Introduction to Assembly and Reverse Engineering

By Sam Goodwin

What is Assembly Language?

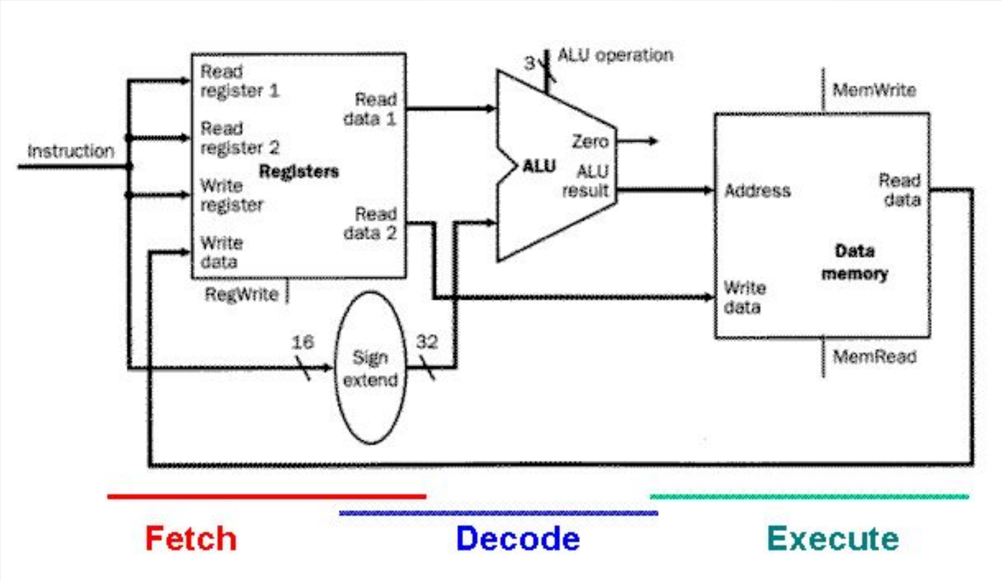
- **Very** low level programming language
- Pretty much macros for writing machine code
    - Machine Code - Computer instructions that directly control the CPU
- Assembly code is run through an assembler that turns it into machine code that can be executed by your computer.
- Old and pretty sucky, but much easier than writing/reading machine code ourselves.
- Represents the lowest level operations of your computer

```
00000000000010c0 <register_tm_clones>:
    10c0:       48 8d 3d 49 2f 00 00    lea    0x2f49(%rip),%rdi        # 4010 <__TMC_END__>
    10c7:       48 8d 35 42 2f 00 00    lea    0x2f42(%rip),%rsi        # 4010 <__TMC_END__>
    10ce:       48 29 fe                sub    %rdi,%rsi
    10d1:       48 89 f0                mov    %rsi,%rax
    10d4:       48 c1 ee 3f             shr    $0x3f,%rsi
    10d8:       48 c1 f8 03             sar    $0x3,%rax
    10dc:       48 01 c6                add    %rax,%rsi
    10df:       48 d1 fe                sar    %rsi
    10e2:       74 14                   je     10f8 <register_tm_clones+0x38>
    10e4:       48 8b 05 05 2f 00 00    mov    0x2f05(%rip),%rax        # 3ff0 <_ITM_registerTMCloneTable>
    10eb:       48 85 c0                test   %rax,%rax
    10ee:       74 08                   je     10f8 <register_tm_clones+0x38>
    10f0:       ff e0                   jmpq   *%rax
    10f2:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    10f8:       c3                      retq
    10f9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001100 <__do_global_dtors_aux>:
    1100:       f3 0f 1e fa             endbr64
    1104:       80 3d 05 2f 00 00 00    cmpb   $0x0,0x2f05(%rip)        # 4010 <__TMC_END__>
    110b:       75 2b                   jne    1138 <__do_global_dtors_aux+0x38>
    110d:       55                      push   %rbp
    110e:       48 83 3d e2 2e 00 00    cmpq   $0x0,0x2ee2(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
    1115:       00
    1116:       48 89 e5                mov    %rsp,%rbp
    1119:       74 0c                   je     1127 <__do_global_dtors_aux+0x27>
    111b:       48 8b 3d e6 2e 00 00    mov    0x2ee6(%rip),%rdi        # 4008 <__dso_handle>
    1122:       e8 19 ff ff ff          callq  1040 <__cxa_finalize@plt>
    1127:       e8 64 ff ff ff          callq  1090 <deregister_tm_clones>
    112c:       c6 05 dd 2e 00 00 01    movb   $0x1,0x2edd(%rip)        # 4010 <__TMC_END__>
    1133:       5d                      pop    %rbp
    1134:       c3                      retq
    1135:       0f 1f 00                nopl   (%rax)
    1138:       c3                      retq
    1139:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001140 <frame_dummy>:
    1140:       f3 0f 1e fa             endbr64
    1144:       e9 77 ff ff ff          jmpq   10c0 <register_tm_clones>

0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 3d ac 0e 00 00    lea    0xeac(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    1158:       e8 f3 fe ff ff          callq  1050 <puts@plt>
    115d:       b8 00 00 00 00          mov    $0x0,%eax
    1162:       5d                      pop    %rbp
    1163:       c3                      retq
    1164:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    116b:       00 00 00
    116e:       66 90                   xchg   %ax,%ax

0000000000001170 <__libc_csu_init>:
    1170:       f3 0f 1e fa             endbr64
    1174:       41 57                   push   %r15
```

mov rax, rbx → **Assembler** → **0x48 0x89 0xd8**

What is Assembly Language? Cont.

- Assembly language changes based on your processor architecture
  - x86 instructions are different from ARM, MIPS, RISC-V, …
- This is why you may have heard some computers “speak different languages”



MIPS load/store instruction datapath

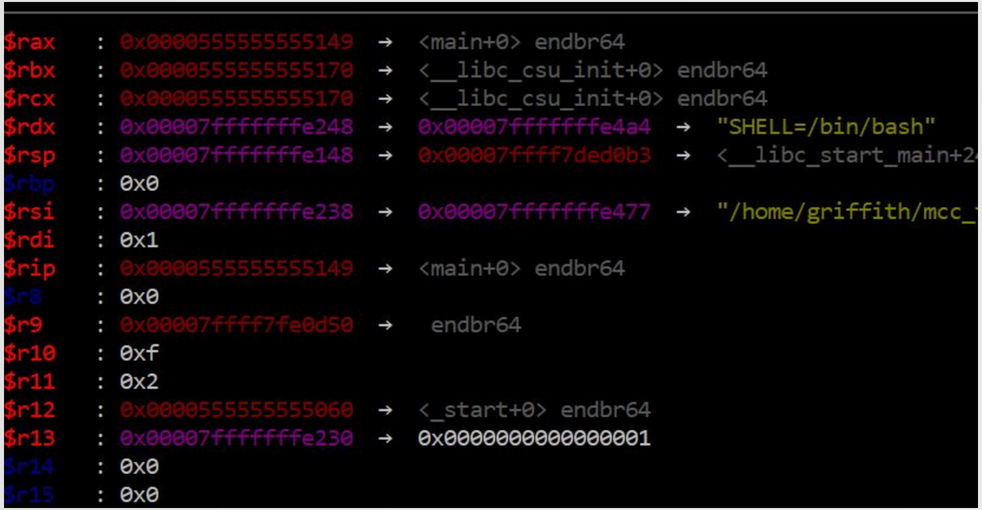
https://www.cise.ufl.edu/~mssz/CompOrg/Figure4.9-MIPSdatapathBranch.gif

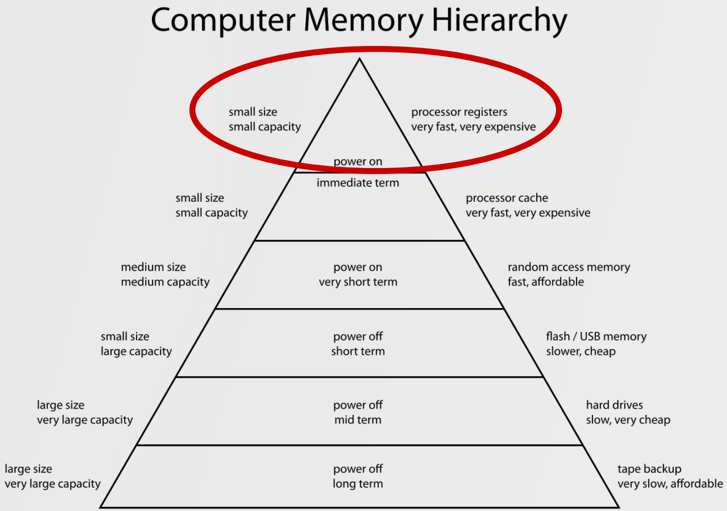
Why Should I Care About Assembly?

- Who wants to take the time to learn a frustrating caveman language?
- Mastery of assembly means mastery of the low-level computer stack, very impressive!
- Still used in some low-memory embedded systems environments
- Useful for understanding a program's behaviour for debugging purposes
- Pwn and RE categories are in almost all CTFs
- If you want to reverse engineer a pre-compiled binary, reading assembly is your best option without getting into some advanced software

CPU Registers

- The fastest memory your computer has. Baked right on top of the CPU.
- Very small, only enough room for the most important information
  - x64 has 16 64-bit registers (128 bytes)
  - Typically holds the current instruction and some stack information. Can also store local variables, function arguments, and more. These also vary by architecture.





Common Instructions

- Mov *S* , *D* - Move data from source to destination

- Add *S* , *D* - add source to destination
- Sub *S* , *D* - subtract source from destination

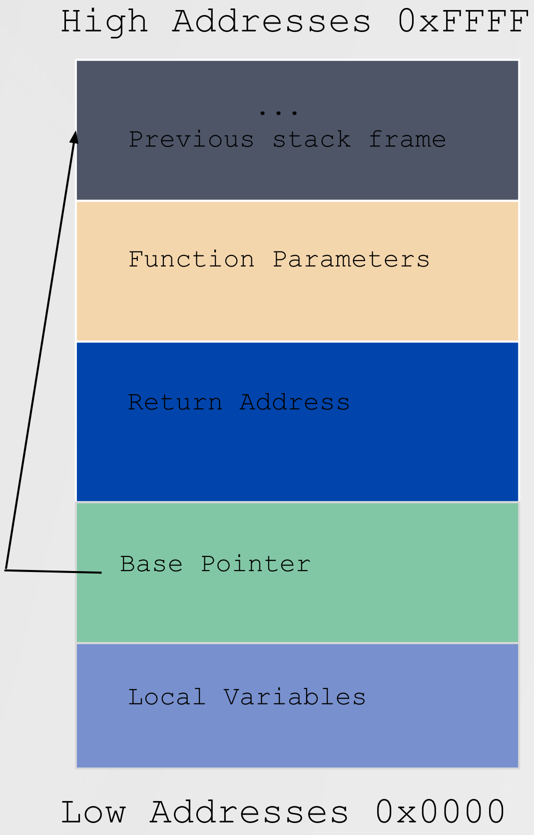
- and/or/xor *S* , *D* - bitwise operations on destination by source

- Jmp *Label* - jump to execute at *label*
- Jg/Jl *Label* - jump if greater/less than

Nice x64 cheatsheet: https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Stack Frame

- What if we want to store something in memory but it can’t fit in a register? Put it on the stack!
- What exactly goes on stack varies by architecture but typically includes at least return address, base pointer, and some local variables.
- Each function gets its own stack frame set up for itself when it gets called
- The return address points to the code that will execute after this frame is closed
    - Hmmm, that sounds interesting
- Base pointer holds the address of the top of the stack frame and is used to reference the local variables



C to Assembly

```c
#include<stdio.h>

int main(){
  puts("hello world!\n");
}
```

Disassembled

```
    10eb:       48 85 c0                test   %rax,%rax
    10ee:       74 08                   je     10f8 <register_tm_clones+0x38>
    10f0:       ff e0                   jmpq   *%rax
    10f2:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
    10f8:       c3                      retq
    10f9:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001100 <__do_global_dtors_aux>:
    1100:       f3 0f 1e fa             endbr64
    1104:       80 3d 05 2f 00 00 00    cmpb   $0x0,0x2f05(%rip)        # 4010 <__TMC_END__>
    110b:       75 2b                   jne    1138 <__do_global_dtors_aux+0x38>
    110d:       55                      push   %rbp
    110e:       48 83 3d e2 2e 00 00    cmpq   $0x0,0x2ee2(%rip)        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
    1115:       00
    1116:       48 89 e5                mov    %rsp,%rbp
    1119:       74 0c                   je     1127 <__do_global_dtors_aux+0x27>
    111b:       48 8b 3d e6 2e 00 00    mov    0x2ee6(%rip),%rdi        # 4008 <__dso_handle>
    1122:       e8 19 ff ff ff          callq  1040 <__cxa_finalize@plt>
    1127:       e8 64 ff ff ff          callq  1090 <deregister_tm_clones>
    112c:       c6 05 dd 2e 00 00 01    movb   $0x1,0x2edd(%rip)        # 4010 <__TMC_END__>
    1133:       5d                      pop    %rbp
    1134:       c3                      retq
    1135:       0f 1f 00                nopl   (%rax)
    1138:       c3                      retq
    1139:       0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000001140 <frame_dummy>:
    1140:       f3 0f 1e fa             endbr64
    1144:       e9 77 ff ff ff          jmpq   10c0 <register_tm_clones>

0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 3d ac 0e 00 00    lea    0xeac(%rip),%rdi        # 2004 <_IO_stdin_used+0x4>
    1158:       e8 f3 fe ff ff          callq  1050 <puts@plt>
    115d:       b8 00 00 00 00          mov    $0x0,%eax
    1162:       5d                      pop    %rbp
    1163:       c3                      retq
    1164:       66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
    116b:       00 00 00
    116e:       66 90                   xchg   %ax,%ax

0000000000001170 <__libc_csu_init>:
    1170:       f3 0f 1e fa             endbr64
    1174:       41 57                   push   %r15
    1176:       4c 8d 3d 3b 2c 00 00    lea    0x2c3b(%rip),%r15        # 3db8 <__frame_dummy_init_array_entry>
    117d:       41 56                   push   %r14
    117f:       49 89 d6                mov    %rdx,%r14
    1182:       41 55                   push   %r13
    1184:       49 89 f5                mov    %rsi,%r13
    1187:       41 54                   push   %r12
    1189:       41 89 fc                mov    %edi,%r12d
    118c:       55                      push   %rbp
    118d:       48 8d 2d 2c 2c 00 00    lea    0x2c2c(%rip),%rbp        # 3dc0 <__do_global_dtors_aux_fini_array_entry>
    1194:       53                      push   %rbx
```

C to Assembly cont.

```
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   rbp
   0x000000000000114e <+5>:     mov    rbp,rsp
   0x0000000000001151 <+8>:     lea    rdi,[rip+0xeac]        # 0x2004
   0x0000000000001158 <+15>:    call   0x1050 <puts@plt>
   0x000000000000115d <+20>:    mov    eax,0x0
   0x0000000000001162 <+25>:    pop    rbp
   0x0000000000001163 <+26>:    ret
End of assembler dump.
```

Save previous frame pointer

Frame pointer is also the top of our stack

Move string "Hello world!" Into rdi register

Call puts function

Close stack frame and return execution to previous location

Loops in Assembly

```c
#include<stdio.h>

int main(){
   int i;
   for(i = 0; i < 10; i++){
         puts("hello\n");
   }
}
```

Disassembled

```
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   rbp
   0x000000000000114e <+5>:     mov    rbp,rsp
   0x0000000000001151 <+8>:     sub    rsp,0x10
   0x0000000000001155 <+12>:    mov    DWORD PTR [rbp-0x4],0x0
   0x000000000000115c <+19>:    jmp    0x116e <main+37>
   0x000000000000115e <+21>:    lea    rdi,[rip+0xe9f]        # 0x2004
   0x0000000000001165 <+28>:    call   0x1050 <puts@plt>
   0x000000000000116a <+33>:    add    DWORD PTR [rbp-0x4],0x1
   0x000000000000116e <+37>:    cmp    DWORD PTR [rbp-0x4],0x9
   0x0000000000001172 <+41>:    jle    0x115e <main+21>
   0x0000000000001174 <+43>:    mov    eax,0x0
   0x0000000000001179 <+48>:    leave
   0x000000000000117a <+49>:    ret
End of assembler dump.
```

Loops in Assembly cont.

```
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   rbp
   0x000000000000114e <+5>:     mov    rbp,rsp
   0x0000000000001151 <+8>:     sub    rsp,0x10
   0x0000000000001155 <+12>:    mov    DWORD PTR [rbp-0x4],0x0
   0x000000000000115c <+19>:    jmp    0x116e <main+37>
   0x000000000000115e <+21>:    lea    rdi,[rip+0xe9f]        # 0x2004
   0x0000000000001165 <+28>:    call   0x1050 <puts@plt>
   0x000000000000116a <+33>:    add    DWORD PTR [rbp-0x4],0x1
   0x000000000000116e <+37>:    cmp    DWORD PTR [rbp-0x4],0x9
   0x0000000000001172 <+41>:    jle    0x115e <main+21>
   0x0000000000001174 <+43>:    mov    eax,0x0
   0x0000000000001179 <+48>:    leave
   0x000000000000117a <+49>:    ret
End of assembler dump.
```

Make room for local variable "i"

Initialize i to 0

Jump to comparison

Increment i

Continue loop if i < 9

Assembly Conclusion

- Assembly sucks, wouldn’t it be great if there was a better way of pulling apart programs?
- Learning assembly is crucial for understanding the most low-level operations of a computer

Surely there must be a better way! This is the 21st century!

There is.

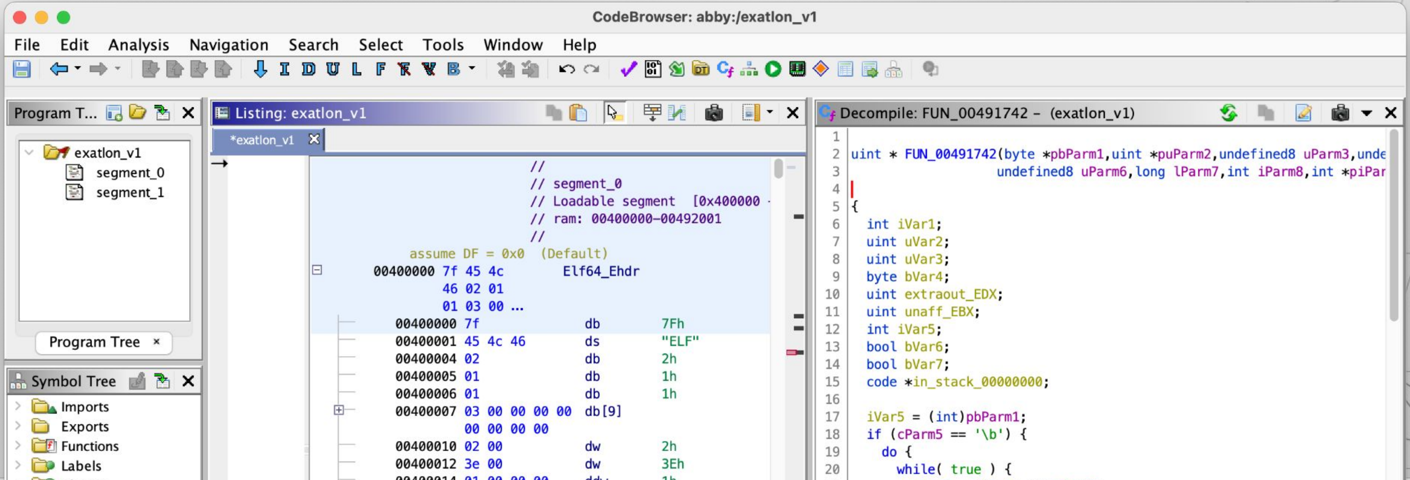
REAL Reverse Engineering

Time to bring out the big guns.

What is Reverse Engineering?

- Taking apart software without any source code and trying to figure out how it works
- A very useful and in-demand skill to have in the security field
- If we can pull apart a program and find vulnerabilities and exploits, so could evil hackers!

How can we use what we know to find exploits? Can we automate exploit discovery?



Decompilers

- Best invention of the century
- Analyze binaries and shows you the disassembly, but also shows you a nice view of what it **thinks** the source code may have looked like
- MUCH easier to read than assembly, but not always accurate

Decompile: main – (hello_world)

```c

undefined8 main(void)

{
  _puts("Hello world!\n");
  return 0;
}

```

Decompilers cont

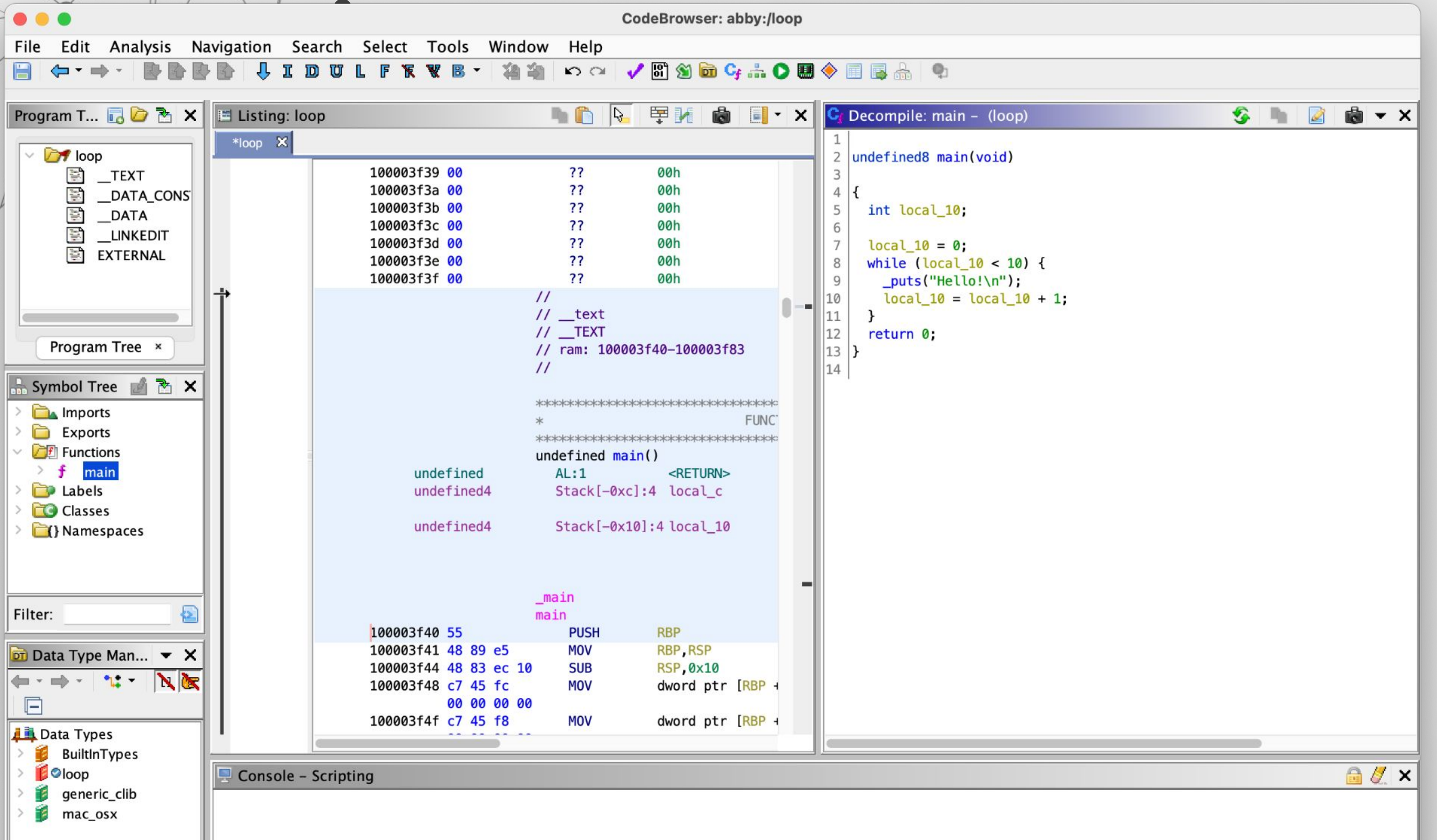
- Comes in many flavors
- You should start with Ghidra because it's free and open source

https://www.ninefx.com/img/products/ida_pro.png

https://upload.wikimedia.org/wikipedia/commons/a/a3/Ghidra_Logo.png

https://binary.ninja/ico/apple-icon-180x180.png

Much better!



CodeBrowser: abby:/loop
File Edit Analysis Navigation Search Select Tools Window Help
Program T...
loop
__TEXT
__DATA_CONS
__DATA
__LINKEDIT
EXTERNAL
Program Tree
Symbol Tree
Imports
Exports
Functions
main
Labels
Classes
Namespaces
Filter:
Data Type Man...
Data Types
BuiltInTypes
loop
generic_clib
mac_osx
Listing: loop
*loop
100003f39 00 ?? 00h
100003f3a 00 ?? 00h
100003f3b 00 ?? 00h
100003f3c 00 ?? 00h
100003f3d 00 ?? 00h
100003f3e 00 ?? 00h
100003f3f 00 ?? 00h
//
// __text
// __TEXT
// ram: 100003f40-100003f83
//
undefined main()
undefined AL:1 <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined4 Stack[-0x10]:4 local_10
_main
main
100003f40 55 PUSH RBP
100003f41 48 89 e5 MOV RBP,RSP
100003f44 48 83 ec 10 SUB RSP,0x10
100003f48 c7 45 fc MOV dword ptr [RBP +
00 00 00 00
100003f4f c7 45 f8 MOV dword ptr [RBP +
Decompile: main – (loop)
undefined8 main(void)
{
int local_10;
local_10 = 0;
while (local_10 < 10) {
_puts("Hello!\n");
local_10 = local_10 + 1;
}
return 0;
}
Console - Scripting

GDB

- Viewing the decompilation is a form of static analysis
- What if we want to view what a program is doing while it’s running? We want dynamic analysis!
- And the best tool for that is the GNU Debugger (GDB)!
  - If you want to be really cool, use the GEF extension to get the most out of GDB
- From the GDB website:

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- **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 saw screenshots of GDB with GEF on previous slides!

GDB w/ GEF

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x0000555555555149  →  <main+0> endbr64
$rbx   : 0x0000555555555170  →  <__libc_csu_init+0> endbr64
$rcx   : 0x0000555555555170  →  <__libc_csu_init+0> endbr64
$rdx   : 0x00007fffffffe278  →  0x00007fffffffe555  →  "SHELL=/bin/bash"
$rsp   : 0x00007fffffffe178  →  0x00007ffff7ded0b3  →  <__libc_start_main+243> mov edi, eax
$rbp   : 0x0
$rsi   : 0x00007fffffffe268  →  0x00007fffffffe531  →  "/home/griffith/assembly/hello_world"
$rdi   : 0x1
$rip   : 0x0000555555555149  →  <main+0> endbr64
$r8    : 0x0
$r9    : 0x00007ffff7fe0d50  →   endbr64
$r10   : 0x0
$r11   : 0x0
$r12   : 0x0000555555555060  →  <_start+0> endbr64
$r13   : 0x00007fffffffe260  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
──────────────────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffe178│+0x0000: 0x00007ffff7ded0b3  →  <__libc_start_main+243> mov edi, eax  ← $rsp
0x00007fffffffe180│+0x0008: 0x00007ffff7ffc620  →  0x0005081200000000
0x00007fffffffe188│+0x0010: 0x00007fffffffe268  →  0x00007fffffffe531  →  "/home/griffith/assembly/hello_world"
0x00007fffffffe190│+0x0018: 0x0000000100000000
0x00007fffffffe198│+0x0020: 0x0000555555555149  →  <main+0> endbr64
0x00007fffffffe1a0│+0x0028: 0x0000555555555170  →  <__libc_csu_init+0> endbr64
0x00007fffffffe1a8│+0x0030: 0xefd2b5c60c0187b7
0x00007fffffffe1b0│+0x0038: 0x0000555555555060  →  <_start+0> endbr64
────────────────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x555555555139 <__do_global_dtors_aux+57> nop    DWORD PTR [rax+0x0]
   0x555555555140 <frame_dummy+0>  endbr64
   0x555555555144 <frame_dummy+4>  jmp    0x5555555550c0 <register_tm_clones>
 → 0x555555555149 <main+0>         endbr64
   0x55555555514d <main+4>         push   rbp
   0x55555555514e <main+5>         mov    rbp, rsp
   0x555555555151 <main+8>         lea    rdi, [rip+0xeac]        # 0x555555556004
   0x555555555158 <main+15>        call   0x555555555050 <puts@plt>
   0x55555555515d <main+20>        mov    eax, 0x0
────────────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "hello_world", stopped 0x555555555149 in main (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x555555555149 → main()
───────────────────────────────────────────────────────────────────────────────────────────────
gef➤
```

Concolic Analysis

- Super modern technology allows us to combine both static and dynamic execution to create: concolic analysis
- Angr is a concolic analysis framework that uses the Z3 theorem prover to essentially simulate the execution of a binary, giving us the ability to analyze maximum code coverage and use symbolic execution to automate exploit discovery

This could be its own talk (or 4 credit course) so check out their documentation if you're interested.

If you flex your Angr ability in an interview, jaws will hit the floor.

https://angr.io/img/angry_face.png

Proud Sponsors

CACI

EVER VIGILANT

VOLGENAU SCHOOL

MASON

ENGINEERING

LIVE DEMO