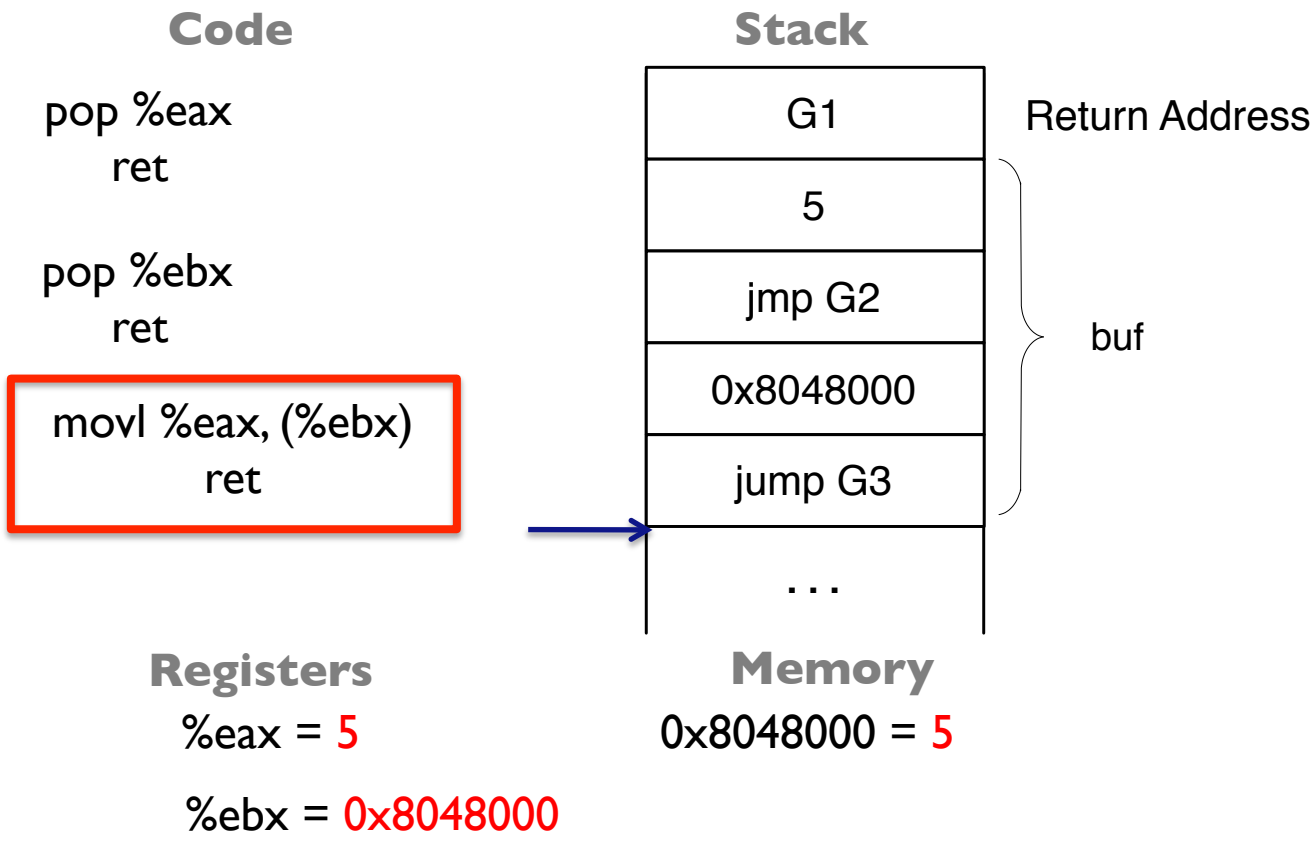
ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



ROP

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code



Return-oriented Programming

PENNSTATE 1855

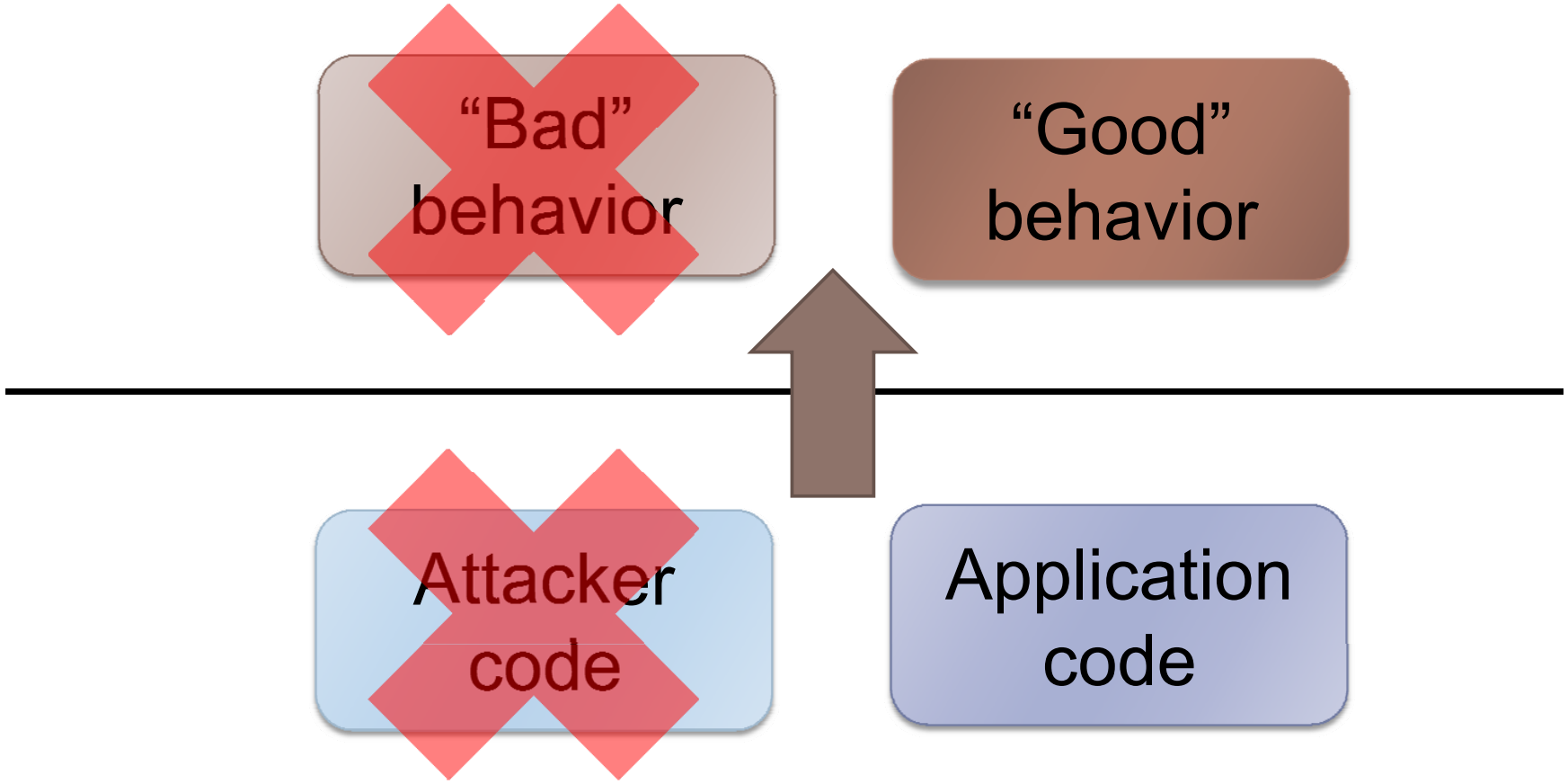
- How can an adversary make this happen?

Return-oriented Programming:
Exploitation without Code Injection

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

Return-oriented Programming

Bad code versus bad behavior



Problem: this implication is false!

ROP Thesis

any sufficiently large program codebase

arbitrary attacker computation and behavior,
without code injection

(in the absence of control-flow integrity)

Return-to-libc

- Divert control flow of exploited program into libc code
  - system(), printf(),
- No code injection required

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc — remove system()?

- We show: this perception is *false.*

ROP vs. Return-to-libc

attacker control of stack

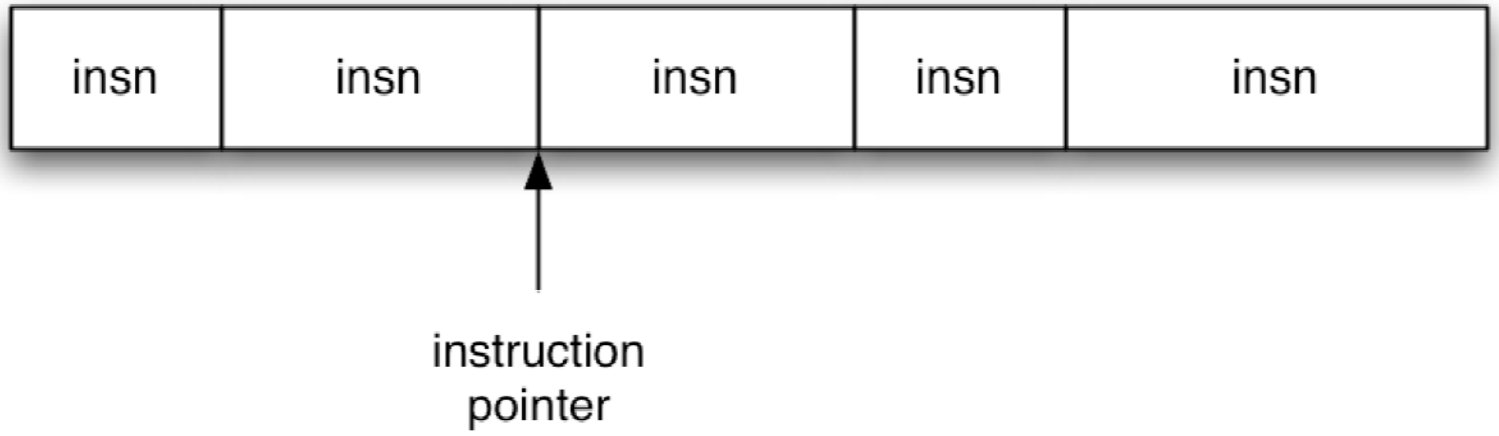
arbitrary attacker computation and behavior
via return-into-libc techniques

(given any sufficiently large codebase to draw on)

ROP Attacks

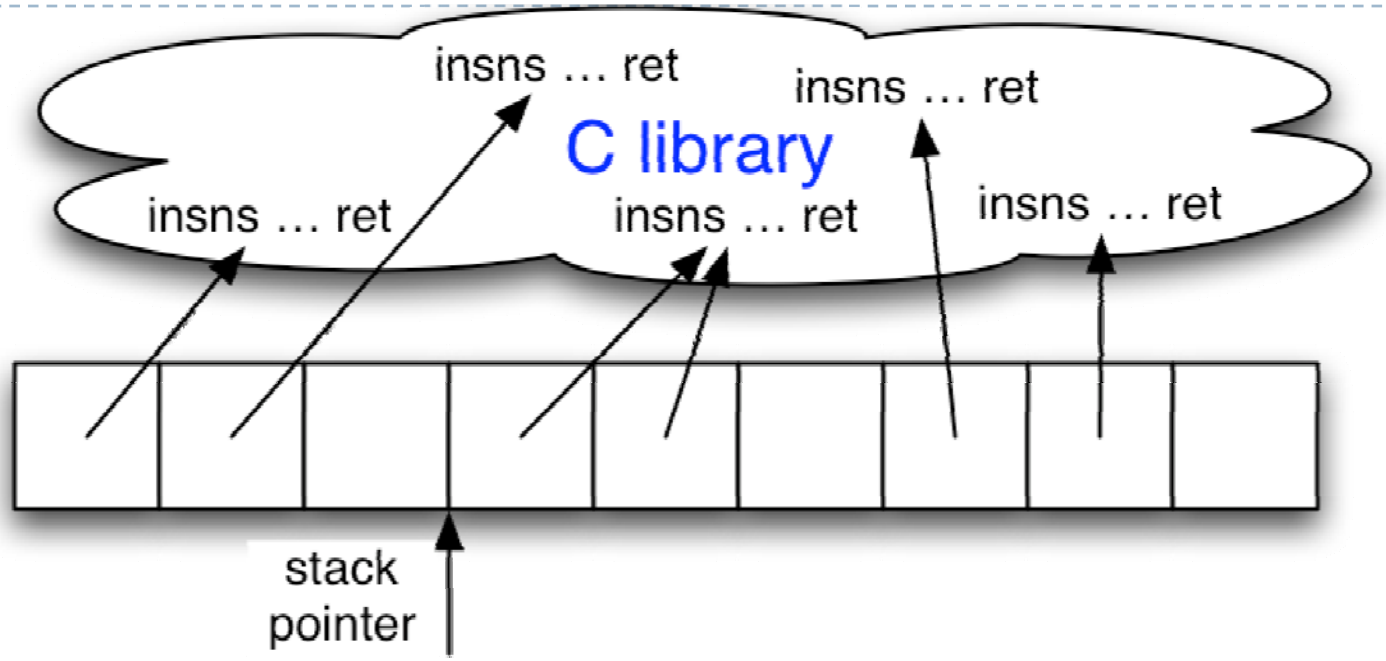
- Need control of memory around %esp
- Rewrite stack:
  - Buffer overflow on stack
  - Format string vuln to rewrite stack contents
- Move stack:
  - Overwrite saved frame pointer on stack;
    on leave/ret, move %esp to area under attacker control
  - Overflow function pointer to a register spring for %esp:
    - set or modify %esp from an attacker-controlled register
    - then return

Machine Instructions



- 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

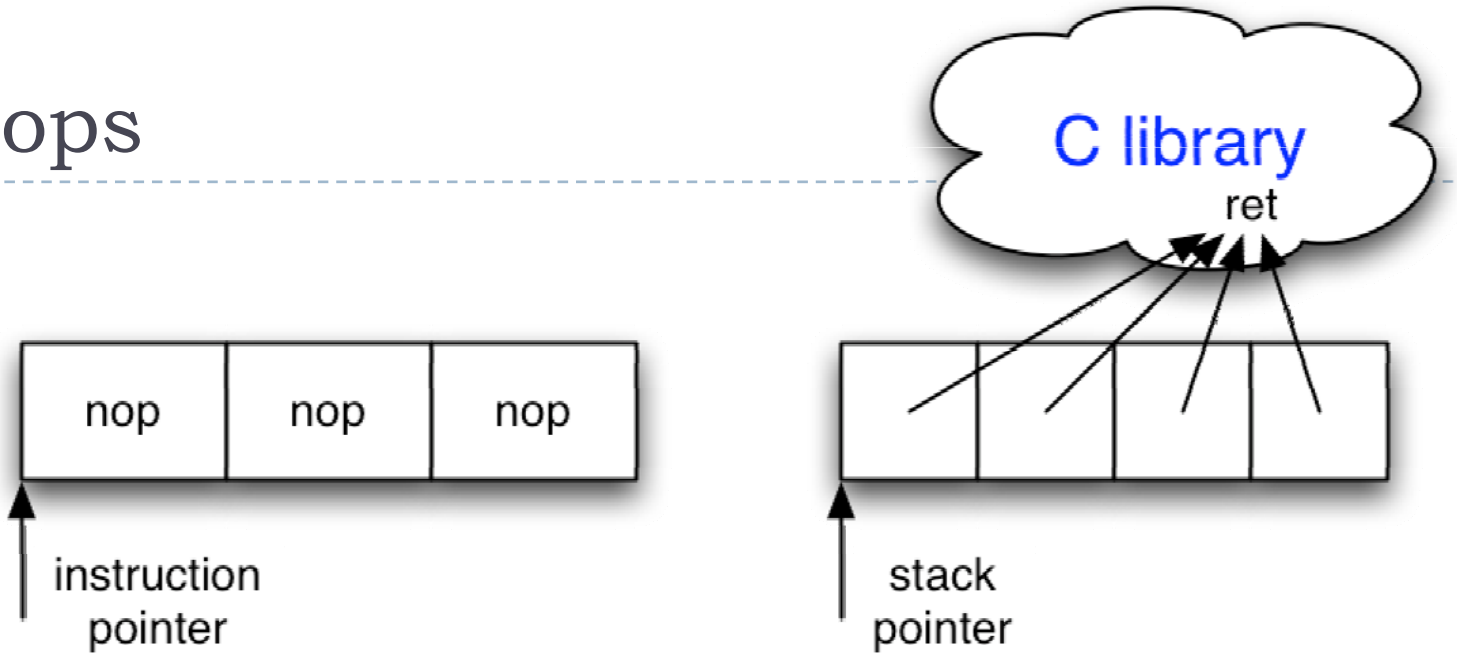
ROP Execution



- *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

Building ROP Functionality

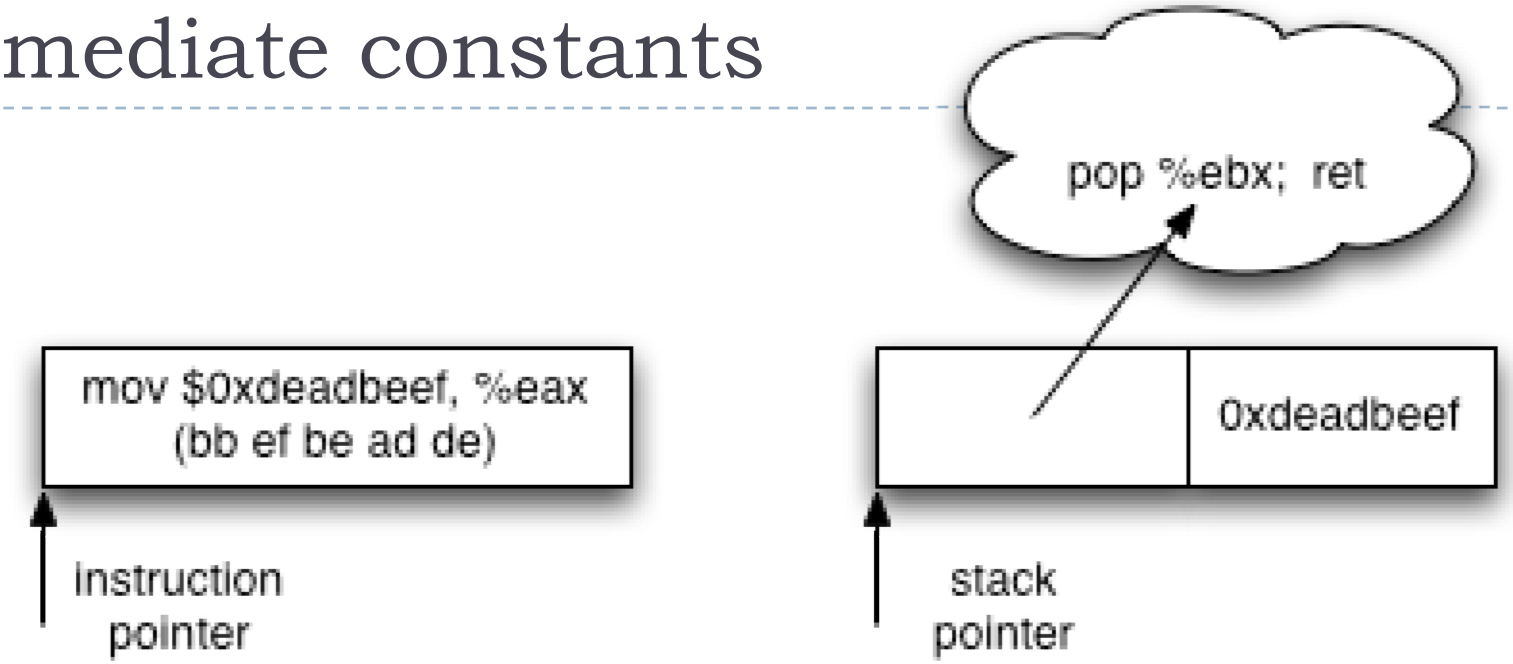
No-ops



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

Building ROP Functionality

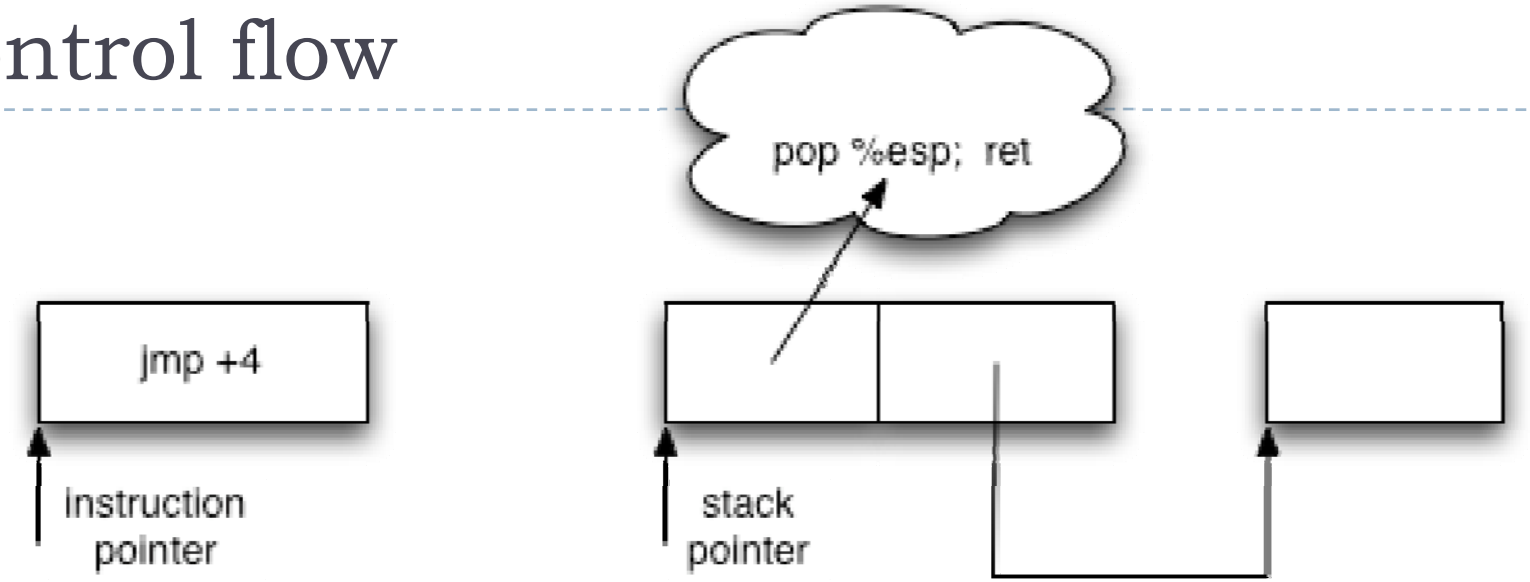
Immediate constants



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

Building ROP Functionality

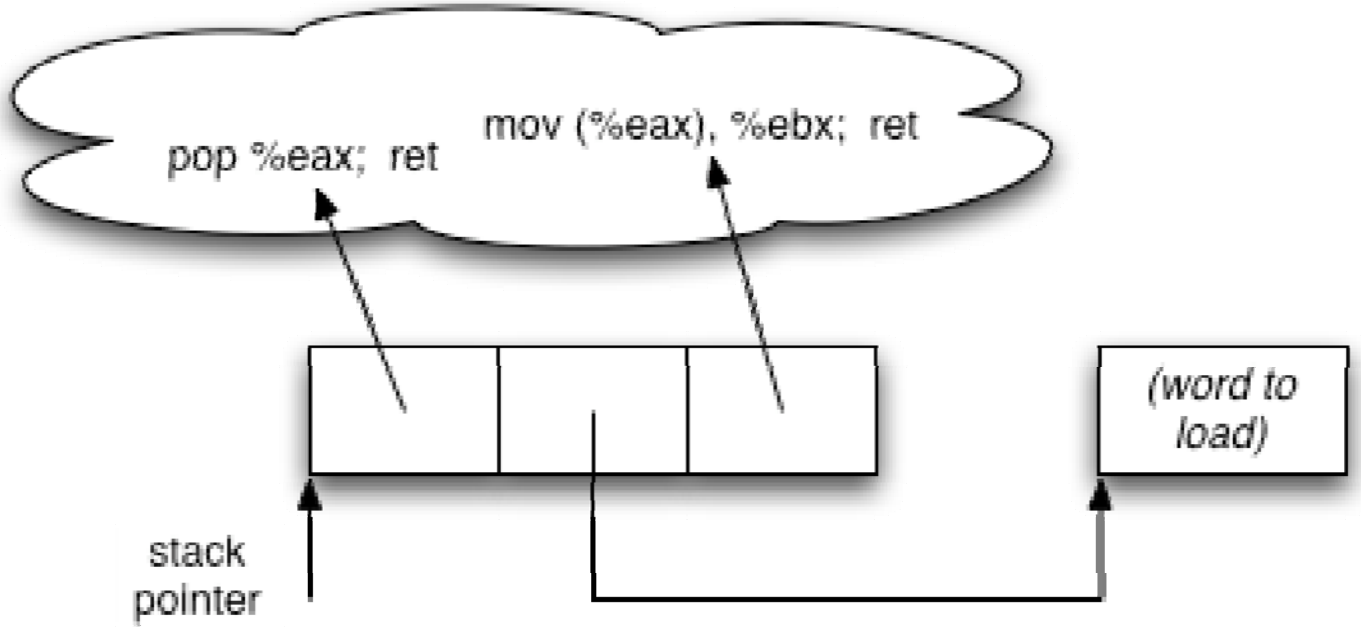
Control flow



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

Creating Programs

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

Finding Gadgets

PENNSTATE 1855

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?
  - recursefrom found instructions
- Collect instruction sequences in a trie

Works on non-x86 Systems

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.

Works on non-x86 Systems

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

Summary

- The types of attacks that we must defend against are becoming more complex
- Return-oriented programming shows us that *any* attacker-dictated change in program control flow can lead to arbitrary malice
- Stuxnet shows that ad hoc system defenses can be evaded by an adversary
- We must apply principled approaches to defense to make significant strides in defense