CSE 484 / CSE M 584 (Winter 2013)

Software Security

Tadayoshi Kohno

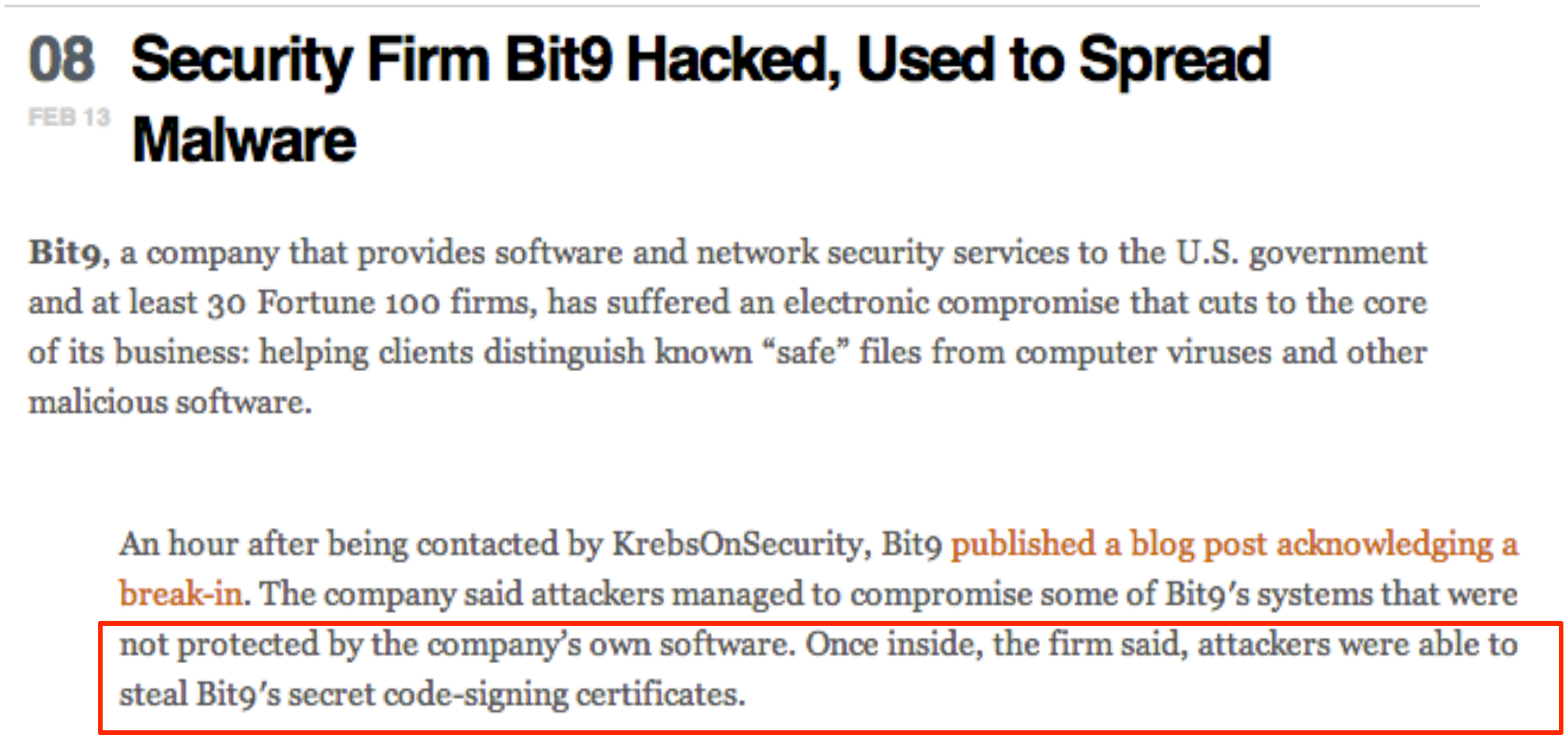
Thanks to Vitaly Shmatikov, Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Bennet Yee, and many others for sample slides and materials ...

Goals for Today

- Software security

- Lab 1: Awesome!
- HW2 out last week

Compromising Asymmetric Private Keys



08 FEB 13

Security Firm Bit9 Hacked, Used to Spread Malware

Bit9, a company that provides software and network security services to the U.S. government and at least 30 Fortune 100 firms, has suffered an electronic compromise that cuts to the core of its business: helping clients distinguish known "safe" files from computer viruses and other malicious software.

An hour after being contacted by KrebsOnSecurity, Bit9 published a blog post acknowledging a break-in. The company said attackers managed to compromise some of Bit9's systems that were not protected by the company's own software. Once inside, the firm said, attackers were able to steal Bit9's secret code-signing certificates.

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

Following slides adopted from Vitaly Shmatikov and Hovav Shacham

W⊕X / DEP

- Mark all writeable memory locations as non-executable
  - Example: Microsoft's DEP - Data Execution Prevention
  - This blocks many (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
- 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
- May not look like a huge threat
  - Attacker cannot execute arbitrary code
  - … especially if system() is not available

return-to-libc on Steroids (Hovav Shacham, CCS 2007)

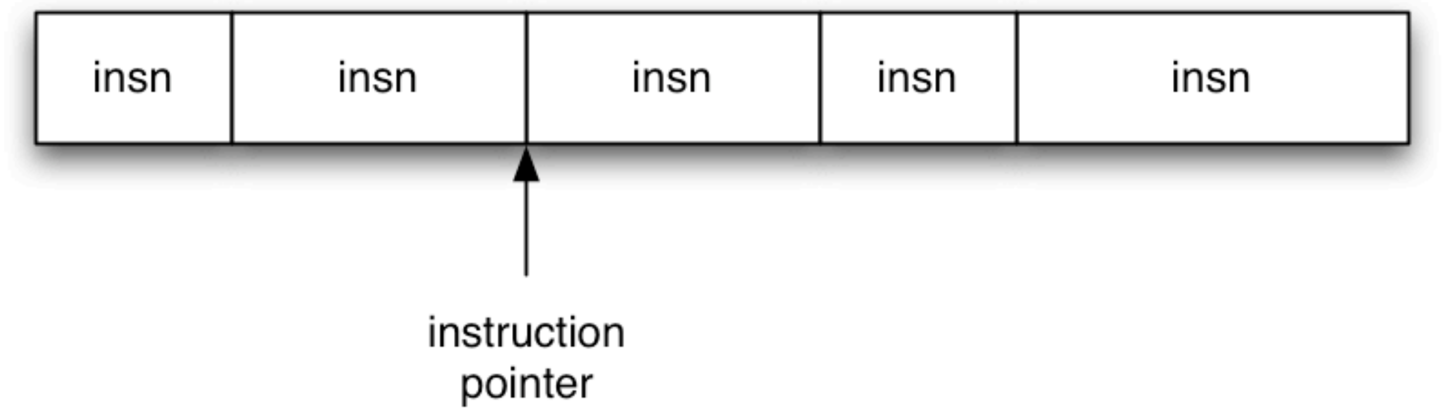
- Overwritten saved EIP need not point to the beginning of a library routine
- **Any** existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read the word pointed to by stack pointer (ESP)
    - Guess what? Its value is under attacker's control! (why?)
      - 0x80004c3 <main+51>: movl %ebp,%esp
      - 0x80004c5 <main+53>: popl %ebp
      - 0x80004c6 <main+54>: ret
  - 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]

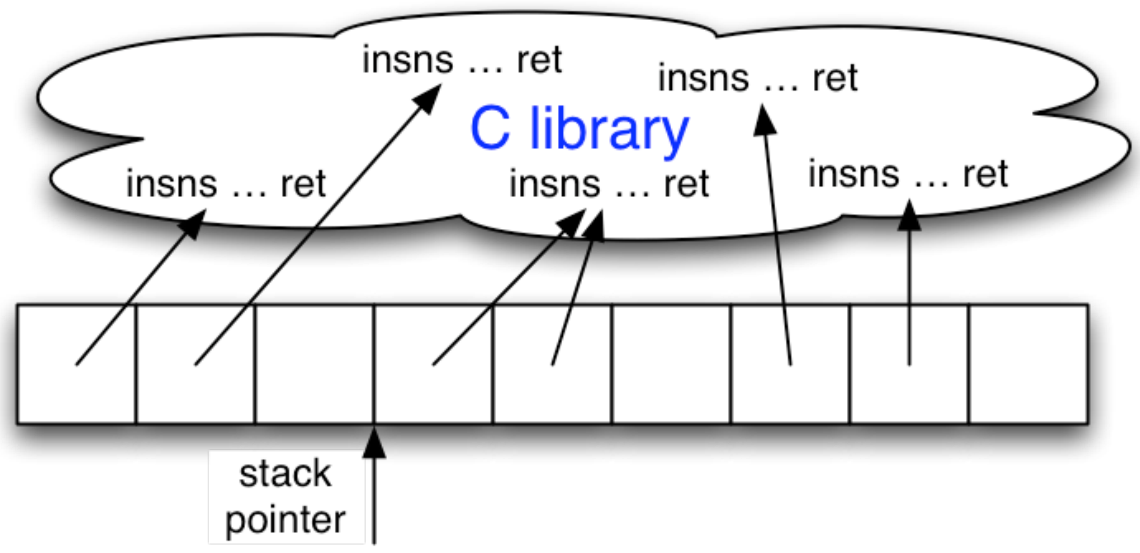
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control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

Ordinary Programming



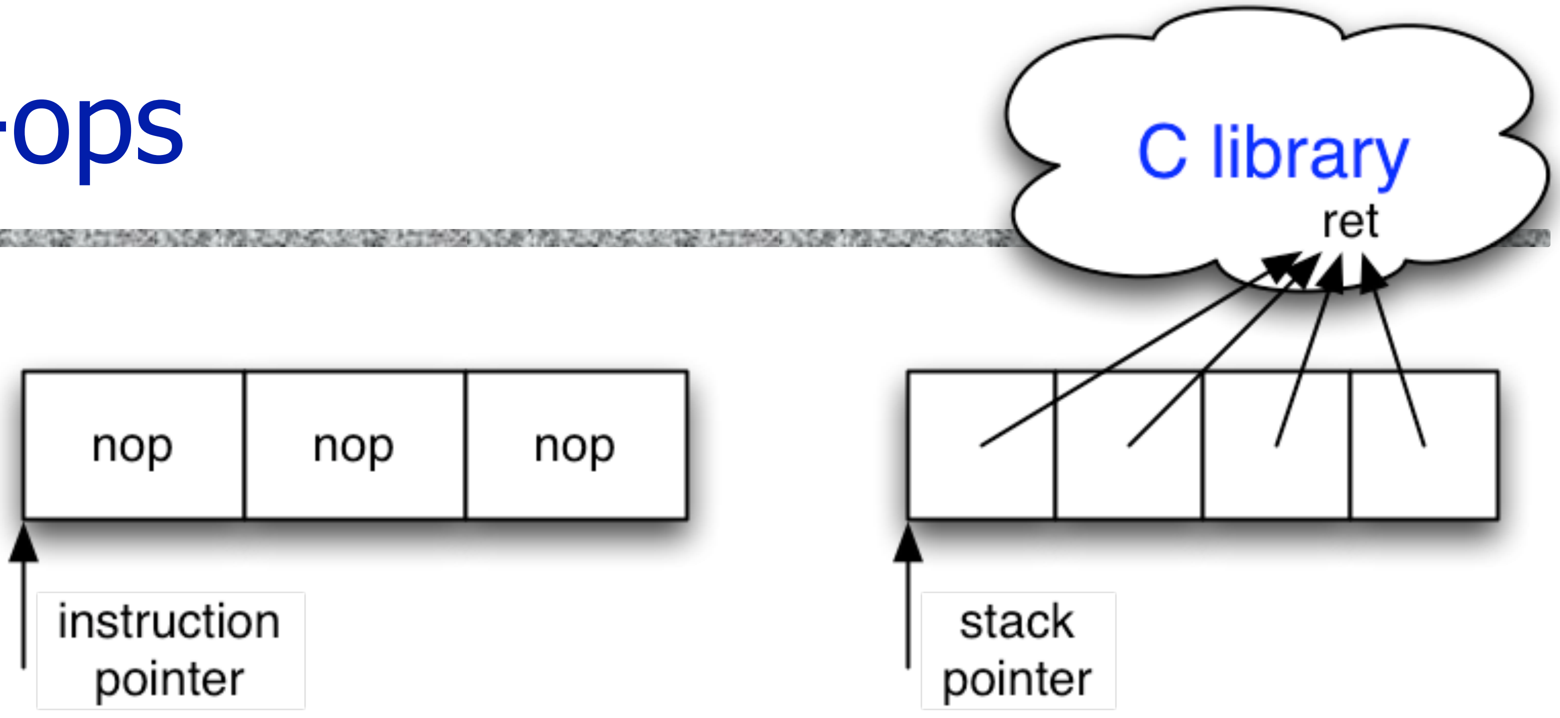
- Instruction pointer (EIP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

Return-Oriented Programming



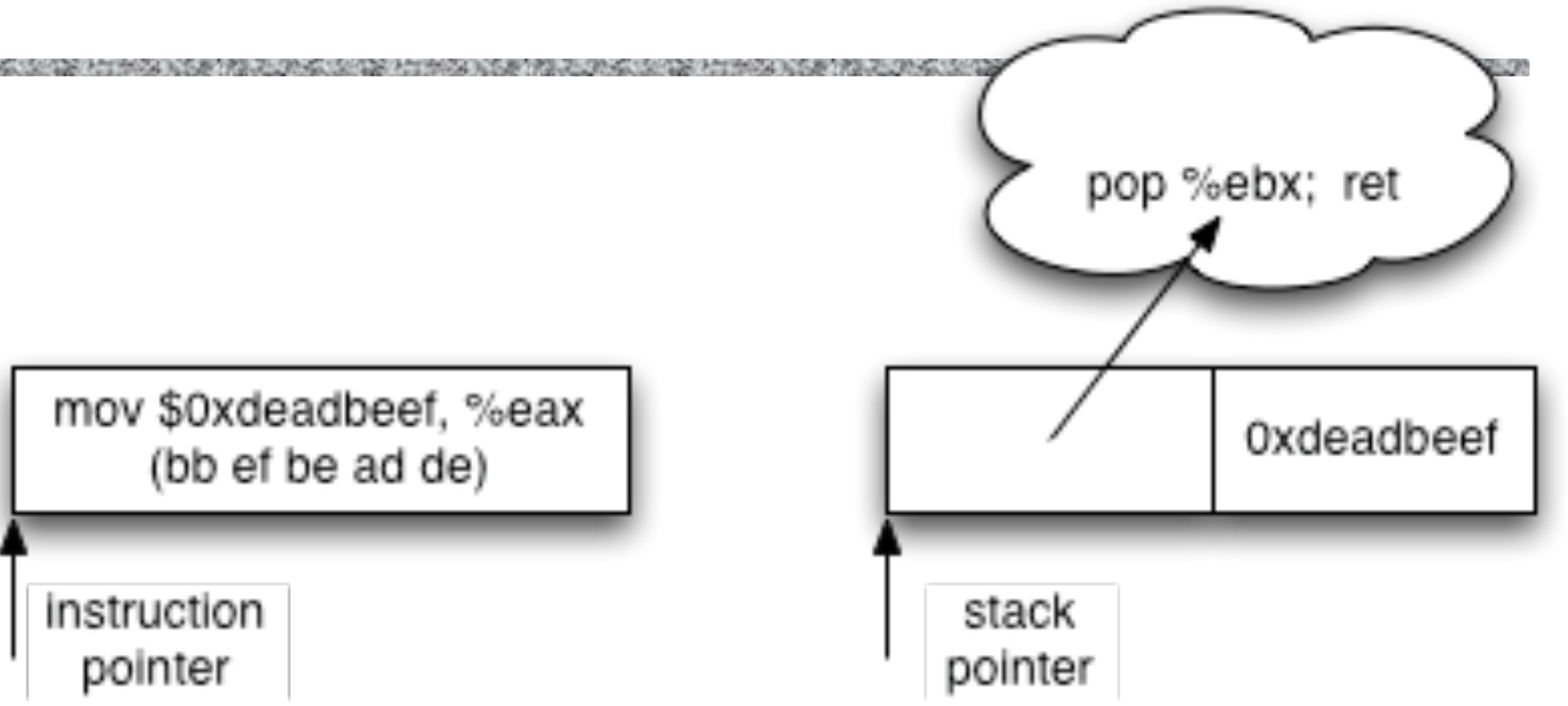
- Stack pointer (ESP) determines which instruction sequence to fetch and execute
- Processor doesn't automatically increment ESP
  - But the RET at end of each instruction sequence does

No-ops



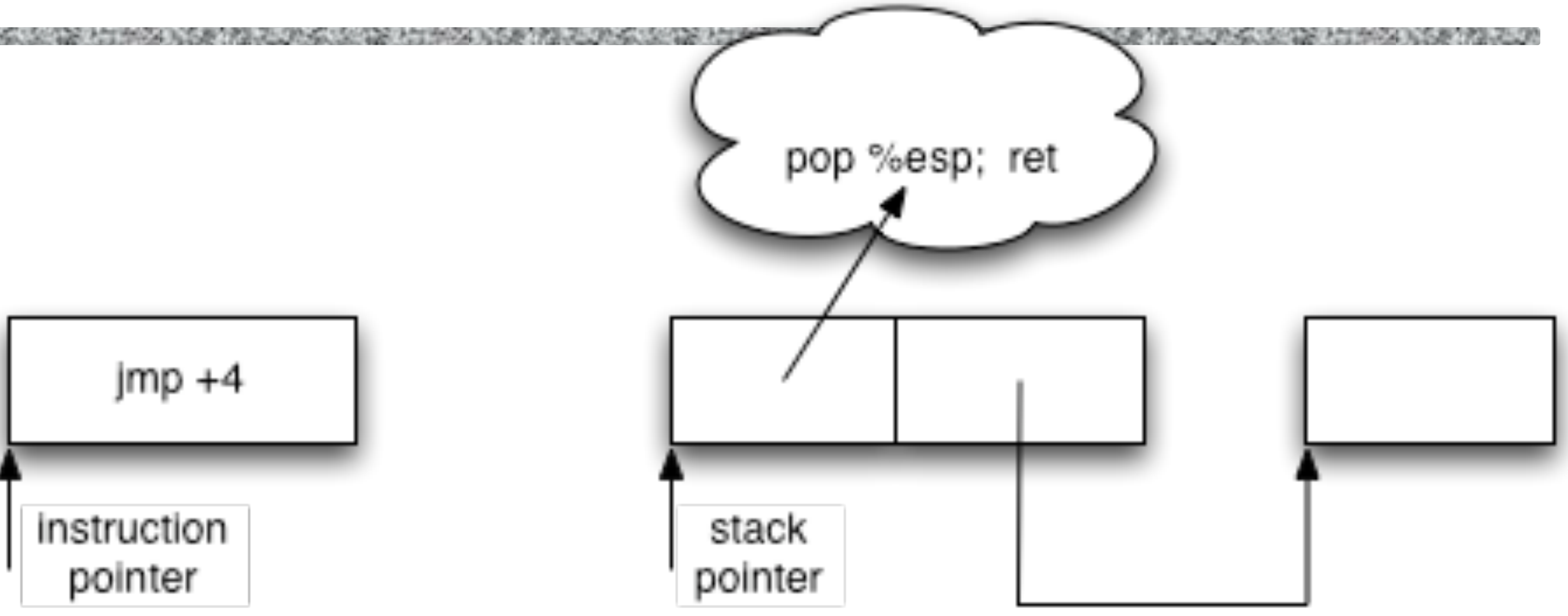
- No-op instruction does nothing but advance EIP
- Return-oriented equivalent
  - Point to return instruction
  - Advances ESP
- Useful -- like a NOP sled

Immediate Constants



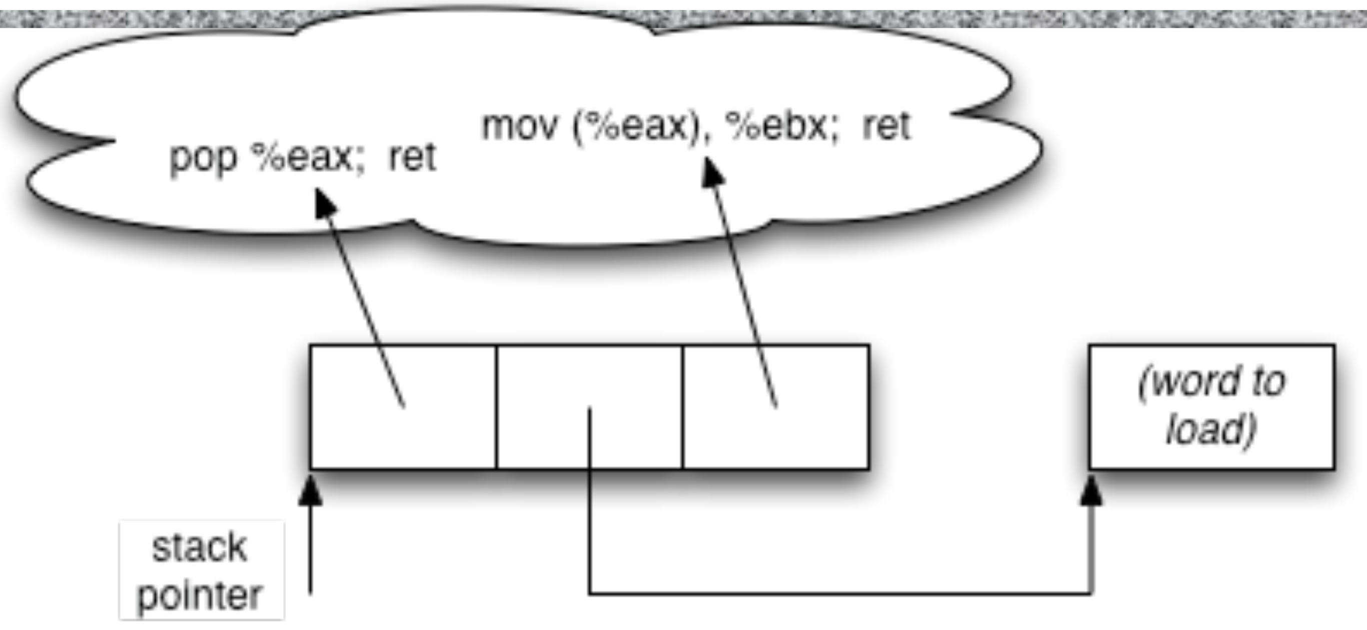
- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

Control Flow



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

Gadgets: 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

Gadget Design

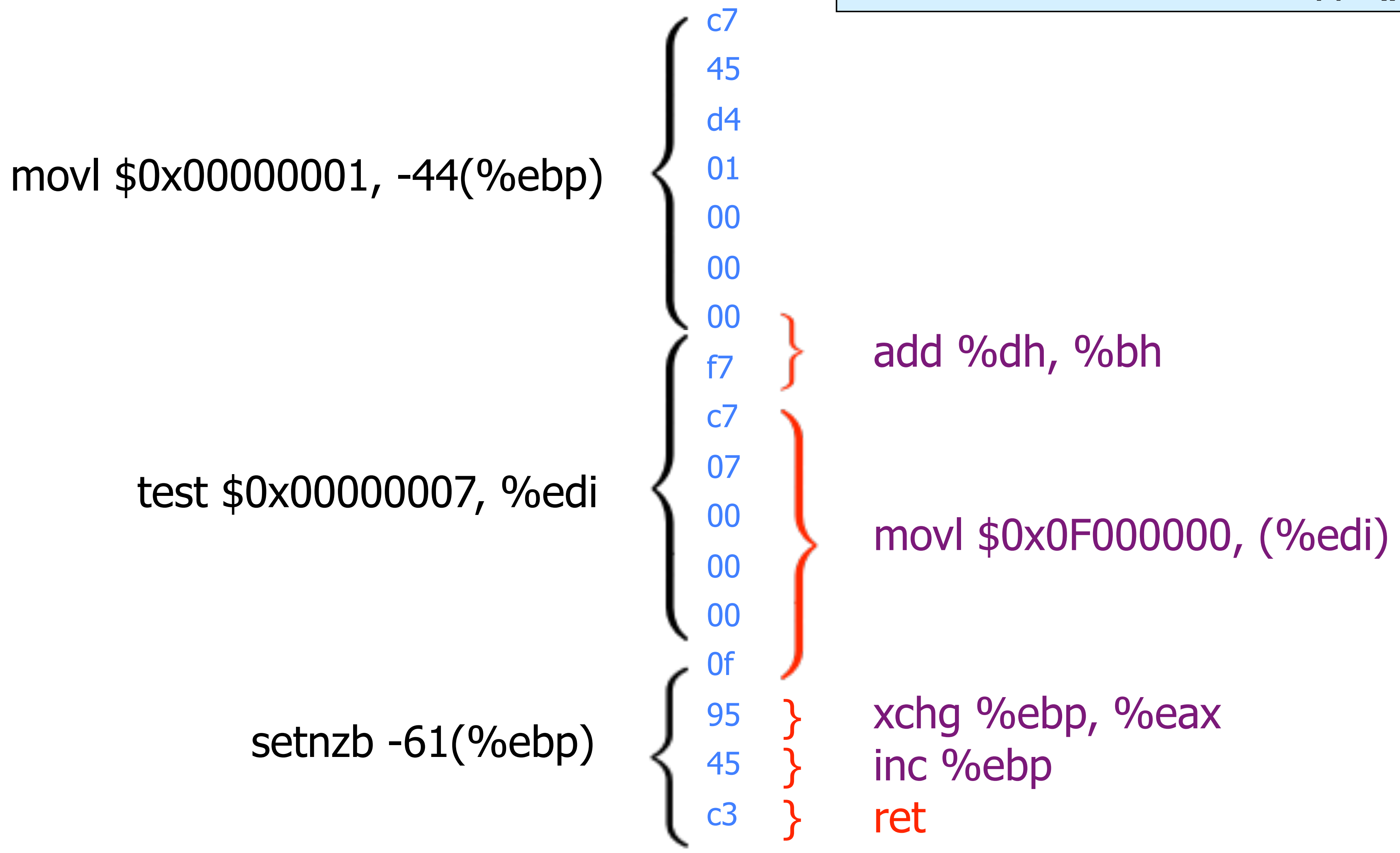
- Testbed: libc-2.3.5.so, Fedora Core 4
- Gadgets built from found code sequences:
  - Load-store, arithmetic & logic, control flow, syscalls
- Found code sequences are challenging to use!
  - Short; perform a small unit of work
  - No standard function prologue/epilogue
  - Haphazard interface, not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

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 tree

Unintended Instructions

Actual code from ecb_crypt()



```
movl $0x00000001, -44(%ebp)   c7
                              45
                              d4
                              01
                              00
                              00
                              00   add %dh, %bh
test $0x00000007, %edi        f7
                              c7   movl $0x0F000000, (%edi)
                              07
                              00
                              00
                              00
                              0f
setnzb -61(%ebp)              95   xchg %ebp, %eax
                              45   inc %ebp
                              c3   ret
```

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 (Skim)

- 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 (Skim)

- Use instruction sequences that are <u>suffixes</u> 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?
  - (Ineffective) 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
  - Inject attack code over memory containing Java objects, pass control to them

Security Systems Endangered w/ Return-oriented Programming

- W-xor-X aka DEP
  - Linux, OpenBSD, Windows XP SP2, MacOS X
  - Hardware support: AMD NX bit, Intel XD bit
- Trusted computing
- Also
  - Code signing: Xbox
  - Binary hashing: Tripwire, etc.
  - … and others

General Principles

Principles

- Check inputs

Principles

- Least privilege

Principles

- Check all return values

Principles

- Securely clear memory (passwords, keys, etc)

Principles

- Failsafe defaults

Principles

- Defense in Depth
- Also
  - Prevent
  - Detect
  - Deter

Principles

- Reduce size of TCB
- Simplicity(*)
- Modularity(*)
- (*) But: Be careful at interface boundaries

Principles

- Minimize attack surface

Principles

- Use vetted components

Principles

- Security by design

Principles (Concepts)

- Tension between security and other goals

Principles

- Open design? Open source? Closed Source?
- Different Perspectives

- Linux Kernel Backdoor Attempt: http://www.freedom-to-tinker.com/?p=472

Vulnerability Analysis and Disclosure

- What do you do if you’ve found a security problem in a real system?
- Say
  - A commercial website?
  - UW grade database?
  - Boeing 787?
  - TSA procedures?