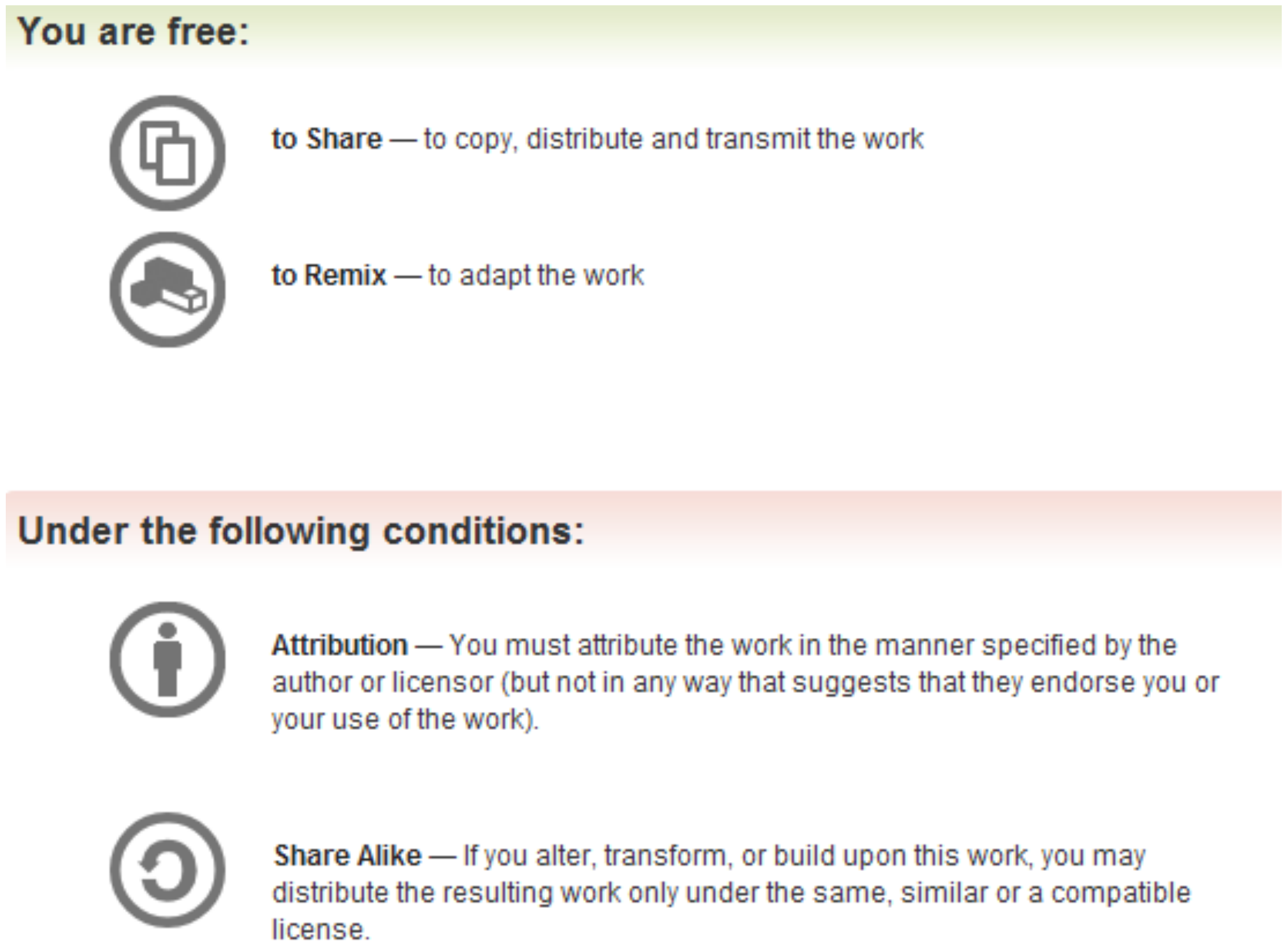
Intro x86 Part 3: Linux Tools & Analysis

Xeno Kovah – 2009/2010
xkovah at gmail

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Intel vs. AT&T Syntax

- Intel: Destination <- Source(s)
  - Windows. Think algebra or C: y = 2x + 1;
  - mov   ebp, esp
  - add   esp, 0x14 ; (esp = esp + 0x14)
- AT&T: Source(s) -> Destination
  - *nix/GNU. Think elementary school: 1 + 2 = 3
  - mov %esp, %ebp
  - add $0x14,%esp
  - So registers get a % prefix and immediates get a $
- My classes will use Intel syntax except in this section
- But it's important to know both, so you can read documents in either format.

Intel vs AT&T Syntax 2

- In my opinion the hardest-to-read difference is for r/m32 values
- For intel it's expressed as

```
[base + index*scale + disp]
```

- For AT&T it's expressed as

```
disp(base, index, scale)
```

- Examples:
  - call   DWORD PTR [ebx+esi*4-0xe8]
  - call   *-0xe8(%ebx,%esi,4)

  - mov   eax, DWORD PTR [ebp+0x8]
  - mov   0x8(%ebp), %eax

  - lea   eax, [ebx-0xe8]
  - lea   -0xe8(%ebx), %eax

Intel vs AT&T Syntax 3

- For instructions which can operate on different sizes, the mnemonic will have an indicator of the size.
  - movb - operates on bytes
  - mov/movw - operates on word (2 bytes)
  - movl - operates on “long” (dword) (4 bytes)
- Intel does indicate size with things like “mov dword ptr [eax], but it’s just not in the actual mnemonic of the instruction

gcc - GNU project C and C++ compiler

- Available for many *nix systems (Linux/BSD/OSX/Solaris)
- Supports many other architectures besides x86
- Some C/C++ options, some architecture-specific options
  - Main option we care about is building debug symbols. Use "-ggdb" command line argument.
- Basically all of the VisualStudio options in the project properties page are just fancy wrappers around giving their compiler command line arguments. The equivalent on *nix is for to developers create "makefile"s which are a configuration or configurations which describes which options will be used for compilation, how files will be linked together, etc. We won't get that complicated in this class, so we can just specify command line arguments manually.

Book p. 53

gcc basic usage

- gcc -o <output filename> <input file name>
  - gcc -o hello hello.c
  - If -o and output filename are unspecified, default output filename is “a.out” (for legacy reasons)
- So we will be using:
  - gcc -ggdb -o <filename> <filename>.c
  - gcc -ggdb -o Example1 Example1.c

objdump - display information from object files

- Where ”object file” can be an intermediate file created during compilation but before linking, or a fully linked executable
  - For our purposes means any ELF file - the executable format standard for Linux
- The main thing we care about is -d to disassemble a file.
- Can override the output syntax with “-M intel”
  - Good for getting an alternative perspective on what an instruction is doing, while learning AT&T syntax

Book p. 63

objdump -d hello

```
hello:     file format elf32-i386

Disassembly of section .init:

08048274 <_init>:
 8048274:      55                      push   %ebp
 8048275:      89 e5                   mov    %esp,%ebp
 8048277:      53                      push   %ebx
 8048278:      83 ec 04                sub    $0x4,%esp
 804827b:      e8 00 00 00 00          call   8048280 <_init+0xc>
…
08048374 <main>:
 8048374:      8d 4c 24 04             lea    0x4(%esp),%ecx
 8048378:      83 e4 f0                and    $0xfffffff0,%esp
 804837b:      ff 71 fc                pushl  -0x4(%ecx)
 804837e:      55                      push   %ebp
 804837f:      89 e5                   mov    %esp,%ebp
 8048381:      51                      push   %ecx
…
```

objdump -d -M intel hello

```
hello:     file format elf32-i386

Disassembly of section .init:

08048274 <_init>:
 8048274:      55                      push   ebp
 8048275:      89 e5                   mov    ebp,esp
 8048277:      53                      push   ebx
 8048278:      83 ec 04                sub    esp,0x4
 804827b:      e8 00 00 00 00          call   8048280 <_init+0xc>
…
08048374 <main>:
 8048374:      8d 4c 24 04             lea    ecx,[esp+0x4]
 8048378:      83 e4 f0                and    esp,0xfffffff0
 804837b:      ff 71 fc                push   DWORD PTR [ecx-0x4]
 804837e:      55                      push   ebp
 804837f:      89 e5                   mov    ebp,esp
 8048381:      51                      push   ecx
…
```

hexdump & xxd

- Sometimes useful to look at a hexdump to see opcodes/operands or raw file format info
- hexdump, hd - ASCII, decimal, hexadecimal, octal dump
  - hexdump -C for "canonical" hex & ASCII view
  - Use for a quick peek at the hex
- xxd - make a hexdump or do the reverse
  - Use as a quick and dirty hex editor
  - xxd hello > hello.dump
  - Edit hello.dump
  - xxd -r hello.dump > hello

GDB - the GNU debugger

- A command line debugger - quite a bit less user-friendly for beginners.
  - There are wrappers such as ddd but I tried them back when I was learning asm and didn't find them to be helpful. YMMV
- Syntax for starting a program in GDB in this class:
  - gdb <program name> -x <command file>
  - gdb Example1 -x myCmds

Book p. 57

About GDB -x <command file>

- Somewhat more memorable long form is “--command=<command file>”
- <command file> is a plaintext file with a list of commands that GDB should execute upon starting up. Sort of like scripting the debugger.
- Absolutely **essential** to making GDB reasonable to work with for extended periods of time (I used GDB for many years copying and pasting my command list every time I started GDB, so I was super ultra happy when I found this option)

GDB commands

- “help” - internal navigation of available commands
- “run” or “r” - run the program
- “r <argv>” - run the program passing the arguments in <argv>
  - I.e. for Example 2 “r 1 2” would be what we used in windows

GDB commands 2

- “help display”
- “display” prints out a statement every time the debugger stops
- display/FMT EXP
- FMT can be a combination of the following:
  - i - display as asm instruction
  - x or d - display as hex or decimal
  - b or h or w - display as byte, halfword (2 bytes), word (4 bytes - as opposed to intel calling that a double word. Confusing!)
  - s - character string (will just keep reading till it hits a null character)
  - <number> - display <number> worth of things (instructions, bytes, words, strings, etc)
- “info display” to see all outstanding display statements and their numbers
- “undisplay <num>” to remove a display statement by number

GDB commands 3

- “x/FMT EXP” - x for “Examine memory” at expression
  - Always assumes the given value is a memory address, and it dereferences it to look at the value **at** that memory address
- “print/FMT EXP” - print the value of an expression
  - Doesn’t try to dereference memory
- Both commands take the same type of format specifier as display
- Example:

```
(gdb) x/x $ebp
0xbffbcb78:     0xbffbcbe8
(gdb) print/x $ebp
$2 = 0xbffbcb78
(gdb) x/x $eax
0x1:    Cannot access memory at address 0x1
(gdb) print/x $eax
$3 = 0x1
```

GDB commands 4

- For all breakpoint-related commands see "help breakpoints"
- "break" or "b" - set a breakpoint
  - With debugging symbols you can do things like "b main". Without them you can do things like

    "b *<address>" to break at a given memory address.
  - Note: gdb's interpretation of where a function begins may exclude the function prolog like "push ebp"…
- "info breakpoints" or "info b" - show currently set breakpoints
- "delete <num> - deletes breakpoint number <num>, where <num> came from "info breakpoints"

GDB 7 commands

- New for GDB 7, released Sept 2009
  - Thanks to Dave Keppler for notifying me of the availability of these new commands
  - **reverse-step** ('rs') -- Step program backward until it reaches the beginning of a previous source line
  - **reverse-stepi** -- Step backward exactly one instruction
  - **reverse-continue** ('rc') -- Continue program being debugged but run it in reverse
  - **reverse-finish** -- Execute backward until just before the selected stack frame is called

GDB 7 commands 2

- **reverse-next** ('rn') -- Step program backward, proceeding through subroutine calls.
- **reverse-nexti** ('rni') -- Step backward one instruction, but proceed through called subroutines.
- **set exec-direction (forward/reverse)** -- Set direction of execution. All subsequent execution commands (continue, step, until etc.) will run the program being debugged in the selected direction.

- The "**disassemble**" command now supports an optional **/m** modifier to print mixed source+assembly.
- "**disassemble**" command with a **/r** modifier, print the raw instructions in hex as well as in symbolic form.
- See “help disassemble” for full syntax

initial GDB commands file

- display/10i $eip
- display/x $eax
- display/x $ebx
- display/x $ecx
- display/x $edx
- display/x $edi
- display/x $esi
- display/x $ebp
- display/32xw $esp
- break main

```
(gdb) r
Starting program: /home/user/hello

Breakpoint 1, main () at hello.c:4
4           printf("hello\n");
9: x/32xw $esp
0xbf9a2550:    0xb7f46db0    0xbf9a2570    0xbf9a25c8    0xb7df2450
0xbf9a2560:    0xb7f53ce0    0x080483b0    0xbf9a25c8    0xb7df2450
<snip>
8: /x $ebp = 0xbf9a2558
7: /x $esi = 0xb7f53ce0
6: /x $edi = 0x0
5: /x $edx = 0xbf9a2590
4: /x $ecx = 0xbf9a2570
3: /x $ebx = 0xb7f26ff4
2: /x $eax = 0x1
1: x/10i $eip
0x8048385 <main+17>:    movl   $0x8048460,(%esp)
0x804838c <main+24>:    call   0x80482d4 <puts@plt>
0x8048391 <main+29>:    mov    $0x1234,%eax
0x8048396 <main+34>:    add    $0x4,%esp
0x8048399 <main+37>:    pop    %ecx
0x804839a <main+38>:    pop    %ebp
0x804839b <main+39>:    lea    -0x4(%ecx),%esp
0x804839e <main+42>:    ret
0x804839f:      nop
0x80483a0 <__libc_csu_fini>:    push   %ebp
```

Source code line printed here if source is available

Stepping

- “stepi” or “si” - steps one asm instruction at a time
  - Will always “step into” subroutines
- “step” or “s” - steps one source line at a time (if no source is available, works like stepi)
- “until” or “u” - steps until the next source line, not stepping into subroutines
  - If no source available, this will work like a stepi that will “step over” subroutines

GDB misc commands

- “set disassembly-flavor intel” - use intel syntax rather than AT&T
  - Again, not using now, just good to know
- “continue” or “c” - run until you hit another breakpoint or the program ends
- “backtrace” or “bt” - print a trace of the call stack, showing all the functions which were called before the current function

Lab time:
Running examples with GDB