Introduction to x86 disassembly

Defcon 25
2017
DazzleCat Duo
Architecture Overview
Computer Architecture

• CPU (Central Processing Unit)
  – Processes information
  – ALU (Arithmetic logic unit)
    • Does math
  – Registers
    • Store data (very fast)
    • Register size: 1 word
    • Generally *named*, rather than *addressed*
  – Control unit
    • Executes code
Computer Architecture

• Registers vs. Memory
  • Registers serve the same purpose as memory
    – They store data
    – Memory
      • Moderate access speed
      • Cheap
      • Lots
    – Registers
      • Fast
      • Expensive
      • Few
  • Your program/data/etc sit in memory, while registers are used to process very small pieces at a time
Abstractions

• All of this is normally abstracted away from the programmer

• The Operating System manages...
  – Processes
    • Makes it look like your program has control of the processor
  – Memory
    • Makes it look like your process has it
  – Files
    • Makes them look like a sequence of bytes
Abstractions

• But none of these things are true
• Goal of learning assembly is to start seeing the world as it really is
Assembly

• Everything the CPU does is through digital logic
  – On/Off, 1/0

• Including running your program

• The series of bits that control the CPU is *machine code*
  – A bunch of numbers
  – Define a set of instructions to run
Assembly

• The machine code for a standard “hello world”:
  55 89 e5 83 e4 f0 83 ec 10 b8 b0 84 04 08 89 04
  24 e8 1a ff ff ff b8 00 00 00 00 c9 c3 90
• This is a series of instructions for the processor to execute
• It flips the right transistors to calculate information, fetch data from memory, send signals to the system buses, communicate with the graphics card, and print out “hello world”
  – With help from additional machine code
Assembly

• We want to directly control the CPU to leverage its full power
  – But we don’t want to write a bunch of numbers that we can’t hope to understand

• *Assembly* is a shorthand, more legible version of machine code
  – Uses mnemonics to save us from memorizing which numbers do what
  – “sub” (subtract) instead of 0x83
  – “add” (add) instead of 0x81
# Assembly

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>push %ebp</td>
</tr>
<tr>
<td>89 e5</td>
<td>mov %esp,%ebp</td>
</tr>
<tr>
<td>83 e4 f0</td>
<td>and $0xfffffffff0,%esp</td>
</tr>
<tr>
<td>83 ec 10</td>
<td>sub $0x10,%esp</td>
</tr>
<tr>
<td>b8 b0 84 04 08</td>
<td>mov $0x80484b0,%eax</td>
</tr>
<tr>
<td>89 04 24</td>
<td>mov %eax,(%esp)</td>
</tr>
<tr>
<td>e8 1a ff ff ff</td>
<td>call 80482f4</td>
</tr>
<tr>
<td>b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>c9</td>
<td>leave</td>
</tr>
<tr>
<td>c3</td>
<td>ret</td>
</tr>
<tr>
<td>90</td>
<td>nop</td>
</tr>
</tbody>
</table>
Assembly

• Writing in pure machine code is fun, and has its uses, but is difficult and uncommon
• Much more practical to write in assembly
• An assembler is a tool that translates from assembly to machine code; this process is called assembling
• A disassembler is a tool that translates from machine code to assembly; this process is called disassembling
Compilation Process

• Source code is compiled into assembly code
• Assembly code is assembled into machine code
• Compilers have been doing all this for you
Instruction Set Architecture

• The *Instruction Set Architecture (ISA)* defines
  – Processor registers
    • One register, or 200? 8 bits, or 128?
  – Address and data format
    • Do I grab a byte from memory at a time? Or 500?
  – Machine instructions
    • Can I add and subtract? Check for equality? Halt?
• Indirectly defines the *assembly language*
  – What low level instructions we have available, what those instructions do
Computer Architecture

• Collectively, the instruction set architecture and microarchitecture define the *computer architecture*

• There are...
  – Thousands of instruction set architectures
  – Thousands of microarchitectures
  – Thousands of computer architectures
Computer Architecture

• Architectures can usually be broadly divided into two categories
  – Reduced Instruction Set Computing (RISC)
  – Complex Instruction Set Computing (CISC)
RISC vs. CISC

- **RISC**
  - Small set of simple instructions
  - Generally...
    - Cheaper to create
    - Easier to design
    - Lower power consumption
    - Physically smaller

- **CISC**
  - Large set of powerful instructions
  - Generally...
    - More expensive
    - Hard to design
    - Higher power requirements
    - Physically larger
RISC vs. CISC

• Hypothetical example
  – Multiply by 5, RISC vs. CISC

• CISC:
  – mul [100], 5

• RISC:
  – load r0, [100]
  – mov r1, r0
  – add r1, r0
  – add r1, r0
  – add r1, r0
  – add r1, r0
  – mov [100], r1
RISC vs. CISC

- Neither RISC nor CISC is better or worse than the other
  - Both have advantages, and disadvantages
  - A CISC instruction may take 100 RISC instructions to implement
  - But a CISC instruction may run at $1/200^{th}$ the speed of the RISC instructions
  - Or consume 1000x the power
  - Or take a year to design
(Some of) The Major Players

• RISC
  – ARM (examples: phones, tablets)
  – MIPS (examples: embedded systems, routers)
  – PowerPC (examples: original Macs, Xbox)

• CISC
  – x86 (examples: consumer computers)
  – Motorola 68k (examples: early PCs, consoles)
Introduction to x86
Introduction to x86

• Why x86?
  – Can build, run, and play with on your own computer
  – Extremely popular, billions of systems, market dominance
  – Core of familiar operating systems (Windows, Mac, Linux)
x86

• Your laptops, desktops, workstations, servers, etc, all use the x86 architecture
• When you buy a new processor to upgrade your computer, that’s an x86 processor
• Makes it an ideal choice for studying assembly and computer architecture
History of x86

• Intel 8080
  – 8 bit microprocessor, introduced in 1974

• Intel 8086
  – 16 bit microprocessor, introduced in 1978

• Intel 80386
  – 32 bit microprocessor, introduced in 1985

• Intel Prescott, AMD Opteron and Athlon 64
  – 64 bit microprocessor, introduced in 2003/2004

History of x86

• Goal of design: backwards compatibility
  – Every generation adds new features
    • But doesn’t break or remove any of the old
    • Even when the old features were later determined to be useless/broken/etc
  – Code that runs on the original 8086 processor can run unmodified on the latest 9th generation architectures
• Has resulted in an immense, complex, interesting architecture
A Complex Architecture

• Intel Software Developer’s manual...


• 4000 pages, doesn’t even begin to scratch the surface

• Goal in class: give you the basics
x86

• Today, “x86” generally refers to all architectures based off of the original 8086
  – The 8086, which contains the 16 bit architecture
  – The 80286, which contains the 32 bit architecture and the 16 bit architecture
  – The 80886, which contain a 64 bit architecture, 32 bit architecture, and 16 bit architecture

• The term “x64” refers specifically to the 64 bit version of the x86 architecture

• We will study the 32 bit x86, since it is the most universal
Assembly Syntax
Rivals

• Two main branches of x86 syntax
  – AT&T
    • Used by gcc
  – Intel
    • Used by Intel
• They both have their pros and cons
• Then hundreds of smaller variations specific to an assembler
Assembler Syntax

• In this class:
  – The assembler is NASM
    • The “netwide assembler”
    • Extremely popular
    • Very powerful
    • Very flexible
  – So we’ll teach NASM’s x86 syntax
    • Uses Intel syntax
Assembler Syntax

• Almost universally true in assembly, and with NASM
  – Lines do not end in a semi-colon
  – Semi-colons are used to start a single line comment
  – instruction ; comment
x86 Registers
Registers

• Registers are how the processor stores information

• The processor can access memory, but since the system’s memory is not part of the actual processor, this is extremely slow

• Registers are contained in the actual processor, they are very fast (access at the same speed as the processor)
Registers

• You can think of registers as 32 bit variables
  – Each register has its own name
  – Can be modified, etc

• But there are a very limited number of registers
  – They must be shared by the whole program
  – When they run out, they need to store their information back to memory

  – Typical execution:
    • Fetch data from memory, store in registers
    • Work with data
    • Save data back to memory
    • Repeat
x86 Registers

Source: http://en.wikipedia.org/wiki/File:Table_of_x86_Registers.png
x86 Registers

• Fortunately, you do not need to know all those
• The ones you will need to know...
• x86 GPRs (General Purpose Registers):
  – eax, ebx, ecx, edx, esi, edi, ebp, esp
• x86 SPRs:
  – eip, eflags
x86 Registers

• Example, accessing pieces of the eax register

EAX (32 bits)

MSB

AH (8 bits) AL (8 bits)

LSB

AX (16 bits)
EBP/ESP

• Intel classifies EBP and ESP as GPRs
• But many people would consider them SPRs
• GPRs are used in arithmetic, memory accesses, etc
• SPRs have some other special purpose
• EBP/ESP control the stack, so they have another special purpose
• You would generally not modify them like you would EAX/EBX/ECX/EDX
EIP

• 32 bit SPR
• The “instruction pointer” register
• Stores the address of the next instruction to execute
• ip: low 16 bits of eip
x86 Memory Access
Accessing Memory

• In assembly, memory is accessed using [ ] notation

• Examples:
  – [ 0x12345678 ]
    • Access the value stored at memory address 0x12345678
  – [ eax ]
    • Access the value stored at the memory pointed to by eax
x86 Instructions
x86 Instructions

- Arithmetic
  - add
  - sub
  - mul
  - inc
  - dec
  - and
  - or
  - xor
  - Not

- Stack
  - call
  - return
  - push
  - pop

- Data movement:
  - mov

- Execution flow
  - jmp
  - Conditional jumps

- Comparison
  - test
  - cmp

- Other
  - lea
  - nop
x86 Instructions

• There are hundreds more, but those are the basics we need for this class

• Even this might seem like a lot, but when you think of all the operators (+, -, *, /, %, &&, ||, &, |, ^, !, ~, <, >, >=, <=, ==, ., ->, etc) and keywords (if, else, switch, while, do, case, break, continue, for, etc) you know for any other language, this is trivial
mov

• Move data from one location (memory, register, etc) to another
• **Syntax:** `mov destination, source`
mov Examples

- `mov eax, 5`
  - Store the value 5 into eax
- `mov eax, [1]`
  - Copy the 32 bit value at memory address 1 into eax
- `mov dx, [0x100]`
  - Copy the 16 bit value at memory address 0x100 into dx
- `mov ecx, eax`
  - Copy the contents of eax into ecx
- `mov [1984], bl`
  - Store the 8 bit value in bl to memory address 1984
- `mov [eax], cx`
  - Store the 16 bit value in cx to the memory pointed to by eax; e.g. if eax is 0x777, store cx to location 0x777 in memory
inc, dec

• Increment, decrement by 1
• Syntax:
  - inc register
  - inc [ memory ]
  - dec register
  - dec [ memory ]
inc, dec Examples

• inc eax
  – Increment the eax register by 1

• dec dx
  – Decrement the dx register by 1

• dec dword [ 0x11223344 ]
  – Decrement the 32 bit value at 0x11223344 by 1

• inc word [ ecx ]
  – Increment the 16 bit value pointed to by ecx by 1
add, sub

• Add and subtract

• Syntax
  – add destination, value
  – sub destination, value

• Destination can be a register or memory

• Value can be a register, memory, or immediate

• Note: operands must all be same size
  – add eax, bx is invalid
add, sub Examples

• **add eax, ebx**
  – Add ebx to eax, store result in eax

• **sub ecx, [ 100 ]**
  – Subtract the 32 bit value at address 100 from ecx, store the result in ecx
  – Note that the memory access is implied to be 32 bits, there is no need to specify “dword”

• **add dword [ edx ], 100**
  – Add 100 to the 32 bit value pointed by edx
  – Note that the *address* is implied to be 32 bits (edx), but the data size must be specified
mul

• Multiply eax by operand, store result in edx:eax
  – edx: high 32 bits of result
  – eax: low 32 bits of result

• Syntax:
  – mul [ memory ]
  – mul register

• mul always uses the eax register as a source
• And always stores the result in edx:eax
mul Examples

• mul eax
  – edx:eax = eax * eax; (Square eax)
• mul ebx
  – edx:eax = eax * ebx;
• mul dword [ 0x555 ]
  – edx:eax = eax * (32 bit value at address 0x555)
• mul byte [ 0x123 ]
  – edx:eax = eax * (8 bit value at address 0x123)
and, or, xor

• Binary AND, OR, and XOR
• Syntax:
  - and destination, source
  - or destination, source
  - xor destination, source
• Destination can be a register or memory address
• Source can be a register, memory address, or immediate
and, or, xor Examples

• or eax, 0xffffffff
  – Set eax to all 1’s
• and dword [0xdeadbeef], 0x1
  – Mask off low bit of 32 bit value at 0xdeadbeef
• xor ecx, eax
  – ecx = ecx ^ eax
  – Evaluate exclusive or of bits in ecx and eax, store result in ecx
and, or, xor Examples

- **xor eax, eax**
  - Fastest way to clear a register in x86
  - Other ways
    - mov eax, 0
    - and eax, 0
    - sub eax, eax
  - Involve extra computation or longer machine encodings, which slow them down

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A XOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
not

• Binary NOT

• Syntax:
  – not register
  – not [ memory ]

• Retrieves the value of the operand, computes its one’s complement, and stores it back to the operand
not Examples

• not ch
  – Inverts all the bits of ch

• not dword [ 2020 ]
  – Inverts all the bits of 32 bit value at address 2020
nop

• “No operation”
• Literally does nothing
• **Syntax:** `nop`
• Compiles to exactly one byte in machine code (0x90)
• Commonly used for...
  – Timing
  – Memory alignment
  – Hazard prevention
  – Branch delay slot (RISC architectures)
  – A placeholder to be replaced later
  – Hacking (nop sleds)
  – Cracking (nop outs)
lea

- Load Effective Address
- **Syntax:** `lea destination, [ source ]`
- Computes the address of the source operand, and places it in the destination operand
- Similar to the `&` operator in C
- Often used for simple math, rather than anything to do with addresses
lea examples

• lea eax, [ 100 ]
  – Computes the effective address of [ 100 ] (which is 100) and stores it in eax

• lea ecx, [ ebx ]
  – Computes the effective address of [ ebx ] (which is ebx) and stores it in ecx
Examples

• Evaluate 0x13 * 0x100 + 0x37 using assembly

```assembly
mov eax, 0x13
mov ecx, 0x100
mul ecx
add eax, 0x37
```

Multiplies eax by ecx, saving result in edx:eax. Could not use immediate value in multiplication, needed a scratch register.
x86 reference

• One of my favorite x86 references
• http://ref.x86asm.net/coder32.html
Conditional Codes

- Eflags register contains the current state of flags AKA conditional codes
- There are 9 conditional codes on x86
- Flags are used to track the outcome of operations
- Flags are used to conditional execute code

- CF, PF, ZF, SF, OF, AF, TF, IF, DF
Condition Flags

• The most useful two:
  – CF – Carry – Last arithmetic resulted in a carry
  – ZF – Zero – Last arithmetic/logical operation resulted in a zero
Examples

• Rewrite the following C code in assembly:
  – int i = 7; char j = 5; int k = i + j;

• Assume:
  – i is at address 100
  – j is at address 200
  – k is at address 300
int i = 7; char j = 5; int k = i + j;

mov dword [ 100 ], 7 ; set i
mov byte [ 200 ], 5  ; set j

mov eax, [ 100 ]   ; load i into eax
xor ebx, ebx       ; zero ebx
mov bl, [ 200 ]    ; load j into ebx

add eax, ebx       ; add ebx to eax, store in eax

mov [ 300 ], eax    ; save result to k
Examples

• Rewrite the following C code in assembly:
  – int i = 7; char j = 5; int k = i * i + j * j;

• Assume:
  – i is at address 100
  – j is at address 200
  – k is at address 300
int i = 7; char j = 5; int k = i * i + j * j;

```
mov dword [ 100 ], 7 ; set i
mov byte [ 200 ], 5 ; set j

mov ecx, [ 100 ] ; load i into ecx
xor ebx, ebx ; zero ebx
mov bl, [ 200 ] ; load j into ebx

mov eax, ecx ; copy ecx into eax (eax = ecx = i)
mul ecx ; multiply ecx by eax, store result in eax
mov ecx, eax ; save result back to ecx to free up eax

mov eax, ebx ; copy ebx into eax (eax = ebx = j)
mul ebx ; multiply ebx by eax, store result in eax

add eax, ecx ; add ecx to eax, store result in eax
mov [ 300 ], eax ; save final value to k
```
Writing your own

- name your code file `.asm`
- in Linux
  ```
  //assemble the code into an object file
  nasm -f elf myasm.asm
  //link the object file
  ld -melf_i386 myasm.o -o myasm.out
  //running the output
  ./myasm.out
  ```
Writing your own

• assemble the code into an object file
  
  `nasm -f elf myasm.asm`

• This uses the nasm assembler to translate the assembly code into machine code, with additional information that can later be used to create an executable file
Writing your own

• link the object file

    ld -melf_i386 myasm.o -o myasm.out

    – This creates an executable file called “myasm.out” in your directory.
    – -melf_i386 tells ld to link for a x86 elf
Writing your own

- Disassembling
  - `objdump -D -Mintel myasm.out`
  - dumps code section & data section
  - `-Mintel tells it to use intel syntax`
Assembly Makefile

all: myasm.o

myasm.o: myasm.asm
    nasm -f elf myasm.asm
    ld -melf_i386 myasm.o

clean:
    rm myasm.o a.out
Makefile

all: printreg-shift.o

printreg-shift.o: printreg-shift.asm
    nasm -f elf32 -g printreg-shift.asm
    ld -melf_i386 -g printreg-shift.o -o printreg-shift.out

clean:
    rm printreg-shift.o printreg-shift.out

• Once we get into more complicated assembly you won’t be able to do much debugging if you don’t include extra debug symbols
NASM sections

section .text ; Section for all code

global _start ; Exports start method

_start: ; Code Starts Executing here!

........ CODE HERE

section .data ; Section for global data

........ VARIABLES HERE
Declaring Variables

- General form: Name <granularity> <initial value>
- db = 1 byte
- dw = 2 bytes
- dd = 4 bytes
- dq = 8 bytes

```section .data
v1 db 0x55 ; just the byte 0x55
v2 db 0x55,0x56,0x57 ; three bytes in succession
v3 db 'a',0x55 ; character constants are OK
v4 db 'hello',13,10,'$' ; so are string constants
v5 dw 0x1234 ; 0x34 0x12
v6 dw 'a' ; 0x61 0x00
v7 dw 'ab' ; 0x61 0x62
```
Declaring Variables

- General form: Name <granularity> <initial value>
- db = 1 byte
- dw = 2 bytes
- dd = 4 bytes
- dq = 8 bytes

```
section .data
v8 dw 'abc' ; 0x61 0x62 0x63 0x00 (string)
v9 dd 0x12345678 ; 0x78 0x56 0x34 0x12
v10 dd 1.234567e20 ; floating-point constant
v11 dq 0x123456789abcdef0 ; eight byte constant
v12 dq 1.234567e20 ; double-precision float
v13 dt 1.234567e20 ; extended-precision float
```
Debugging x86 with GDB

• Run “gdb <executable_name>”
Debugging x86 with GDB

• Tell gdb we want to look at intel assembly
• (gdb) set disassembly-flavor intel
Debugging x86 with GDB

• Show the different parts of the file
• (gdb) info files

(gdb) info files
Symbols from "/home/swagger/Documents/osu/x86/a.out".
Unix child process:
Using the running image of child process 61165.
While running this, GDB does not access memory from...
Local exec file:
'/home/swagger/Documents/osu/x86/a.out', file type elf32-i386.
Entry point: 0x80480d1
0x08048080 - 0x080480dd is .text
0x080490e0 - 0x080490e5 is .data
(gdb)
Debugging x86 with GDB

• disassemble
  – disassembles where IP currently is

• disassemble address
  – disassemble 0x8048080

• disassemble label
  – disassemble loop
  – disassemble main

• add ‘+ number’ to print number instructions
  – disassemble main +50
```assembly
loop:
    mov edx, eax ; copy the value into edx for us to do manipulations on
    mov ecx, ebx
    shl ecx, 2 ; multiply by 4
    shr edx, cl
    and edx, 0xf ; get rid of all but the bottom nibble
    cmp dl, 10 ; check if the remainder is less than 10
    jge _ascii_to_hex ; if it was greater or equal to 10 then we know its A-F
    add dl, '0' ; its a numeric digit, add '0' to convert to ascii
    jmp _ascii_to_end
_ascii_to_hex:
    add dl, '7' ; its A-F, add 0x55 which how to convert to a letter
_ascii_to_end:
    dec ebx
    mov [byteToPrint], dl; store the result into memory
    ; save our values
    push eax
    push ebx
    ; print it
    mov eax, 4 ; system call #4 = sys_write
    mov ebx, 1 ; file descriptor 1 = stdout
```

(gdb) disassemble loop
```
Dump of assembler code for function loop:
  0x080483f1 <+0>:  mov     edx,eax
  0x080483f3 <+2>:  mov     ecx,ebx
  0x080483f5 <+4>:  shl     ecx,0x2
  0x080483f8 <+7>:  shr     edx,cl
  0x080483fa <+9>:  and     edx,0xf
  0x080483fd <+12>: cmp      dl,0xa
  0x08048400 <+15>: jge     0x08048407 <_ascii_to_hex>
  0x08048402 <+17>: add     dl,0x30
  0x08048405 <+20>: jmp     0x0804840a <_ascii_to_end>
```

github - dazzlecatduo

copyright(c) dazzlecatduo
Debugging x86 with GDB

• break address
• break label

(gdb) disassemble loop
Dump of assembler code for function loop:
0x080483f1 <+0>: mov edx,eax
0x080483f3 <+2>: mov ecx,ebx
0x080483f5 <+4>: shl ecx,0x2
0x080483f8 <+7>: shr edx,cl
0x080483fa <+9>: and edx,0xf
0x080483fd <+12>: cmp dl,0xa
0x08048400 <+15>: jge 0x8048407 <__ascii_to_hex>
0x08048402 <+17>: add dl,0x30
0x08048405 <+20>: jmp 0x804840a <__ascii_to_end>

End of assembler dump.
(gdb) break loop
Breakpoint 1 at 0x80483f1: file printreg-shift.asm, line 21.
(gdb)
Debugging x86 with GDB

• (gdb) info register
  – Show the current values in the x86 registers

Starting program: /home/swagger/Documents/osu/x86/debug/printreg-shift.out

Breakpoint 1, loop () at printreg-shift.asm:21
21       mov edx, eax    ;copy the value into edx for us to do manipulations on
        (gdb) info register
eax     0xabcd0012    -1412567278
ecx     0xffffffff04    -11516
edx     0xffffffff294 -11628
ebx     0x7            7
esp     0xffffffff68    0xffffffff68
ebp     0x0             0x0
esi     0x0             0
edi     0x0             0
eip     0x80483f1       0x80483f1 <loop>
eflags  0x246          [ PF ZF IF ]
cs      0x23           35
ss      0x2b           43
ds      0x2b           43
es      0x2b           43
fs      0x0            0
gs      0x63           99

Flags currently set
Decimal Value
Hex Value
Debugging x86 with GDB

• You can print individual registers
  – `print $reg`

```
(gdb) print $esp
$1 = (void *) 0xffffffff260
(gdb)
```
Debugging x86 with GDB

• Step 1 instruction at a time

(gdb) ste pi
34 dec ebx
(gdb) ste pi
35 mov [byteToPrint], dl; store the result into memory
(gdb) ste pi
37 push eax
(gdb) ste pi
38 push ebx
(gdb) ste pi
40 mov eax, 4 ; system call #4 = sys_write
(gdb) ste pi
41 mov ebx, 1 ; file descriptor 1 = stdout

Notice how we have our comments? Because we did a debug build those are left in
Debugging x86 with GDB

• If we would have wanted to step OVER a “call” (just like stepping over a function call in C when debugging), we would have used “nexti” instead

• “stepi” will step INTO any function if you call, “nexti” will step OVER it
Debugging x86 with GDB

• See all defined variables in the application
  – \texttt{(gdb) info variables}

\begin{verbatim}
(gdb) info variables
All defined variables:
Non-debugging symbols:
0x080490e0  loop_index
0x080490e4  byteToPrint
0x080490e5  __bss_start
0x080490e5  __edata
0x080490e8  __end
(gdb)
\end{verbatim}
Debugging x86 with GDB

- When wanting to debug your C code you could compile using the debug flag in GCC
  - gcc –g myfile.c

- What if you want to debug something that you can't rebuild with debug symbols?
  - You can debug in assembly!
Debugging with GDB

• When debugging code not built for debugging (i.e. someone elses code)

```
swagger@ubuntu:~/Documents/osu/ec$ gdb keychecker.out
GNU gdb (GDB) 7.5-ubuntu
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/swagger/Documents/osu/ec/keychecker.out. (no debugging symbols found)...done.
```

Debugging with GDB

Applications built in C have much more overhead and compiler generated sections than programming in straight assembly.
Debugging with GDB

0xf7fcff4 - 0xf7f7d00 is .got in /lib/ld-linux.so.2
0xf7f7d000 - 0xf7f7d024 is .got.plt in /lib/ld-linux.so.2
0xf7f7d040 - 0xf7f7d874 is .data in /lib/ld-linux.so.2
0xf7f7d878 - 0xf7f7f938 is .bss in /lib/ld-linux.so.2
0xf7e19174 - 0xf7e19198 is .note.gnu.build-id in /lib/i386-linux-gnu/libc.so.6
0xf7e19198 - 0xf7e191b8 is .note.ABI-tag in /lib/i386-linux-gnu/libc.so.6
0xf7e191b8 - 0xf7e1ce38 is .gnu.hash in /lib/i386-linux-gnu/libc.so.6
0xf7e1ce38 - 0xf7e26158 is .dynsym in /lib/i386-linux-gnu/libc.so.6
0xf7e26158 - 0xf7e2bcef is .dynstr in /lib/i386-linux-gnu/libc.so.6
0xf7e2bcf0 - 0xf7e2cf54 is .gnu.version in /lib/i386-linux-gnu/libc.so.6
0xf7e2cf54 - 0xf7e2d374 is .gnu.version_d in /lib/i386-linux-gnu/libc.so.6
0xf7e2d374 - 0xf7e2fdec is .rel.dyn in /lib/i386-linux-gnu/libc.so.6
0xf7e2fdec - 0xf7e2fe44 is .rel.plt in /lib/i386-linux-gnu/libc.so.6
0xf7e2fe50 - 0xf7e2ff10 is .text in /lib/i386-linux-gnu/libc.so.6
0xf7f646a0 - 0xf7f65613 is __libc_freeres_fn in /lib/i386-linux-gnu/libc.so.6
0xf7f65620 - 0xf7f65846 is __libc_thread_freeres_fn in /lib/i386-linux-gnu/libc.so.6
0xf7f65860 - 0xf7f843c8 is .rodata in /lib/i386-linux-gnu/libc.so.6
0xf7f843c8 - 0xf7f843db is .interp in /lib/i386-linux-gnu/libc.so.6
0xf7f843dc - 0xf7f88be8 is .eh_frame_hdr in /lib/i386-linux-gnu/libc.so.6
0xf7f88be8 - 0xf7f883c is .eh_frame in /lib/i386-linux-gnu/libc.so.6
0xf7f883c - 0xf7f8964 is .gcc_except_table in /lib/i386-linux-gnu/libc.so.6
0xf7f8964 - 0xf7f8be30 is .hash in /lib/i386-linux-gnu/libc.so.6
0xf7f8be30 - 0xf7f81d4 is .tdata in /lib/i386-linux-gnu/libc.so.6
0xf7f81d4 - 0xf7f81ec is .tbss in /lib/i386-linux-gnu/libc.so.6
0xf7f81ec - 0xf7f8d1c0 is .init_array in /lib/i386-linux-gnu/libc.so.6
0xf7f8d1c0 - 0xf7f8dd24 is __libc_subfreeres in /lib/i386-linux-gnu/libc.so.6
0xf7f8dd24 - 0xf7f8dd38 is __libc_atexit in /lib/i386-linux-gnu/libc.so.6
0xf7f8dd38 - 0xf7f8f248 is __libc_thread_subfreeres in /lib/i386-linux-gnu/libc.so.6
Debugging with GDB

• We can't view C code

```
(gdb) start
Temporary breakpoint 1 at 0x804848f
Starting program: /home/swagger/Documents/osu/ec/keychecker.out
Temporary breakpoint 1, 0x0804848f in main ()
(gdb)
No symbol table is loaded. Use the "file" command.
(gdb)
```

• There are no symbols!
Debugging with x86

- We can ALWAYS view the disassembly

```
(gdb) set disassembly flavor intel
(gdb) disassemble 0x80483a0
Dump of assembler code for function _start:
 0x080483a0 <+0>: xor ebp,ebp
 0x080483a2 <+2>: pop esi
 0x080483a3 <+3>: mov ecx,esp
 0x080483a5 <+5>: and esp,0xffffffff0
 0x080483a8 <+8>: push eax
 0x080483a9 <+9>: push esp
 0x080483aa <+10>: push edx
 0x080483ab <+11>: push 0x8048640
 0x080483b0 <+16>: push 0x80485d0
 0x080483b5 <+21>: push ecx
 0x080483b6 <+22>: push esi
 0x080483b7 <+23>: push 0x804848c
 0x080483bc <+28>: call 0x8048380 "libc_start_main@plt"
 0x080483c1 <+33>: hlt
 0x080483c2 <+34>: xchg ax,ax
 0x080483c4 <+36>: xchg ax,ax
 0x080483c6 <+38>: xchg ax,ax
 0x080483c8 <+40>: xchg ax,ax
 0x080483ca <+42>: xchg ax,ax
 0x080483cc <+44>: xchg ax,ax
 0x080483ce <+46>: xchg ax,ax
End of assembler dump.
```

copyright(c) dazzlecato
Debugging with x86

• Locate the call to __libc_start_main
• The value pushed on the stack just before that call is the address of OUR main() function in memory
Debugging with x86

• Locate the push right before libc_start
  – The code starts at 0x804848c

End of assembler dump.
(gdb) set disassembly-flavor intel
(gdb) disassemble 0x80483a0
Dump of assembler code for function _start:
  0x080483a0 <+0>:   xor   ebp,ebp
  0x080483a2 <+2>:   pop   esi
  0x080483a3 <+3>:   mov   ecx,esp
  0x080483a5 <+5>:   and   esp,0xffffffff0
  0x080483a8 <+8>:   push  eax
  0x080483a9 <+9>:   push  esp
  0x080483aa <+10>:  push  edx
  0x080483ab <+11>:  push  0x8048640
  0x080483b0 <+16>:  push  0x80485d0
  0x080483b5 <+21>:  push  ecx
  0x080483b6 <+22>:  push  esi
  0x080483b7 <+23>:  push  0x804848c
  0x080483bc <+28>:  call   0x8048380  <__libc_start_main@plt>
  0x080483c1 <+33>:  hlt
  0x080483c2 <+34>:  xchg  ax,ax
  0x080483c4 <+36>:  xchg  ax,ax
  0x080483c6 <+38>:  xchg  ax,ax
  0x080483c8 <+40>:  xchg  ax,ax
  0x080483ca <+42>:  xchg  ax,ax
  0x080483cc <+44>:  xchg  ax,ax
  0x080483ce <+46>:  xchg  ax,ax
End of assembler dump.
Debugging with x86

- Now we can disassemble the main

```assembly
End of assembler dump.
(gdb) disassemble 0x804848c
Dump of assembler code for function main:
0x0804848c <+0>: push ebp
0x0804848d <+1>: mov ebp,esp
0x0804848f <+3>: and esp,0xfffffffff0
0x08048492 <+6>: sub esp,0x20
0x08048495 <+9>: mov DWORD PTR [esp],0x8048668
0x0804849c <+16>: call 0x8048350 <printf@plt>
0x080484a1 <+21>: lea eax,[esp+0x1c]
0x080484a5 <+25>: mov DWORD PTR [esp+0x4],eax
0x080484a9 <+29>: mov DWORD PTR [esp],0x804868d
0x080484b0 <+36>: call 0x8048390 __isoc99_scanf@plt>
0x080484b5 <+41>: mov eax,DWORD PTR [esp+0x1c]
0x080484b9 <+45>: mov DWORD PTR [esp],eax
0x080484bc <+48>: call 0x80484eb <is_valid>
0x080484c1 <+53>: test eax,eax
0x080484c3 <+55>: je 0x80484d8 <main+76>
0x080484c5 <+57>: mov DWORD PTR [esp],0x8048690
0x080484cc <+64>: call 0x8048360 <puts@plt>
0x080484d1 <+69>: mov eax,0x1
0x080484d6 <+74>: jmp 0x80484e9 <main+93>
0x080484d8 <+76>: mov DWORD PTR [esp],0x804869a
0x080484df <+83>: call 0x8048360 <puts@plt>
0x080484e4 <+88>: mov eax,0x0
0x080484e9 <+93>: leave
0x080484ea <+94>: ret
End of assembler dump.
```
Using GDB

- Basic commands are usually sufficient
  - Starting and stopping
    - quit, run, kill
  - Breakpoints
    - break, delete
  - Execution
    - stepi, nexti, continue, finish
  - Examining code and data
    - disas, print, x
  - Useful information
    - info, help
  - Listing source code line numbers
    - list
SharkSim 3000

• In the linux VM (user: dazzlecat/ pass: dazzleme)
• cd ~/training/sharksim
• READ sharksim3000.pdf
  – you've spent thousands of hours crafting the most terrifying, realistic shark simulation the world has ever seen – all in x86 assembly!
  – There's just one problem – the program is segfaulting!!!
  – You'll need to debug the code to save your reputation and unleash the greatest game of all time – Shark Sim 3000!
Control Flow Instructions

- ip register (instruction pointer)
  - Holds the address of the current instruction
- ip register cannot be manipulated directly
- Updated by control flow instructions

- In x86 we use labels to denote locations in program text.
  - label name followed by a colon
    
    mov esi, [ebp+8]
    begin: xor, ecx, ecx
    mov eax, [esi]
Control Flow Instructions

- $\textit{jmp} \text{ op1}$
- Jump
- Transfers program control flow to the instruction at the memory location indicated by the op1
- Syntax
  $\textit{jmp} <$label$>$
- Example
  $\textit{jmp}$ begin  — Jump to the instruction labeled begin
Jump

- Using the JMP instruction, we can create an infinite loop that counts up from zero using the eax register:

```
  mov eax, 0
  loop: inc eax
  jmp loop
```
Conditional Jumps

• Conditional jumps take into consideration the current state of the flags to determine if a jump is taken or not
Control Flow Instructions

- Jumps
- *Syntax*
  
  `je <label>` (jump when equal)
  `jne <label>` (jump when not equal)
  `jz <label>` (jump when last result was zero)
  `jg <label>` (jump when greater than)
  `jge <label>` (jump when greater than or equal to)
  `jl <label>` (jump when less than)
  `jle <label>` (jump when less than or equal to)

- *Example*
  
  `cmp eax, ebx`
  
  `jle done` - If the contents of EAX are less than or equal to the contents of EBX, jump to the label *done*. Otherwise, continue to the next instruction.
Conditional Jumps

- **jge** - jump when greater than or equal to
  - Conditions: SF = 0 || ZF = 1

- **jl** - jump when less than
  - Conditions: SF = 1

- **jle** - jump when less than or equal to
  - Conditions: SF = 1 || ZF = 1
Conditional Jumps

• **je** - jump equals
  – Conditions: ZF = 1

• **jne** - jump when not equal
  – Conditions: ZF = 0

• **jz** - jump when last result was zero
  – Conditions: ZF = 1

• **jg** - jump when greater than
  – Conditions: SF = 0 && ZF = 0
Arithmetic and Logic

- *cmp*
  - subtracts op2 from op1, result is discarded
- *Flags*
  - o..szapc
- *Syntax*
  - `cmp <reg>,<reg>`
  - `cmp <reg>,<con>`
  - `cmp <reg>,<mem>`
  - `cmp <mem>,<mem>`
  - `cmp <mem>,<reg>`
  - `cmp <mem>,<con>`
- *Examples*
  - `cmp ax, 5` - sets ZF, OF, PF, and SF to appropriate state based on value in ax
Compare Example

• Why is this useful?
• Check if value is equal, greater, or less than another value

```assembly
mov eax, 0x100
mov ebx, 0x200
cmp eax, ebx; does eax - ebx

eax is less than ebx
SF = 1, ZF = 0
```
Compare Example

Why is this useful?
Check if value is equal, greater, or less than another value

```assembly
mov eax, 0x300
mov ebx, 0x200
cmp eax, ebx; does eax-ebx

eax is greater than ebx
SF = 0, ZF = 0
```
Compare Example

• Why is this useful?
• Check if value is equal, greater, or less than another value

```
mov eax, 0x500
mov ebx, 0x500
cmp eax, ebx; does eax-ebx

eax is equal to ebx
SF = 0, ZF = 1
```
Compare

cmp eax, ebx

<table>
<thead>
<tr>
<th>if</th>
<th>SF</th>
<th>ZF</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax &gt; ebx</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eax = ebx</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>eax &lt; ebx</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Conditional Jump

- Using conditional jumps we can implement a non-infinite loop

Loop to count eax from 0 to 5:
```
mov eax, 0
loop: inc eax
cmp eax, 5
jle loop
; if eax <= 5 then go to loop
```
Example

;C if ( a == b ) x = 1;

cmp ax, bx ; (ax-bx) == 0
jne skip ; not equal, so skip
mov cx, 1; since a == b, x=1
skip:
 nop ; no operation
Example

; C if ( a > b ) x = 1;

cmp a, b ; (a-b) > 0
jle skip ; skip if a <= b
mov x, 1

skip:

... ;stuff
Control Flow Instructions

• call
• pushes the address of the next instruction onto the stack, then performs an unconditional jump to the code location in op1
• Syntax
call <label>
Control Flow Instructions

• *ret*
  
  pops an address off of the stack and performs an unconditional jump to the address

• *Syntax*
  
  ret
C constructs in x86
C constructs in x86

- if (...) { ... }
- if (...) { ... } else { ... }
- if (...) { ... } else if (...) { ... } else { ... }
- while (...) { ... }
- do { ... } while (...);
- for (...; ...; ...) { ... }
- switch (...) { ... }
C constructs in x86

- All of these can be written using comparisons (cmp) and jumps (jmp, je, jne, jl, jle, jg, jge)
- When compiling your program, the compiler does this translation for you
- When writing your own assembly, you need to figure out the translation
Translation: C to Assembly

• Assembly does not have concepts of code blocks {...} like higher level languages do
  – Everything is just a stream of instructions
• Translation step 1: remove code blocks
  – Requires you to rewrite code using goto statements
• Translation step 2: rewrite as assembly
if (...) { ... }

With blocks:

```c
if (condition) {
    code_if_true;
}
```

Without blocks:

```c
if (!condition) goto skip_block;
```
```c
code_if_true;
```
```c
skip_block:
```
if (...) { ... }

With blocks:

```cpp
if (x==5)
{
    x++;  
    y=x;
}
```

Without blocks:

```cpp
if (x!=5)
    goto skip_block;
```

```cpp
x++;  
y=x;
```

```cpp
skip_block:
```

if (...) {
    ...
}

C:
if (x!=5)
    goto skip_block;

x++;
y=x;

skip_block:

x86:
cmp dword [x], 5
    jne skip

inc dword [x]
    mov eax, [x]
    mov [y], eax

skip:
if (...) { ... } else { ... }

With blocks:

```c
if (condition) {
    code_if_true;
}
else {
    code_if_false;
}
```

Without blocks:

```c
if (!condition) {
    goto false_block;
    code_if_true;
    goto skip_block;
}
false_block:
    code_if_false;
skip_block:
```
if (...) { ... } else { ... }

With blocks:

```c
if (x)
{
    x++;  
}
else
{
    x--;  
}
```

Without blocks:

```c
if (!x)
    goto false_block; 

x++; 

goto skip_block; 

false_block: 
    x--; 

skip_block: 
```
if (...) { ... } else { ... }

C:

if (!x)
    goto false_block;

x++;
goto skip_block;

false_block:
    x--;

skip_block:

x86:

cmp dword [x], 0
je false_block

inc dword [x]
jmp skip_block

false_block:
    dec dword [x]

skip_block:
```plaintext
while (...) {
    ...
}

With blocks:

while (condition) {
    code;
}

Without blocks:

loop:
    if (!condition) {
        goto done;
    }
    code;
    goto loop;

done:

github - dazzlecatduo
```
while (...) { ... }

With blocks:

while (tired) {
    sleep();
}

Without blocks:

loop:
if (!tired)
    goto done;

sleep();
goto loop;

done:
while (...) {
    ...
}

C:

loop:
if (!tired)
    goto done;
sleep();
goto loop;
done:

x86:

loop:
    cmp dword [tired], 0
    je done
call sleep
goto loop
done:
for (..., ..., ...) {
  ...
}

With blocks:

```
for (expr_1; expr_2; expr_3) {
  code;
}
```

Without blocks:

```
expr_1;
loop:
  if (!expr_2)
    goto done;
  code;
  expr_3;
  goto loop;

done:
```
for (...; ...; ...) { ... }

With blocks:

```c
for (i=0; i<100; i++)
{
    sum+=i;
}
```

Without blocks:

```c
i=0;
loop:
if (i>=100)
    goto done;
sum+=i;
i++; goto loop;
done:
```
for (...; ...; ...) { ... }

C:

```c
i=0;
loop:
if (i>=100)
goto done;
sum+=i;
i++;
goto loop;
done:
```

x86:

```assembly
mov dword [i], 0
loop:
cmp dword [i], 100
jge done
mov eax,[i]
add [sum],eax
inc dword [i]
jmp loop
```
Objdump

- Objdump is a tool that will let you see the assembly code for any application
- syntax: `objdump <application> -Mintel -d`
  - `-Mintel` says you want intel assembly syntax
  - `-d` says to disassemble the code
IDA

- IDA – Interactive Disassembler
- Allows for binary visualization of disassembly
- About
- Download
  - [https://hex-rays.com/products/ida/support/download.shtml](https://hex-rays.com/products/ida/support/download.shtml)
- Free version supports x86
Drag a file here to disassemble it

Can not set debug privilege: Not all privileges or groups referenced are assigned to the caller.
• IDA recognizes many common file formats
• If it gets it wrong you can select generic ‘binary file’
• Processor type drop down to change architectures
IDA

• IDA shows code in Basic Blocks
• basic block has:
  – one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program;
  – one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.
int main(int argc, char* argv[]) {
    return argc;
}
IDA is smart enough to know that the first argument always starts at ebp+8, so it renames that offset to arg_0 to make it easier to read.
IDA Paths

• IDA shows 3 different types of paths between basic blocks
  • RED – Path taken if conditional jump is not taken
  • GREEN – Path taken if conditional jump is taken
  • BLUE – Guaranteed path
int main(int argc, char* argv[]) {
    if (argc > 1)
        return 0;
    return argc;
}
public main
main proc near

arg_0 = dword ptr 8

push ebp
mov ebp, esp
cmp [ebp+arg_0], 1
jle short loc_8000010

mov eax, 0
jmp short loc_8000013

loc_8000010:
mov eax, [ebp+arg_0]

loc_8000013:
pop ebp

loc_ret_8000014:
ret
main endp
_text ends
C – If Example

```c
public main
main proc near
arg_0= dword ptr 8
push ebp
mov ebp, esp
cmp [ebp+arg_0], 1
jle short loc_8000010
```

Taken if arg_0 greater than 1

- Taken if arg_0 less than or equal to 1

```c
cmp [ebp+arg_0], 1
```
Compares argc to 1

```c
jle – Jump less than equal
```

```
mov eax, 0
jmp short loc_8000013
```

```
loc_8000010:
mov eax, [ebp+arg_0]
```

```
loc_8000013:
```
pop ebp
```

```
locret_8000014:
retn
main endp
_text ends
```
IDA – If And Example

• Lets look at some common C structures in assembly

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    if (argc >= 3 && argc <= 8) {
        printf("valid number of args\n");
    }

    return 0;
}
```
IDA – If And Example

- Objdump view - Hard to read!

```assembly
000000000040051c <main>:
40051c:      55       push    rbp
40051d:      48 89 e5  mov     rbp,rsp
400520:      48 83 ec 10 sub     rsp,0x10
400524:      89 7d fc  mov     DWORD PTR [rbp-0x4],edi
400527:      48 89 75 f0 mov     QWORD PTR [rbp-0x10],rsi
40052b:      83 7d fc 02 cmp     DWORD PTR [rbp-0x4],0x2
40052f:      7e 10 jle 400541 <main+0x25>
400531:      83 7d fc 08 cmp     DWORD PTR [rbp-0x4],0x8
400535:      7f 0a jg 400541 <main+0x25>
400537:      bf f4 05 40 00 mov     edi,0x4005f4
40053c:      e8 af fe ff ff call 4003f0 <puts@plt>
400541:      b8 00 00 00 00 mov     eax,0x0
400546:      c9       leave
400547:      c3       ret
```
IDA View: If And Example

```c
public main
main proc near

arg_0= dword ptr 8
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 10h
cmp [ebp+arg_0], 2
jle short loc_8000021

cmp [ebp+arg_0], 8
jg short loc_8000021

mov dword ptr [esp], offset s ; "valid number of args"
call puts

loc_8000021:
mov eax, 0
leave

locret_8000027:
retn
main endp
_text ends
```
#include <stdio.h>

int main(int argc, char* argv[]) {
    int i;
    i = 0;
    while (i < 10) {
        printf("i: %i\n", i);
        i += 2;
    }

    return 0;
}
public main
main proc near
push ebp
mov ebp, esp
and esp, 0FFFFFFF0h
sub esp, 20h
mov dword ptr [esp+1Ch], 0
jmp short loc_800002C

loc_800002C:
cmp dword ptr [esp+1Ch], 9
jle short loc_8000013

mov eax, 0
leave

loc_8000013:
mov eax, [esp+1Ch]
mov [esp+4], eax
mov dword ptr [esp], offset format ; "i: %d\n"
call printf
add dword ptr [esp+1Ch], 2

locrct_8000039:
retn
main endp
_text ends
#include <stdio.h>
int main(int argc, char* argv[]){
    int i, j;
    i = 0;
    while (i < 10)
    {
        j = 0;
        while (j < 5)
        {
            printf("i: %i, j: %i\n", i, j);
            j++;
        }
        i++;
    }
    return 0;
}
public main
main proc near
push  ebp
mov   ebp, esp
and   esp, 0FFFFFFF0h
sub   esp, 20h
mov   dword ptr [esp+1Ch], 0
jmp   short loc_800004A

loc_800004A:
    cmp   dword ptr [esp+1Ch], 9
    jle   short loc_8000013

loc_8000013:
    mov   dword ptr [esp+18h], 0
    jmp   short loc_800003E

loc_800003E:
    cmp   dword ptr [esp+18h], 4
    jle   short loc_800001D

loc_800001D:
    mov   eax, [esp+18h]
    mov   [esp+8], eax
    mov   eax, [esp+1Ch]
    mov   [esp+4], eax
    mov   dword ptr [esp], offset format ; "i: %i, j: %i\n"
call   printf
    add   dword ptr [esp+18h], 1

loc_8000057:
    ret
main endp
_text ends

mov   eax, 0
leave
#include <stdio.h>

int main(int argc, char* argv[]) 
{
    int i;

    for (i = 0; i < 10; i++)
    {
        printf("i: %i
", i);
    }

    return 0;
}
Goodies

• Windows VM (user: dazzlecat/pass: dazzleme)
• ~Desktop/training/assembly_samples
  – src – Contains simple c programs that incorporate a basic logic flow (if/else/etc)
  – bin – compiled programs of the src with different optimization levels
    • -O0  optimization for compilation time (default)
    • -O2  optimization more for code size and execution time
    • -Os  optimization for code size
IDA – Example

• IDA Patching - Demo
dazzlecatduo on github
dazzle.cat.duo@gmail.com

Dazzlecatduo logo source:
http://embed.polyvoreimg.com/cgi/img-thing/size/y/tid/38030333.jpg