Return Oriented Programming

CSCI 6621: Network Security

Week 11, Lecture 21: Tuesday, 04/04/2011

Daniel Bilar
University of New Orleans
Department of Computer Science
Spring 2011

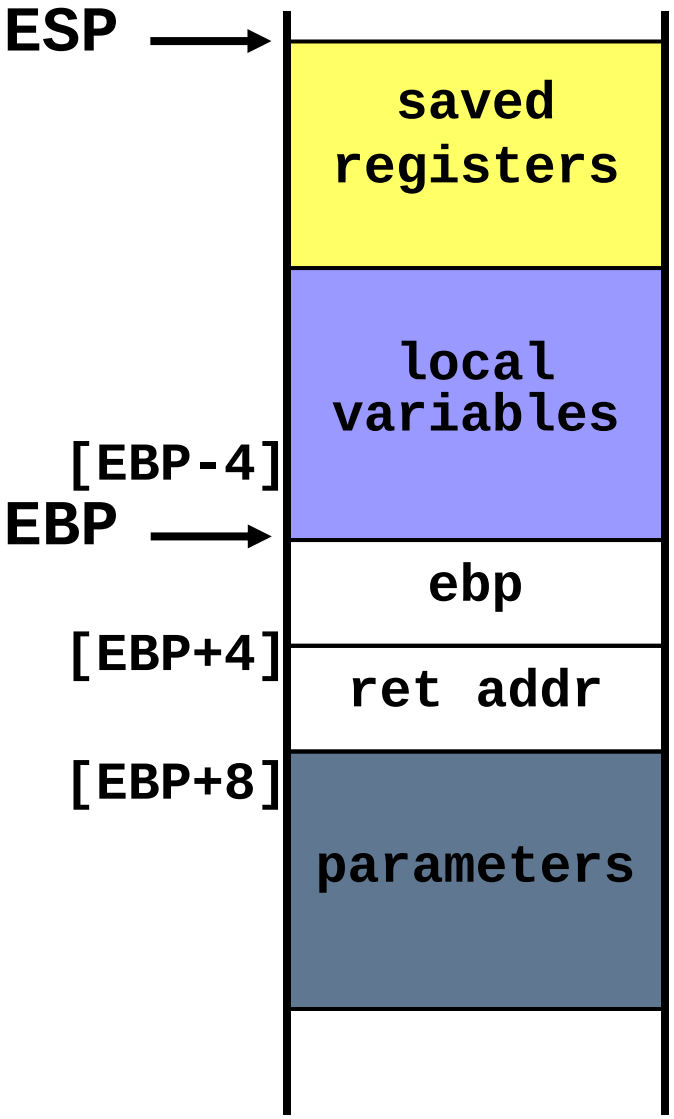
Goals today

- Review: Buffer overflow, format string
- Return Oriented Programming
  - Chain together sequences (‘gadgets’) ending in RET
  - Can use good code chunks as ‘alphabet’, string together to get for bad code
    - Some similarities to an antigram (form of anagram)
      Within earshot ‡ I won't hear this
  - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
  - Attack can perform arbitrary computation **using no injected code at all**

Some slides gratefully adapted from Shacham BH 08 presentation (UCSD)

Review: Stack frame

- Area of the stack set aside for a procedure's return address, passed parameters, saved registers, and local variables
  - Also known as an activation record
- Created by the following steps
  1. Calling procedure pushes arguments on the stack and calls the procedure.
  2. The subroutine is called, causing **the return address to be pushed on the stack**.
  3. The called procedure pushes EBP on the stack, and sets EBP to ESP.
  4. If local variables are needed, a constant is subtracted from ESP to make room on the stack.
  5. The registers needed to be saved are pushed.



Review: Buffer Overflow Causes and Cures

- Typical memory exploit involves **bending pointer** and **code injection**
  - Put malicious code at a predictable location in memory, usually masquerading as data
  - Trick vulnerable program into passing control to it
    - Overwrite saved EIP, function callback pointer, etc.
- One idea: **prevent execution of untrusted code**
  - Make **stack and other data areas non-executable**
    - Can mess up useful functionality (eg: stack manipulation in ActionScript)
  - Digitally **sign** all code
    - Traditional appraoch, but many ways to circumvent, forge signatures etc
  - Ensure that all control transfers are into a trusted, approved code image

Review: W⊕X / DEP

- **Mark all writeable memory locations as non-executable**
  - Example: Microsoft's DEP (Data Execution Prevention)
  - This blocks all code injection exploits
- Hardware support
  - AMD "NX" bit, Intel "XD" bit (in post-2004 CPUs)
  - Makes 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 (vital and almost certainly linked-to library like libc routine)
  - you arrange memory to look like arguments before hand
- Does not look like a huge threat
  - Attacker cannot execute arbitrary code
  - … especially if system() is not available

return-to-libc 2.0

- Idea
  - 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

Mounting a ROP attack

- Remember: Fun starts with control of ONE pointer that is and directs control flow
- So need control of memory around %esp .. How .. Couple of options :

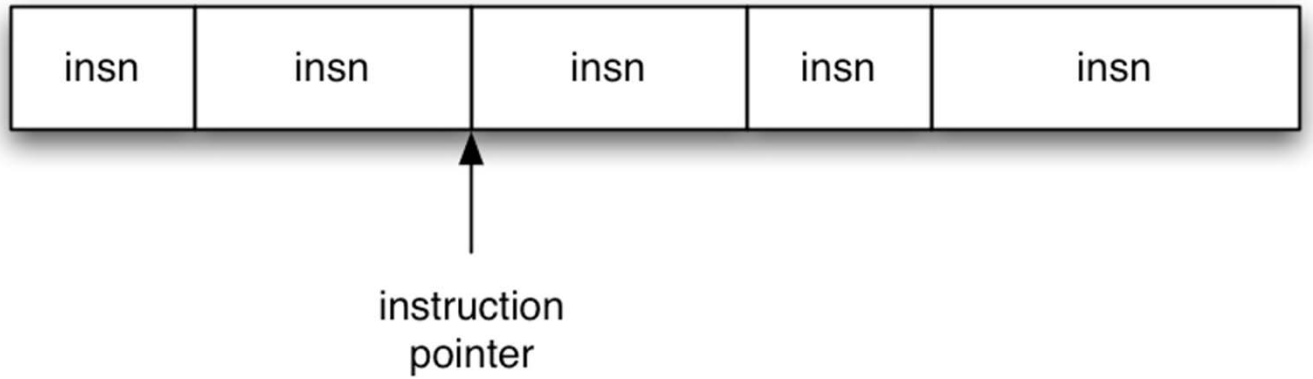
1. Rewrite stack
   - Buffer overflow on stack (we are not *executing* the code, we are just writing to the stack)
   - Format string vuln to rewrite stack contents
2. Move stack:
   - Overwrite saved frame pointer on stack; on leave/ret, move %esp to area under attacker control
   - Overflow function pointer to a register spring for %esp:
     - set or modify %esp from an attacker-controlled register then return

Chaining RETs for Fun and Profit

[Shacham et al]

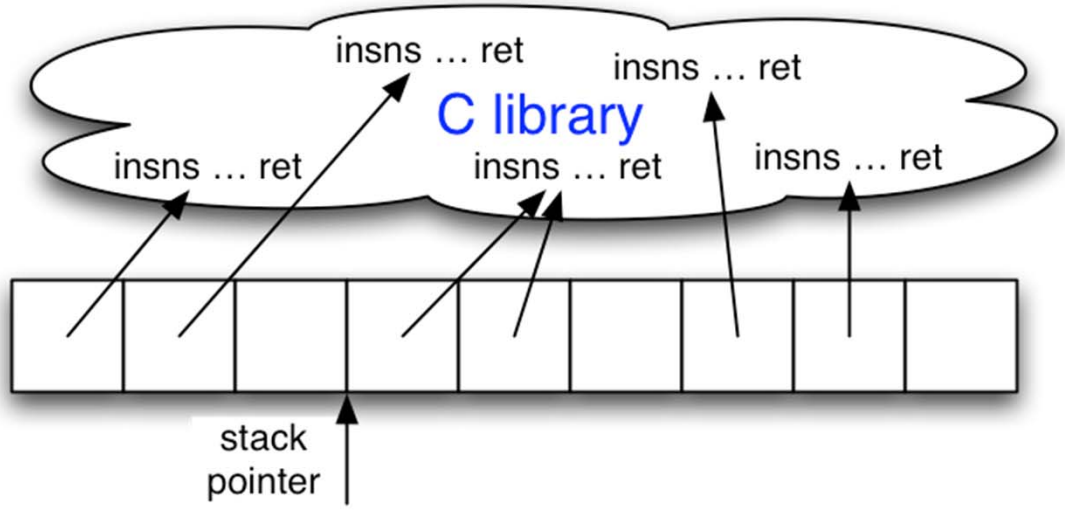
- Can chain together sequences ending in RET
  - Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" (2005)
- What is this good for?
- Answer [Shacham et al.]: everything
  - Turing-complete language
  - Build "gadgets" for load-store, arithmetic, logic, control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

Ordinary Programming



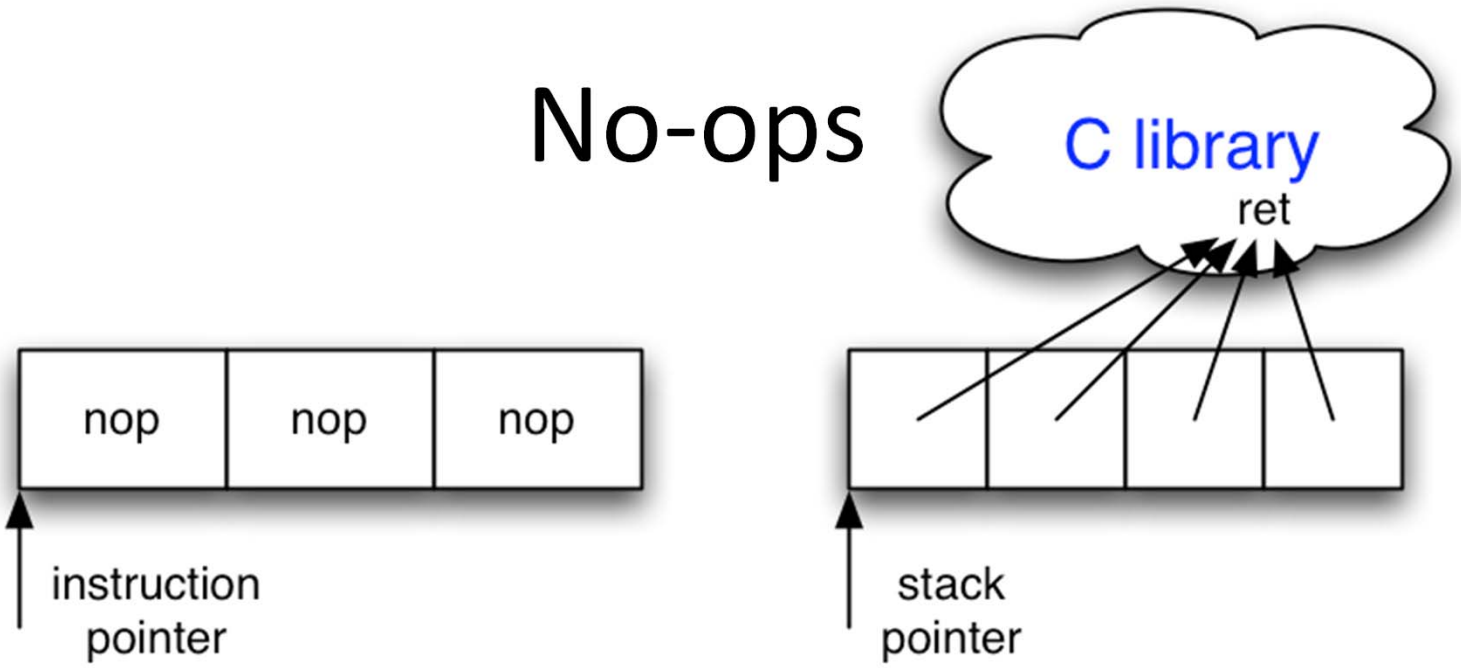
- Instruction pointer (EIP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

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

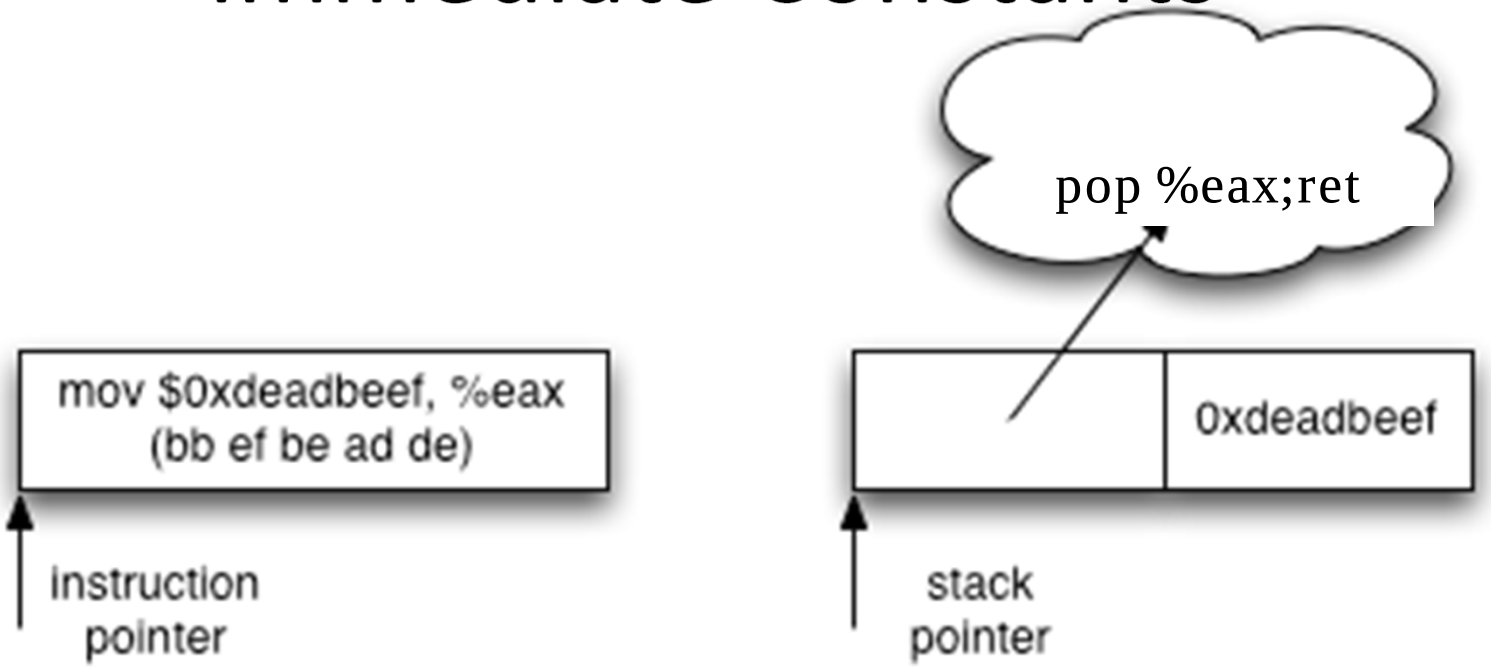


- Easy case, let's do the same thing as a noop
- A noop instruction does nothing but advance EIP

Return-oriented equivalent is

- Point to return instruction
- Advances ESP

Immediate Constants

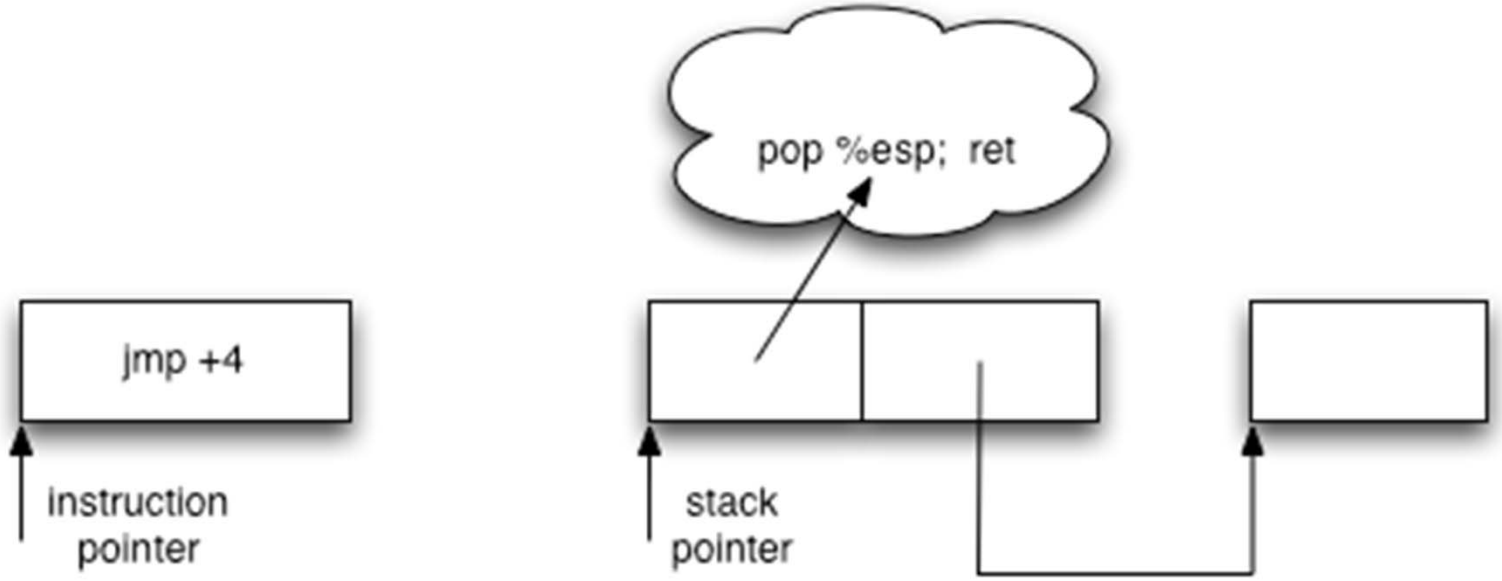


- Instructions can encode constants

Return-oriented equivalent

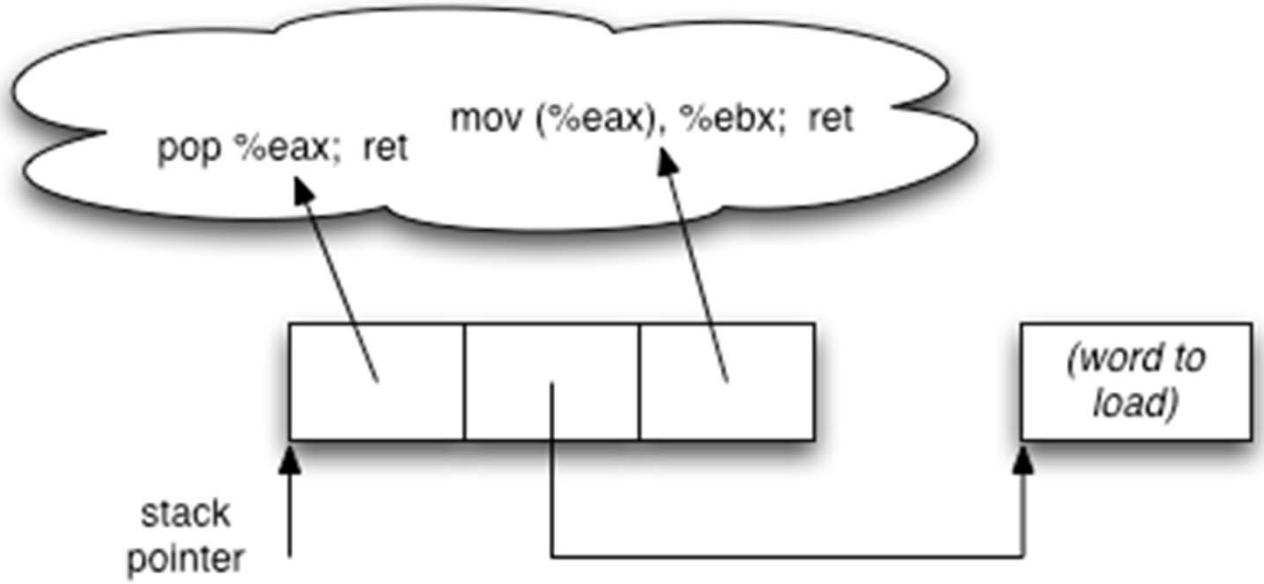
- Store on the stack
- Pop into register to use

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: Want to load from memory into register
  - Load address of source word into EAX
  - Load memory at (EAX) into EBX

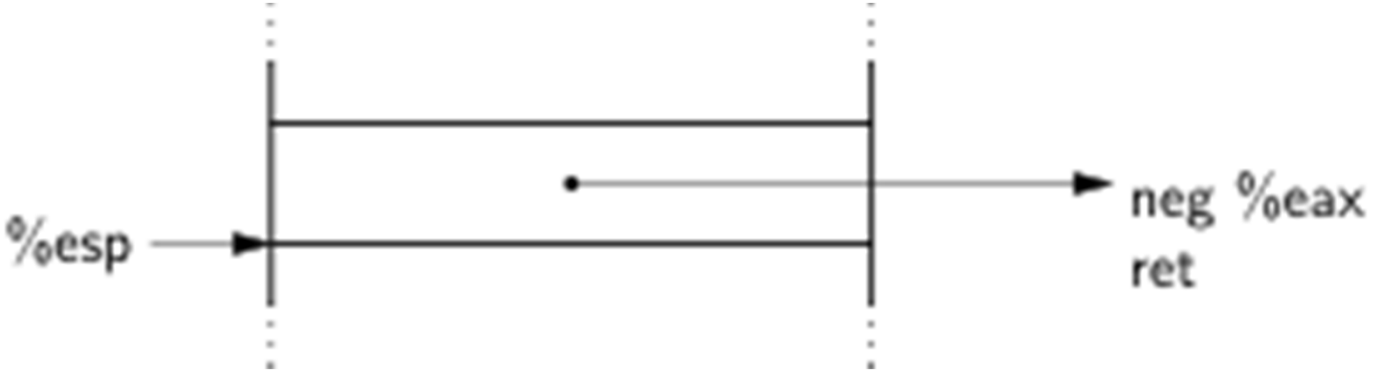
Gadget Design (2007)

- 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
  - Some convenient instructions not always available
- Build actually translation compiler c code into gadget in 2008 2009 .. Muuuch easier now

Conditional Jumps

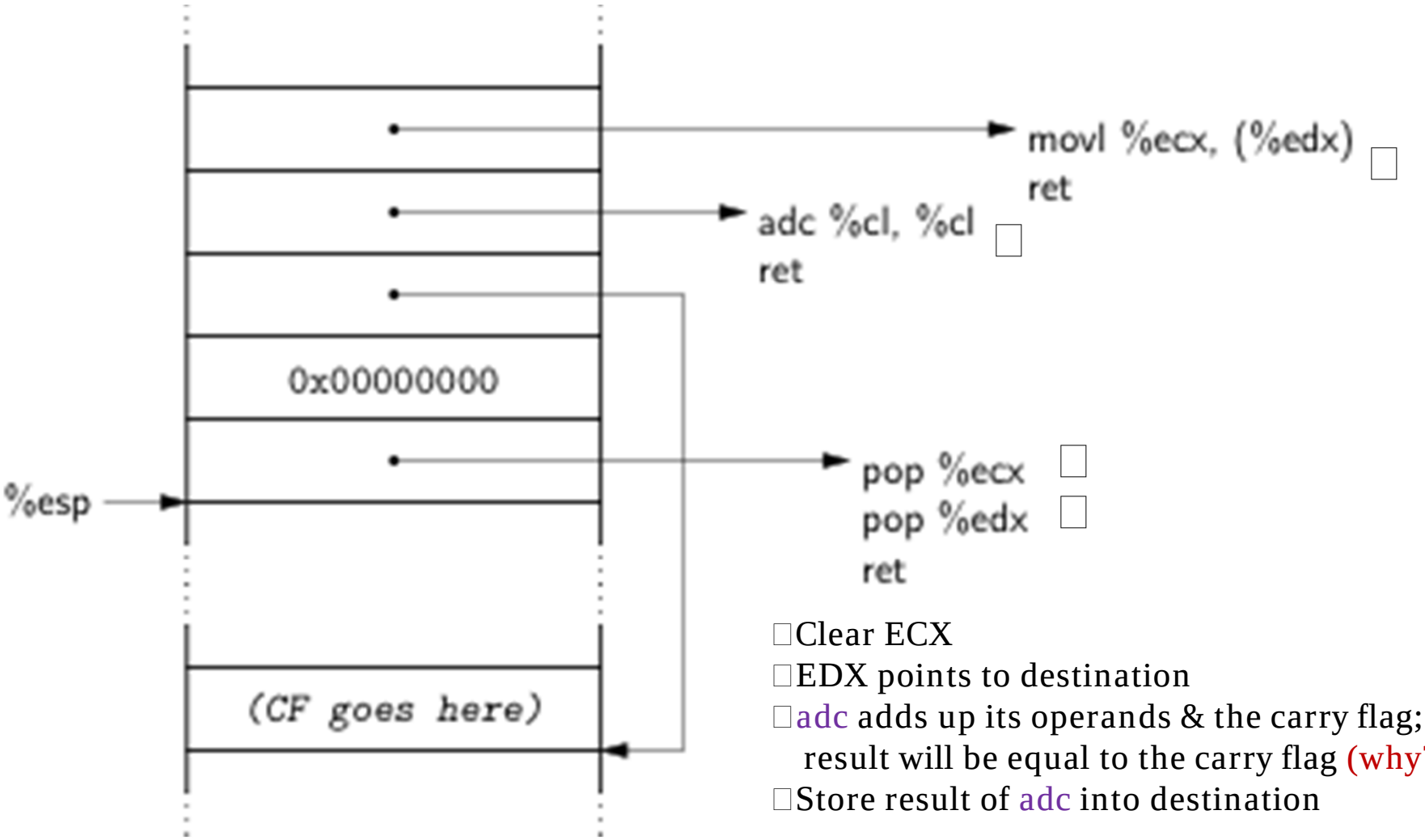
- cmp compares operands and sets a number of flags in the EFLAGS register
  - Luckily, many other ops set EFLAGS as a side effect
- jcc jumps when flags satisfy certain conditions
  - But this causes a change in EIP… not useful (why?)
- Need conditional change in stack pointer (ESP)
- Strategy on ROP:
  - 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
- Involved process! Let's see how it is done ..

Phase 1: Perform Comparison



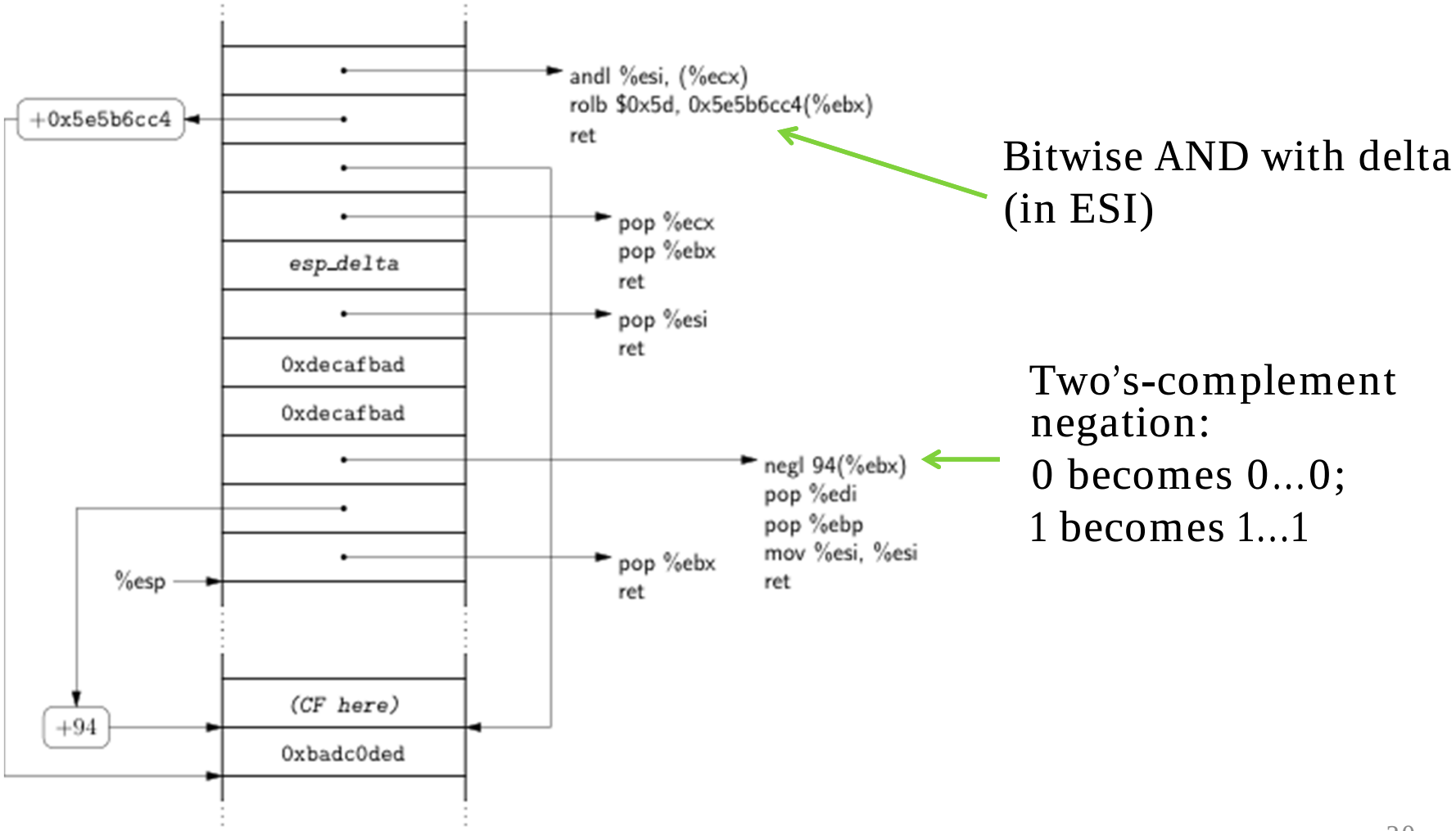
- neg calculates two's complement
  - As a side effect, sets carry flag (CF) if the argument is nonzero
- Use this to test for equality
- sub is similar, use to test if one number is greater than another

Phase 2: Store 1-or-0 to Memory

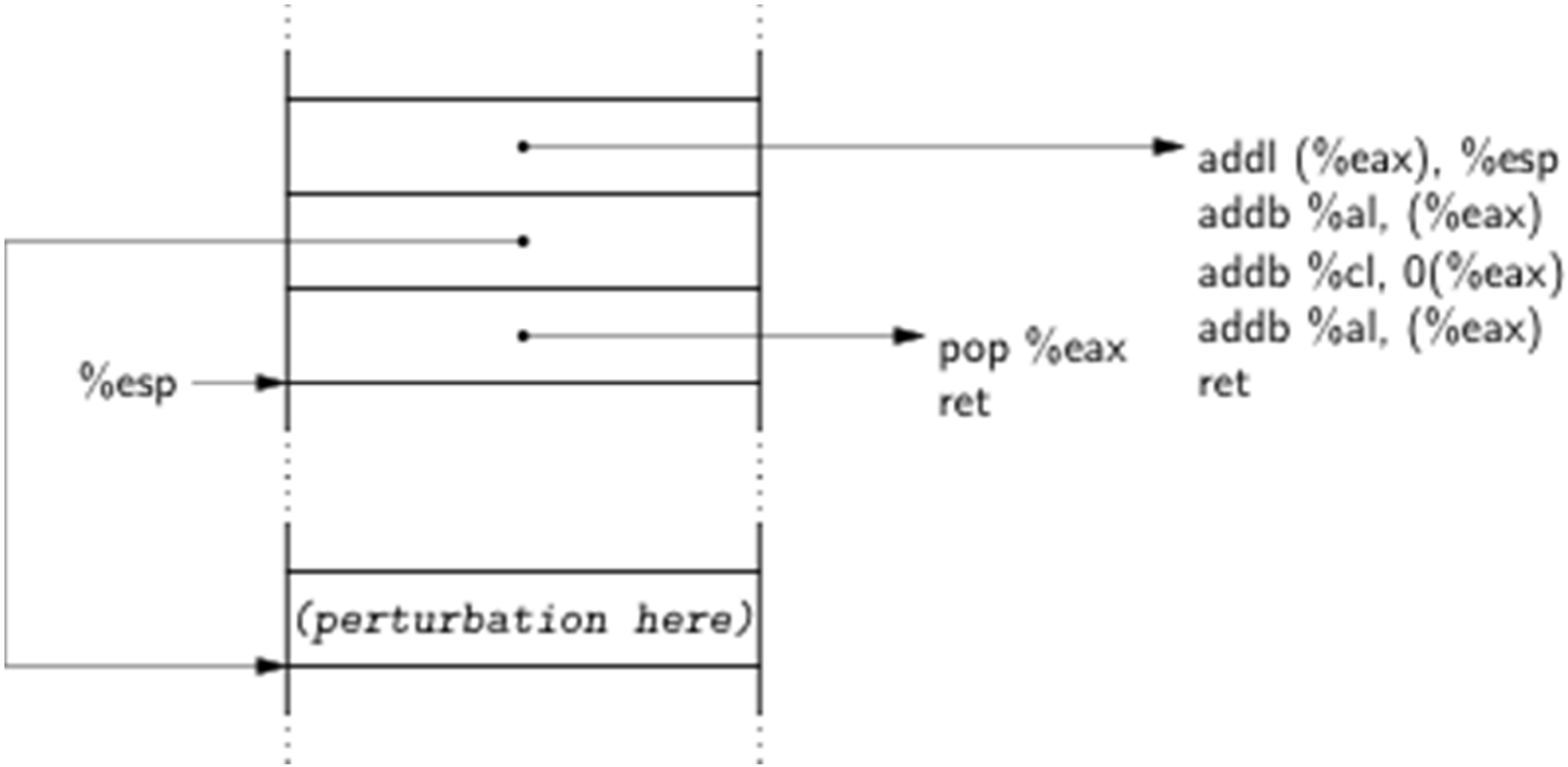


- Clear ECX
- EDX points to destination
- adc adds up its operands & the carry flag; result will be equal to the carry flag (why?)
- Store result of adc into destination

Phase 3: Compute Delta-or-Zero



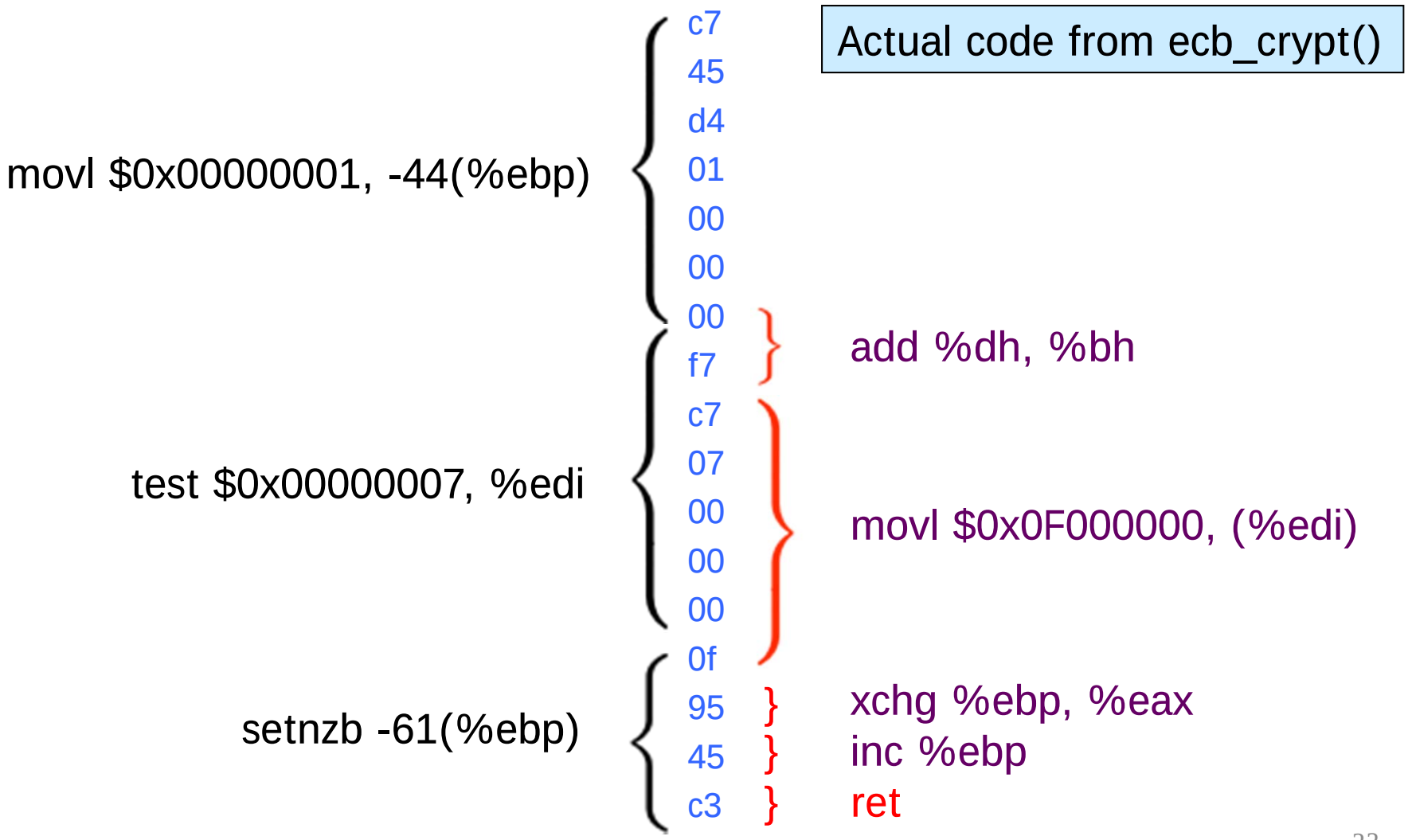
Phase 4: Perturb ESP by Delta



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 instruction sequences in a trie

“New” Parsing
Unintended instructions



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

- Testbed: Solaris 10 libc (1.3 MB)
- 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 .. amazing

Summary

- Cat and mouse game
  - Preventing execution of foreign code not enough
  - Can use good code as ‘alphabet’, string together to get bad code through RETs
    - Some similarities to an antigram (form of anagram)

      Within earshot ‡ I won't hear this
  - Can one use RET frequency to detect this?
- Son of a gun: Shacham (2010) showed you cannot use this detection approach
  - Many RET-like sequences: pop %eax; jmp %eax
    - Indirect register jumps etc

For next Thursday

- Review notes, handouts