Weeks 3 and 4. Tools Overview, Assembly, Hello World!

Tools

- This overview can and should be presented with a combination of slides and live demos, starting with slides and moving to a quick demo of the tool in question. Application of tools with Hello World! will be done after all tools are presented. These tools will be essential to the completion of the Final Project.
- Create the following in notepad / gedit, compile with gcc (will get into gcc later section, run it)

```
#include <stdio.h>
int main() {
    printf("hello, world\n");
return 0; }
```

- Unix Tools
  - Man pages
  - Strings
  - File
- Debuggers Defined
  - **What?**
    - A program that is used to test and debug other programs.
    - The debuggers we will be looking at are low-level debugger or a machine-language debugger it shows the line in the disassembly (unless it also has online access to the original source code and can display the appropriate section of code from the assembly or compilation).
    - Most popular debuggers (GDB) utilize a command line interface (CLI), but can also controlled via GUI Front-End extensions
  - **How?**
    - Typically, debuggers offer a query processor, a symbol resolver, an expression interpreter, and a debug support interface at its top level.
    - Debuggers also offer more sophisticated functions such as running a program step by step (single-stepping or program animation), stopping

(breaking) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint, and tracking the values of variables.

- Some debuggers have the ability to modify program state while it is running. It may also be possible to continue execution at a different location in the program to bypass a crash or logical error.
- GDB offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.

- **Why?**
  - This functionality can help identify where or how something is breaking in your code, and to slowly test the program (in steps) to identify bottlenecks or redundancy.
    - The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features. It often also makes it useful as a general verification tool, fault coverage, and performance analyzer, especially if instruction path lengths are shown.

- GDB
  - GNU debugger
  - Allows you to debug a program providing this functionality:
    - Start your program, specifying anything that might affect its behavior.
    - Make your program stop on specified conditions.
    - Examine what has happened, when your program has stopped.
    - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
    - You can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.
  - Languages Supported
    - Ada

- Assembly
- C
- C++
- D
- Fortran
- Go
- Objective-C
- OpenCL
- Modula-2
- Pascal
- Rust

- GDB Hello World! Output:

```
root@debian-mips:~# gcc hw.c -O3 -o hw

root@debian-mips:~# gdb hw
GNU gdb (GDB) 7.0.1-debian
...
Reading symbols from /root/hw...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x400654
(gdb) run
Starting program: /root/hw

Breakpoint 1, 0x00400654 in main ()
(gdb) set step-mode on
(gdb) disas
Dump of assembler code for function main:
0x00400640 <main+0>:    lui     gp,0x42
0x00400644 <main+4>:    addiu   sp,sp,-32
0x00400648 <main+8>:    addiu   gp,gp,-30624
0x0040064c <main+12>:   sw      ra,28(sp)
0x00400650 <main+16>:   sw      gp,16(sp)
0x00400654 <main+20>:   lw      t9,-32716(gp)
0x00400658 <main+24>:   lui     a0,0x40
0x0040065c <main+28>:   jalr    t9
0x00400660 <main+32>:   addiu   a0,a0,2080
0x00400664 <main+36>:   lw      ra,28(sp)
0x00400668 <main+40>:   move    v0,zero
0x0040066c <main+44>:   jr      ra
0x00400670 <main+48>:   addiu   sp,sp,32
End of assembler dump.
(gdb) s
0x00400658 in main ()
(gdb) s
0x0040065c in main ()
(gdb) s
0x2ab2de60 in printf () from /lib/libc.so.6
(gdb) x/s $a0
0x400820:        "hello, world"
(gdb)
```

Reverse Engineering

- CheatSheet - pg 1046 in our textbook

option	meaning
break filename.c:number	set a breakpoint on line number in source code
break function	set a breakpoint on function
break *address	set a breakpoint on address
b	—"—
p variable	print value of variable
run	run
r	—"—
cont	continue execution
c	—"—
bt	print stack
set disassembly-flavor intel	set Intel syntax
disas	disassemble current function
disas function	disassemble function
disas function,+50	disassemble portion
disas $eip,+0x10	—"—
disas/r	disassemble with opcodes
info registers	print all registers
info float	print FPU-registers
info locals	dump local variables (if known)
x/w ...	dump memory as 32-bit word
x/w $rdi	dump memory as 32-bit word at address in RDI
x/10w ...	dump 10 memory words
x/s ...	dump memory as string
x/i ...	dump memory as code
x/10c ...	dump 10 characters
x/b ...	dump bytes
x/h ...	dump 16-bit halfwords
x/g ...	dump giant (64-bit) words
finish	execute till the end of function
next	next instruction (don't dive into functions)
step	next instruction (dive into functions)
set step-mode on	do not use line number information while stepping
frame n	switch stack frame
info break	list of breakpoints
del n	delete breakpoint
set args ...	set command-line arguments

- IDA
  - "IDA Pro combines an interactive, programmable, multi-processor disassembler coupled to a local and remote debugger and augmented by a complete plugin programming environment. "
  - Very expensive but very powerful disassembler used by cybersecurity and reverse engineering professionals.
  - "IDA has become the de-facto standard for the analysis of hostile code, vulnerability research and COTS validation."
  - **Capabilities?**

- Disassembly
  - As a disassembler, IDA Pro explores binary programs, for which source code isn't always available, to create maps of their execution.
    - This is done by converting machine code into assembly language.
- Debugging
  - The debugger in IDA Pro complements the static analysis capabilities of the disassembler: by allowing an analyst to single step through the code being investigated, the debugger often bypasses the obfuscation and helps obtain data that the more powerful static disassembler will be able to process in depth.
- Interactivity
  - IDA always allows the human analyst to override its decisions or to provide hints. Interactivity culminates in a built-in programming language and an open plugin architecture.
- Programability
  - Contains a very powerful macro-like language that can be used to automate simple to medium complexity tasks.

- **Useful?**
  - Virtually all anti-virus companies, most vulnerability research companies, many of the large software development companies, three letter agencies and military organizations utilize IDA Pro.
- IDAPro is a beast to understand, and too complex for where we are going in this class.
  - Has a freeware version
    - Is good for graphs, and a second set of “eyes”, I recommend Radare2 for the project in this class.

- Radare2
  - RAw DAta REcovery
  - A FREE CLI based disassembler like IDA
    - Has front end GUI extensions I promise

Reverse Engineering

- History:
  - Was created to as a simple hexadecimal editor with support for searching patterns and dumping the search results to disk to recover some PHP files deleted from an HFS partition
  - From there, other features were added on top of the simple command line hexadecimal editor to support disassemblers, file format parsers, debuggers and more.
  - Radare2 began as a rewrite of the radare codebase following a modular design and licensed under LGPLv3.
- Advertised that it can:
  - Disassemble (and assemble for) many different architectures
  - Debug natively or use remote targets (gdb, r2pipe, winedbg, windbg)
  - Run on Linux, *BSD, Windows, OSX, Android, iOS, Solaris and Haiku
  - Perform forensics on filesystems and data carving
  - Be scripted in Python, Javascript, Go and more
  - Support collaborative analysis using the embedded webserver
  - Visualize data structures of several file types
  - Patch programs to uncover new features or fix vulnerabilities
  - Use powerful analysis capabilities to speed up reversing
  - Aid in software exploitation
- Output of Hello World:

```
$ r2 hw
 -- Enhance your graphs by increasing the size of the block and
graph.depth eval variable.
[0x00000540]> aa
[x] Analyze all flags starting with sym. and entry0 (aa)
[0x00000540]> s main
[0x0000064a]> v
/ (fcn) sym.main 19
|   sym.main ();
|              ; DATA XREF from 0x0000055d (entry0)
|           0x0000064a      55             push rbp
|           0x0000064b      4889e5         mov rbp, rsp
```

```
|           0x0000064e      488d3d8f0000.  lea rdi,
str.Hello_World    ; 0x6e4 ; "Hello World"
|           0x00000655      e8d6feffff     call sym.imp.puts
;[1] ; int puts(const char *s)
|           0x0000065a      90             nop
|           0x0000065b      5d             pop rbp
\           0x0000065c      c3             ret
```

- Cheat Sheet
  - Best one: http://b404.xyz/sources/r2-cheatsheet.pdf (linked on website)
- Typical process to start debugging:
  - `r2 <program_name>`
  - `aaa # analyze and autoname all functions`
  - `afl # list the auto named functions`
  - `s <function> # seek to the function you would like to view (main to start)`
  - `pdf # print the disassembly of the current function`
  - `VV # visual mode (optional)`
- To debug, start the program with the -d flag
  - `r2 -d <program_name>`
  - Set breakpoints with `db`
  - Run with `dc`

Assembly

- It is assumed that knowledge of Assembly would be gained from Computer Organization and Architecture which is a prerequisite for this course. If additional time is needed to review common assembly instructions, use resources below:
- Notation:

`<reg32>`	Any 32-bit register (`EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP`, or `EBP`)
`<reg16>`	Any 16-bit register (`AX`, `BX`, `CX`, or `DX`)
`<reg8>`	Any 8-bit register (`AH`, `BH`, `CH`, `DH`, `AL`, `BL`, `CL`, or `DL`)
`<reg>`	Any register
`<mem>`	A memory address (e.g., `[eax]`, `[var + 4]`, or `dword ptr [eax+ebx]`)
`<con32>`	Any 32-bit constant
`<con16>`	Any 16-bit constant

`<con8>`	Any 8-bit constant
`<con>`	Any 8-, 16-, or 32-bit constant

- Common Instructions provided by Adam Ferrari

- `MOV`
  - Move
  - Syntax
    - `mov <reg>,<reg>`
    - `mov <reg>,<mem>`
    - `mov <mem>,<reg>`
    - `mov <reg>,<const>`
    - `mov <mem>,<const>`
  - Examples
    - `mov eax, ebx`
      - copy the value in ebx into eax
    - `mov byte ptr [var], 5`
      - store the value 5 into the byte at location var
- `POP`
  - Pop Stack
  - Syntax
    - `pop <reg32>`
    - `pop <mem>`
  - Examples
    - `pop edi`
      - pop the top element of the stack into EDI.
    - `pop [ebx]`
      - pop the top element of the stack into memory at the four bytes starting at location EBX.
- `PUSH`
  - Push Stack
  - Syntax
    - `push <reg32>`
    - `push <mem>`
    - `push <con32>`

- Examples
  - `push eax`
    - push eax on the stack
  - `push [var]`
    - push the 4 bytes at address var onto the stack

- `LEA`
  - Load Effective Address
  - Syntax
    - `lea <reg32>,<mem>`
  - Examples
    - `lea edi, [ebx+4*esi]`
      - the quantity EBX+4*ESI is placed in EDI.
    - `lea eax, [var]`
      - the value in var is placed in EAX.

- `ADD`
  - Integer Addition
  - Syntax
    - `add <reg>,<reg>`
    - `add <reg>,<mem>`
    - `add <mem>,<reg>`
    - `add <reg>,<con>`
    - `add <mem>,<con>`
  - Examples
    - `add eax, 10`
      - EAX ← EAX + 10
    - `add BYTE PTR [var], 10`
      - add 10 to the single byte stored at memory address var

- `SUB`
  - Integer Subtraction
  - Syntax
    - `sub <reg>,<reg>`
    - `sub <reg>,<mem>`
    - `sub <mem>,<reg>`
    - `sub <reg>,<con>`
    - `sub <mem>,<con>`

- Examples
  - `sub al, ah`
    - AL ← AL - AH
  - `sub eax, 216`
    - subtract 216 from the value stored in EAX

- `JMP`
  - Jump
  - Syntax
    - `jmp <label>`
  - Examples
    - `jmp begin`
      - Jump to the instruction labeled begin.

- `J Condition`
  - Conditional Jump
    - Based on the status of a set of condition codes that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed.
  - 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)`
  - Examples
    - `cmp eax, ebx`
    - `jne done`

- `CMP`
  - Compare

- Syntax
  - `cmp <reg>,<reg>`
  - `cmp <reg>,<mem>`
  - `cmp <mem>,<reg>`
  - `cmp <reg>,<con>`
- Examples
  - `cmp DWORD PTR [var], 10`
  - `jeq loop`
    - If the 4 bytes stored at location var are equal to the 4-byte integer constant 10, jump to the location labeled loop.

- `CALL, RET`
  - Subroutine Call and Return
  - Syntax
    - `call <label>`
    - `ret`
      - The call instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.
      - The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack. It then performs an unconditional jump to the retrieved code location.

- Instructions interacting with Registers - Little Man Computer
  - Live Demo on https://schweigi.github.io/assembler-simulator/ (loops are covered in-depth later)
  - OR / THEN
  - Live Demo stepping through hello world with Radare2 / GDB (professor preference) This is the preview for weeks 7 and 8