

# Introduction to Software Reverse Engineering with Ghidra Session 1

Hackaday U  
Matthew Alt



# #whoami

- Reverse engineer focused on embedded systems
- Security researcher for Caesar Creek Software
- Provide training and assessments through VoidStar
  - [voidstarsec.com](https://voidstarsec.com)
- [@wrongbaud](https://twitter.com/wrongbaud)
  - [wrongbaud.github.io](https://wrongbaud.github.io)



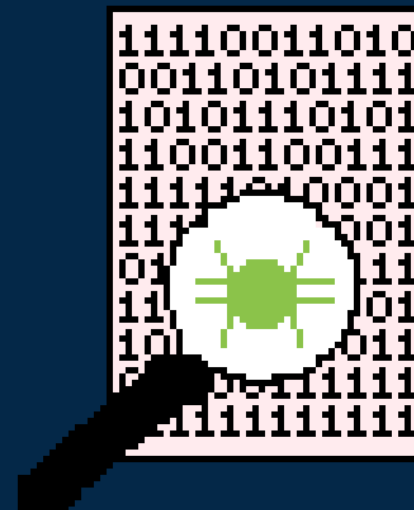
# #Outline

- What is Software Reverse Engineering (SRE)?
  - Software Engineering Review
- SRE 101
  - Extracting Information from Compiled Programs
  - Disassembly / x86 ASM Refresher
- Ghidra 101:
  - Installation
  - Basic Usage and Navigation
- Exercises:
  - Challenge 1/2
- Conclusion / Questions



# #What is SRE?

- Analyzing a software system to extract information
  - Source code not available
- Used to recreate and understand functionality
  - Also used to find bugs!
- Often started from the lowest layer of abstraction
  - Machine code
  - We will be focusing on x86\_64 ELF binaries for Linux



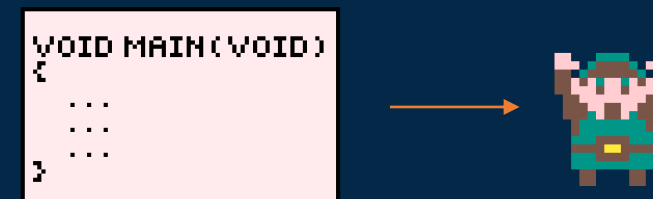
# #Software Engineering Review

- Developers write code in high level languages such as C/C++
- This code is then compiled into machine code – sequences of bytes that the CPU can interpret
- Disassembly is the process of converting these byte sequences into assembly instructions
- As reverse engineers, these byte sequences will be our starting point



# #Compilation Review

- Compiling a program is a multi-stage process\*
  - Preprocessing
  - Compilation
  - Assembly
  - Linking
- The result is machine code that is run on the CPU
- These steps are all typically performed automatically
- After going through these steps, an executable is produced



\* Disclaimer: These are all extremely complex fields of research, and we're only covering a very high level view



# #Compilers

- Compiling is phase two of “compilation”
  - Preprocessing passes over the source code, performing:
    - Comment removal
    - Macro Expansion
    - Include Expansion
    - Conditional Compilation (IFDEF)
- Compiling converts the output of preprocessor into assembly instructions



# #Compilers – An Example

## C Code

```
#include <stdio.h>

int main(){
    printf("Hello!");
    return 0;
}
```

## Assembly Code

```
.LC0:
    .string      "Hello!"
    .text
    .globl       main
    .type        main, @function

main:
.LFB0:
    .cfi_startproc
    pushq       %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq        %rsp, %rbp
    .cfi_def_cfa_register 6
    movl        $.LC0, %edi
    movl        $0, %eax
    call        printf
    movl        $0, %eax
    popq        %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```





# #Assemblers

- Assemblers convert the assembly code into binary opcodes
- Each instruction is represented by a binary opcode
  - `mov rax,1 = 0x48C7C001000000`
- The assembler will produce an object file
  - Object files contain machine code
  - This file will contain fields to be filled by the linker



# #Assemblers – An Example

## Assembly Code

```
.LC0:
    .string      "Hello!"
    .text
    .globl       main
    .type        main, @function

main:
.LFB0:
    .cfi_startproc
    pushq        %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq         %rsp, %rbp
    .cfi_def_cfa_register 6
    movl         $.LC0, %edi
    movl         $0, %eax
    call         printf
    movl         $0, %eax
    popq         %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

## Assembled Bytecode

```
55 48 89 e5 bf 00 00 00 00 b8 00 00 00 00 e8 00
00 00 00 b8 00 00 00 00 5d c3 48 65 6c 6c 6f 21
```



# #Linking

- More is needed before the object code can be executed
  - Entry point, or starting instruction must be defined
- Used to define memory regions on embedded platforms
  - Often done through linker scripts
- The result of linking is the final executable program



# #Linking – An Example

```
$ objdump -x session1.o

session1.o:      file format elf64-x86-64
session1.o
architecture: i386:x86-64, flags 0x0000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

Sections:
Idx Name          Size      VMA           LMA           File off  Algn
 0 .text          0000001a  0000000000000000 0000000000000000 00000040 2**0
CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
 1 .data          00000000  0000000000000000 0000000000000000 0000005a 2**0
CONTENTS, ALLOC, LOAD, DATA
 2 .bss           00000000  0000000000000000 0000000000000000 0000005a 2**0
ALLOC
 3 .rodata        00000007  0000000000000000 0000000000000000 0000005a 2**0
CONTENTS, ALLOC, LOAD, READONLY, DATA
 4 .comment       0000002c  0000000000000000 0000000000000000 00000061 2**0
CONTENTS, READONLY
 5 .note.GNU-stack 00000000  0000000000000000 0000000000000000 0000008d 2**0
CONTENTS, READONLY
 6 .eh_frame      00000038  0000000000000000 0000000000000000 00000090 2**3
CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA

SYMBOL TABLE:
0000000000000000 l  df *ABS*  0000000000000000 session1.c
0000000000000000 l  d  .text  0000000000000000 .text
0000000000000000 l  d  .data  0000000000000000 .data
0000000000000000 l  d  .bss   0000000000000000 .bss
0000000000000000 l  d  .rodata 0000000000000000 .rodata
0000000000000000 l  d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l  d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l  d  .comment 0000000000000000 .comment
0000000000000000 g  F  .text  000000000000001a main
0000000000000000      *UND*  0000000000000000 printf
```

gcc -o session1 session1.o

```
$ objdump -x session1

session1:      file format elf64-x86-64
session1
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000000040270

Program Header:
  PHDR off  0x0000000000000040 vaddr 0x0000000000004000 paddr 0x0000000000004000 align 2**3
    filesz 0x0000000000000150 memsz 0x0000000000000150 flags r-x
  INTERP off 0x0000000000000190 vaddr 0x0000000000004010 paddr 0x0000000000004010 align 2**0
    filesz 0x000000000000000f memsz 0x000000000000000f flags r--
  LOAD off  0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000004000 align 2**21
    filesz 0x00000000000002f8 memsz 0x00000000000002f8 flags r-x
  LOAD off  0x00000000000002f8 vaddr 0x000000000000602f8 paddr 0x000000000000602f8 align 2**21
    filesz 0x0000000000000160 memsz 0x0000000000000160 flags rw-
  DYNAMIC off 0x00000000000002f8 vaddr 0x000000000000602f8 paddr 0x000000000000602f8 align 2**3
    filesz 0x0000000000000140 memsz 0x0000000000000140 flags rw-
  STACK off  0x0000000000000000 vaddr 0x0000000000000000 paddr 0x0000000000000000 align 2**4
    filesz 0x0000000000000000 memsz 0x0000000000000000 flags rw-

Dynamic Section:
NEEDED               libc.so.6
HASH                 0x000000000000401a0
STRTAB               0x000000000000401e8
SYMTAB               0x000000000000401b8
STRSZ                0x000000000000001e
SYMENT               0x0000000000000018
DEBUG                0x0000000000000000
PLTGOT               0x00000000000060438
PLTRELSZ             0x0000000000000018
PLTREL               0x0000000000000007
JMPREL               0x00000000000040230
VERNEED              0x00000000000040210
VERNEEDNUM           0x0000000000000001
VERSYM               0x00000000000040206

Version References:
required from libc.so.6:
```



# #Output Formats

- The output of the compilation process can take many forms:
  - PE (Windows)
  - ELF (Linux)
  - Mach-O (OSX)
  - COFF/ECOFF
- This output file is often your starting point as a reverse engineer
- For this course we will focus on the ELF format



# #ELF Files – An Overview

- ELF = Executable Linking Format
- Contains information identifying:
  - OS,endianness,etc
- ELF files provide information needed for execution by the OS
- ELF Files can be broken up into three components
  - ELF Header
  - Sections
  - Segments



# #ELF Files: Symbols

- Symbols are used to aid in debugging and provide context to the loader
  - The removal of these symbols makes things more difficult to reverse engineer
- ELF objects contain a maximum of two symbol tables
  - .symtab: Symbols used for debugging / labelling (useful for RE!)
  - .dynsym: Contains symbols needed for dynamic linking



# #ELF Files: A Review

- ELF files define how the program is laid out in memory
  - Used by the OS loader to create a process
- ELF files contain machine code that we will be reverse engineering
- Many tools exist to analyze and read ELF files:
  - dumpelf
  - readelf
  - objdump
  - elfutils (package containing multiple utilities)





# #SE Review: Pixelated Edition

```
VOID MAIN(VOID)
{
    ...
    ...
    ...
}
```

Compile

```
XOR EAX,EAX
MOV ECK, 10
LABEL:
INX EAX
LOOP LABEL
...
...
```

Assemble

```
3100B90A
00000000
FFC1E2FC
...
...
...
```



# #SE Review: Pixelated Edition

```
31C0E90A  
00000000  
FFC1E2FC  
...  
...  
...
```

Link



# #Intermission: Why Review this?

- Information can be limited when performing SRE
  - Understanding core concepts is important
  - File formats can be a treasure trove of information
- Our goal is to work backwards from machine code
  - The ELF file will contain machine code
  - This machine code can be converted BACK into assembly language!
  - Machine code -> Assembly Language = Disassembly!

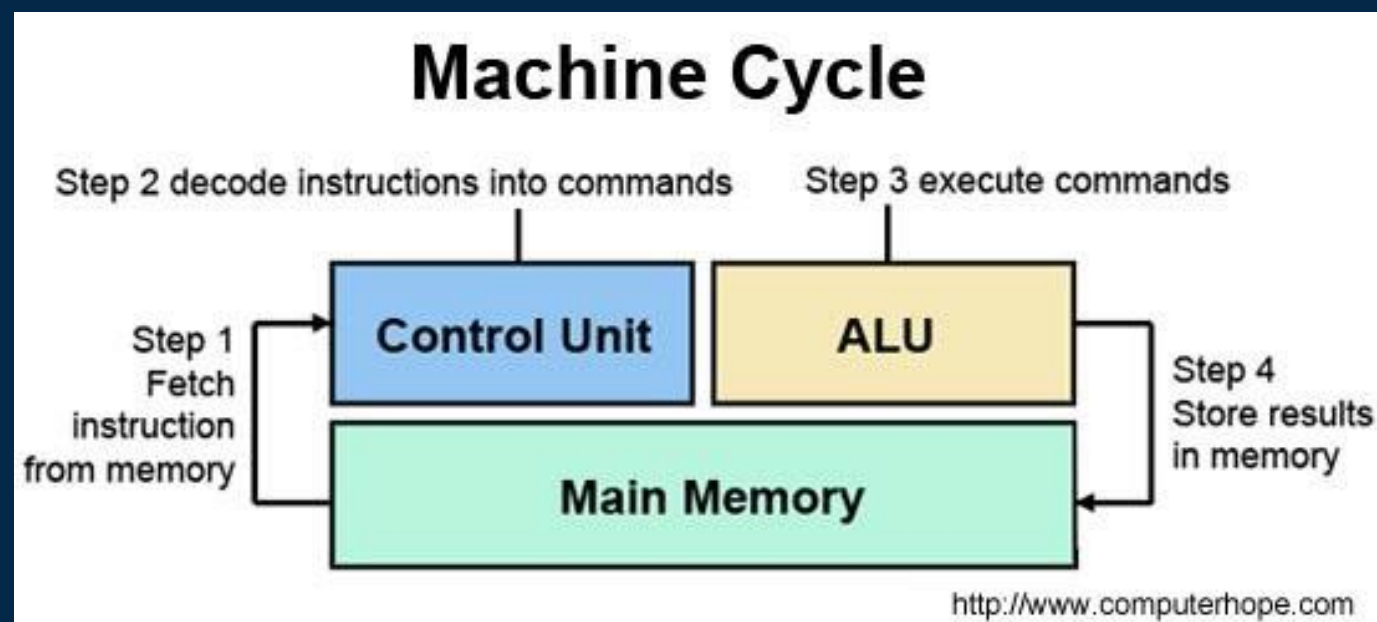


# #Computer Architecture 101

- When a program is running, the following must happen:
  1. An instruction is read into memory
  2. The instruction is processed by the Arithmetic Logic Unit
  3. The result of the operation is stored into registers or memory
- For this course, we'll deconstruct C programs into four core components
  - Registers
  - Instructions
  - Stack
  - Heap



# # Computer Architecture 101



# #x86\_64 Architecture

- We will focus on Intel's x86-64 instruction set
  - 64 bit version of the x86 instruction set
  - Contains multiple operating modes for backwards compatibility
- Original specification was created by AMD in 2000
- Commonly used in desktop and laptop computers



# #x86\_64: Registers

- Registers are small storage areas used by the processor
- x86\_64 assembly uses 16 64 bit general purpose registers (R8-15 not in table)

Register Name	64 Bit	32 Bit	16 Bit	8 Bit
R0	RAX	EAX	AX	AH/AL
R1	RCX	ECX	CX	CH/CL
R2	RDX	EDX	DX	DH/DL
R3	RBX	EBX	BX	BH/BL
R4	RSP	ESP	SP	
R5	RBP	EBP	BP	
R6	RSI	ESI	SI	
R7	RDI	EDI	DI	



# #x86\_64: Registers

- RIP: Instruction pointer
  - Points to the next instruction to be executed
  - 64 bits in width
- RFLAGS: Stores flags used for processor flow control
- FPR0-FPR7: Floating point status and control registers
- RBP/RSP: Stack manipulation and usage





# #x86\_64 Instructions

- These define the operations being performed by the CPU
- For this course will be using the Intel syntax
  - instruction dest, source
- Instructions can have multiple operands
  - These define the arguments for the specified operation
- x86\_64 has a large amount of available instructions
  - We will focus on commonly used ones to start



## #x86\_64 Instructions: mov

- Moves data from one register to another

```
mov rax, rbx
```

- Moves the value stored in RBX to RAX

```
mov rax, [rcx]
```

- Moves the value *pointed* to by RCX into RAX



# #x86\_64 Instructions: add/sub

- Add: Adds the two values together, storing the result in the first argument
  - `add rax, rbx`
    - Adds `rbx` to `rax`, the result is stored in `rax`
    - `rax += rbx`
- Sub: Subtracts the second operand from the first one, storing the result in the first operand
  - `sub rax, rbx`
    - Subtracts `rbx` from `rax`, stores the result in `rax`
    - `rax -= rbx`



# #x86\_64 Instructions: and/xor

- **AND:** Performs the binary operation AND on the two operands, storing the result in the first
  - `and rax,rbx`
    - `rax = rax & rbx`
- This syntax is used for other binary operations as well:
  - `xor`
  - `or`



# #x86\_64: The Stack

- Data structure containing elements in contiguous memory
  - POP: Reads from stack
  - PUSH: Writes to stack
- Elements are removed in the reverse order that they are added
- Grows high to low
- RSP points to top of stack
- RBP contains base pointer

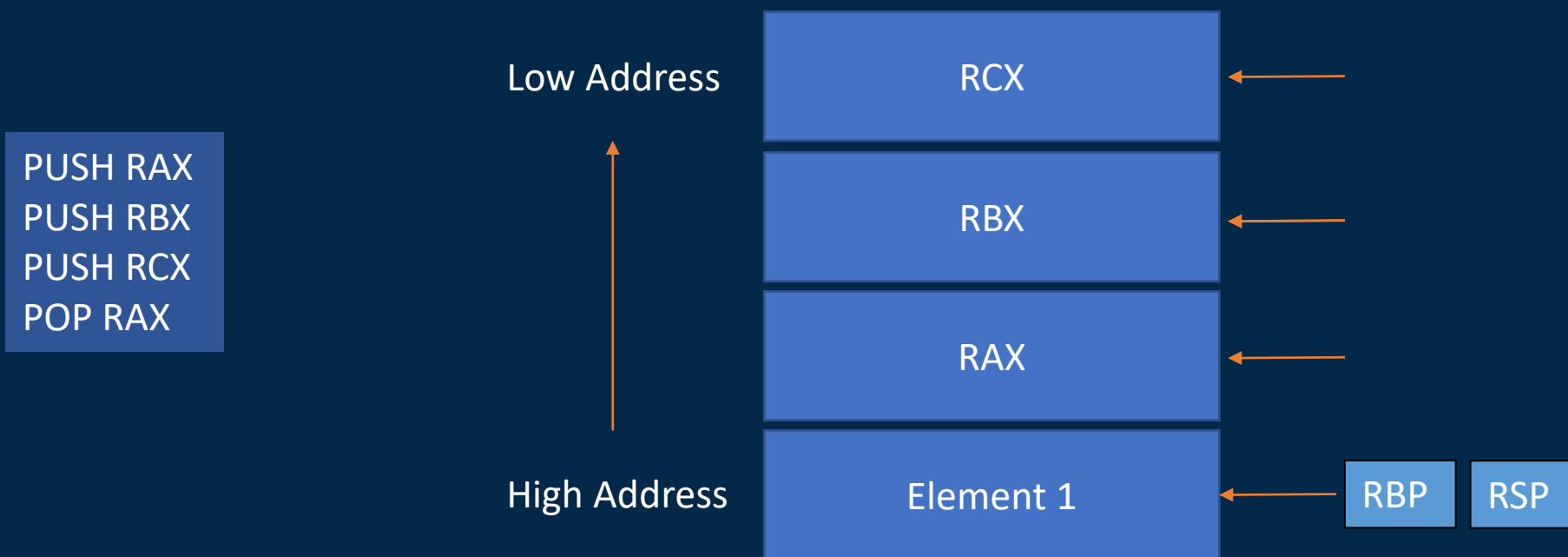


# #x86\_64 Instructions: push/pop

- push will grow the stack by 8 and store the operand contents on the stack
  - push rax
    - Increases the value pointed to by rsp by 8, and stores rax there
- pop will load the value pointed to by rsp into the operand
  - pop rbx
    - Loads the value pointed by rsp into rbx, and decreases rsp by 8



# #x86\_64: The Stack



# #x86\_64 Instructions: jmp/call

- `jmp` is used to change what code is being executed
  - Modifies the value in the instruction pointer (`rip/eip`)
  - `jmp 0x1000300`
    - Set instruction pointer to `0x1000300` and execute the instructions there
- `call` is used to implement function calls
  - Pushes value of `rip` onto stack before jumping
  - `call 0x18000000`





# #x86\_64 Instructions: cmp

- cmp performs a comparison operation by subtracting the operands
  - No storage is performed (unlike sub)
  - Based on the result, fields in RFLAGS are set!
  - cmp rax, #5
- The flags in RFLAGS register are used by jmp variants
  - jnz: Jump if not zero
  - jz: Jump if zero



# #x86\_64: Addressing Modes

- Instructions can access registers and memory in various modes
- Immediate: The value is stored in the instruction
  - `add rax,14`; stores `rax+14` into RAX
- Register to Register
  - `xor rax,rax`; clears the value in RAX
- Indirect Access:
  - `add rax, [rbx]`; adds the value *pointed* to by `rbx` into `rax`
  - `mov rbx, 1234[8*rax+rcx]`
    - move word at address `8*RAX+RCX+1234` into `rbx`



# #x86\_64 Instructions Exercise

```
section .text
```

```
    global _start
_start:
    mov rax, 0x2FFF
    mov rbx, 0x3000
    or rax,rbx
    mov rcx, 0x10000
    sub rcx, rax
    add rcx, rbx
    cmp rax,rbx
    jg _greater
    mov rax, 0x2
_greater:
    mov rax, 0x1
    ret
```

Register	Value
RAX	1
RBX	0x3000
RCX	0xF001
RIP	_greater +5



# #x86\_64: Wrap up

- x86\_64 is a very complicated architecture
  - We've only covered the bare minimum
- Instructions and other reference material can be found on Intel's [website](#)
- Although Ghidra has a decompiler, it is important to understand the underlying assembly



# #Ghidra: Overview

- Open source SRE tool developed by NSA
  - Released in March 2019
  - Written in Java
  - **Free**
- Provides a disassembler and decompiler
  - Large library of supported processors / architectures
  - Custom processors can be added via SLEIGH modules
- Active development community
  - 146 PRs, 2,530 commits



# #Ghidra: Installation

- Download the latest release from <https://ghidra-sre.org/>
  - For this course we will use v9.1.2
- Unzip the installation bundle
  - This contains everything you need to run Ghidra
  - Unzip to somewhere accessible
- Install Java 11 64-bit Runtime and Development Kit (JDK)
- Launch Ghidra!
  - `./ghidraRun.sh` or `./ghidraRun.bat`

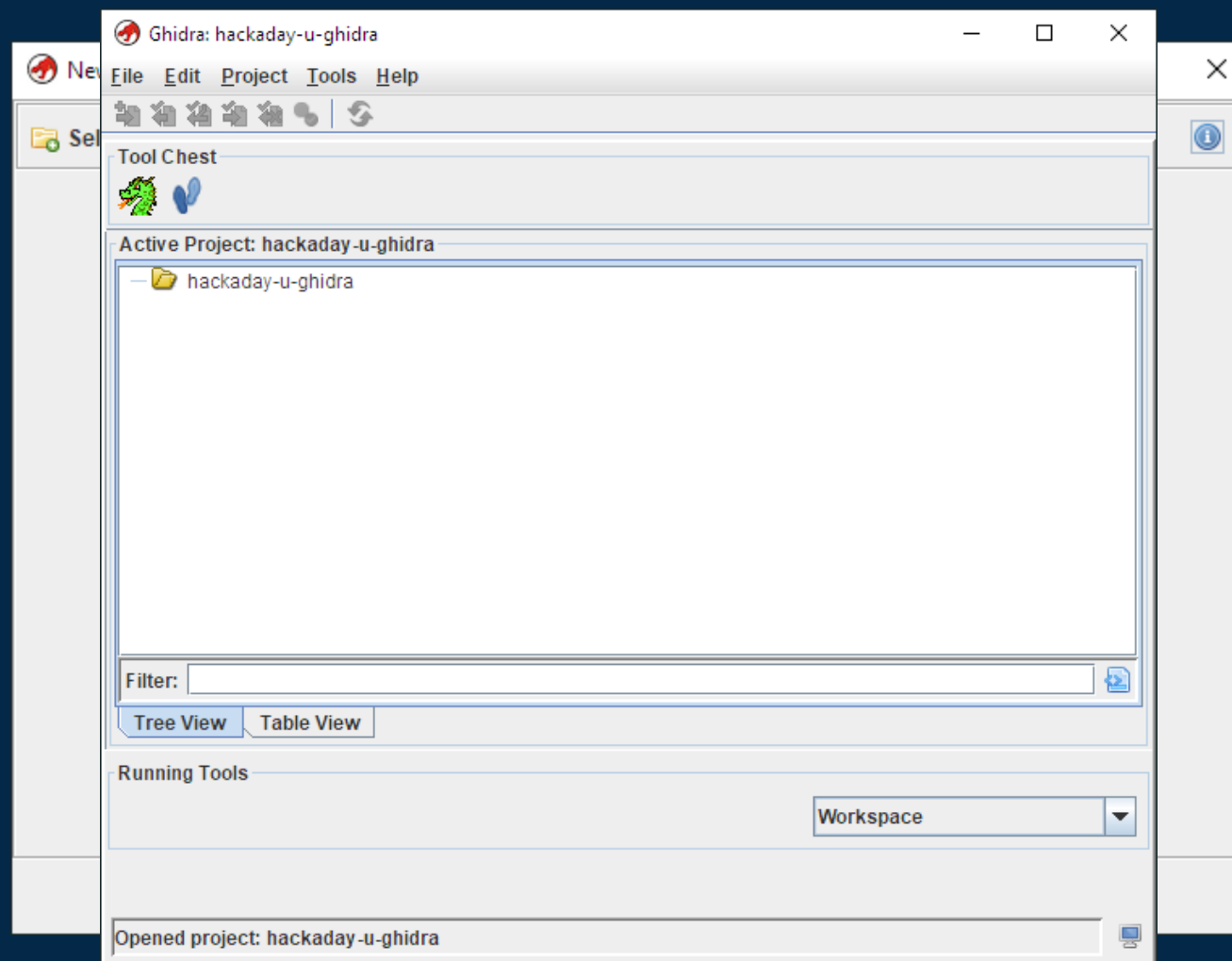


# #Ghidra: Creating a Project

- Ghidra groups binaries into projects
  - Projects can be shared across multiple users
- Programs and binaries can be imported into a project
- File -> New Project
  - Non-Shared Project
  - Select Directory
  - Name the project: “hackaday-u-ghidra”



# #Ghidra: Creating a Project



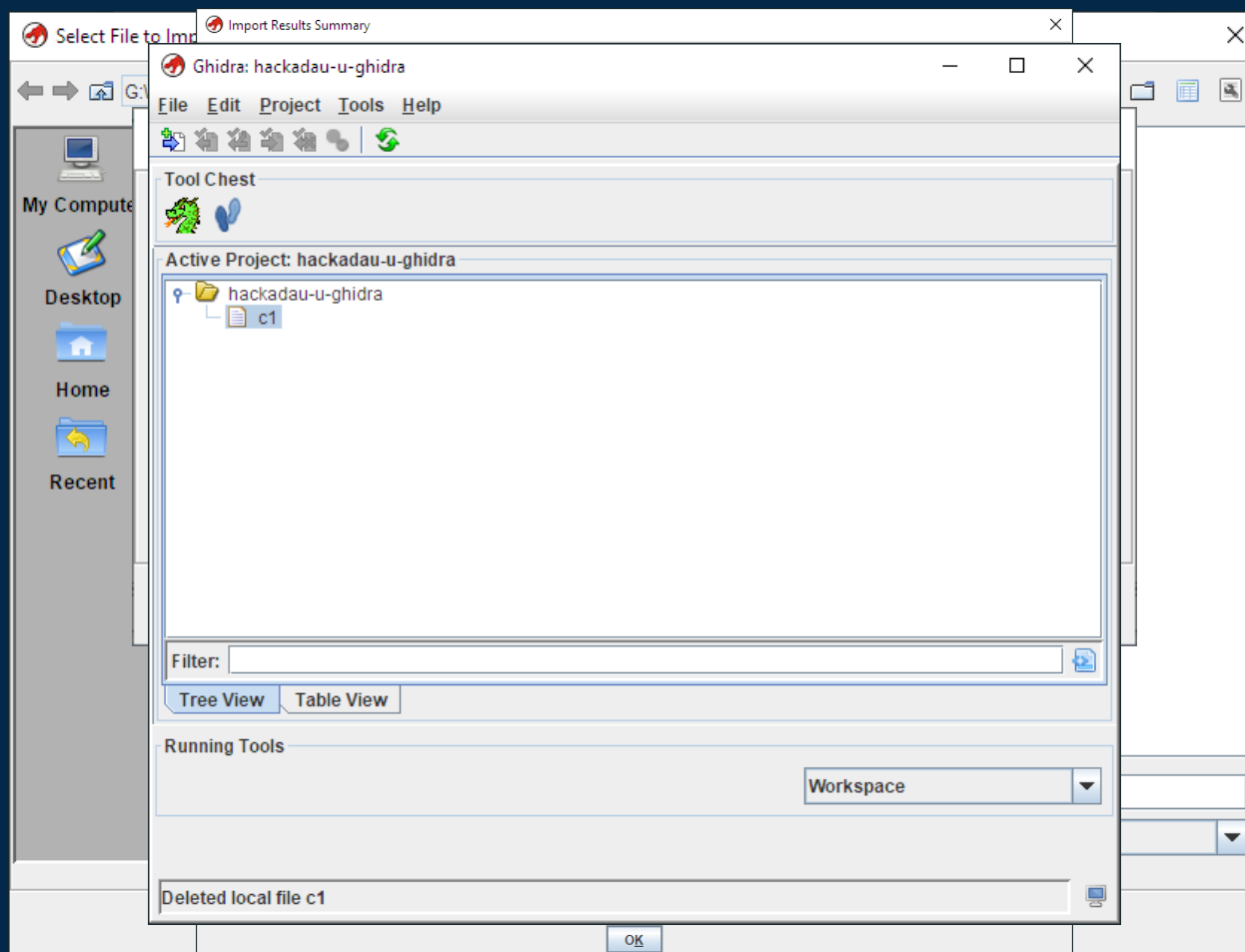


# #Ghidra: Loading a Binary

- Import Window
  - In this window you can inform Ghidra about the target binary
  - Architecture / Language
  - File format
- Ghidra will attempt to autodetect features based on the file format
  - In our case these features are provided by the ELF header
- After the file is imported, a results summary window will appear
  - Various file features will be listed in this window



# #Ghidra: Loading a Binary

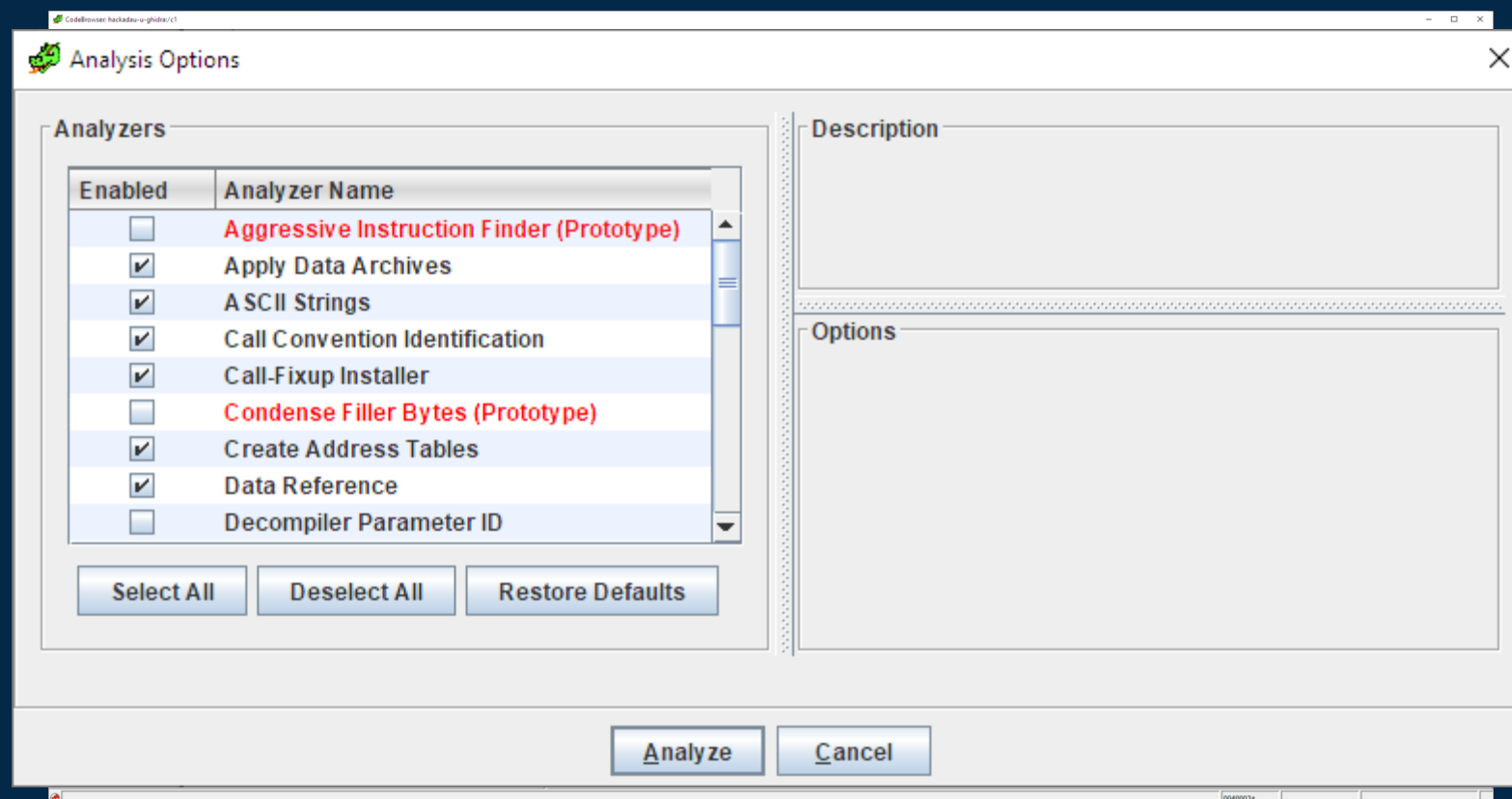


# #Ghidra: Initial Analysis

- Once a program has been loaded into the active project, it can be analyzed
  - Double click on the program in the project view to start analysis
- Ghidra will attempt to automatically analyze the binary
  - This is based on information inferred from the filetype
  - The binary entry point is determined and Ghidra begins the disassembly process
- During auto-analysis Ghidra will also attempt to:
  - Create and label functions
  - Identify cross references in memory (xrefs)



# #Ghidra: Initial Analysis



# #Ghidra: Navigation

- Once the analysis window is done, the program can be explored
  - This is done mainly within the CodeBrowser Window
- Some of the default CodeBrowser windows include:
  - Program Tree – this shows the segments of the ELF file
  - Symbol Tree – lists and displays all currently defined symbols
  - Data Type Manager – shows data types inferred during auto-analysis
  - Listing – the resulting assembly code from auto analysis
  - Console – tool output / debugging information



# #Ghidra Navigation




The screenshot shows the Ghidra Listing window for a file named 'c1'. The window displays assembly code with addresses, disassembled instructions, and comments. The code is organized into two main sections: a GNU build ID section and a GNU hash table section.

```
// .note.gnu.build-id
// SHF_NOTE [0x400274 - 0x400297]
// ram: 00400274-00400297
//
.DAT_00400274                                XREF[1]:  _elfSectionHeaders::000000d0(*)
00400274 04      ??      04h
00400275 00      ??      00h
00400276 00      ??      00h
00400277 00      ??      00h
00400278 14      ??      14h
00400279 00      ??      00h
0040027a 00      ??      00h
0040027b 00      ??      00h
0040027c 03      ??      03h
0040027d 00      ??      00h
0040027e 00      ??      00h
0040027f 00      ??      00h
00400280 47      ??      47h  G
00400281 4e      ??      4Eh  N
00400282 55      ??      55h  U
00400283 00      ??      00h
00400284 de      ??      DEh
00400285 4c      ??      4Ch  L
00400286 0c      ??      0Ch
00400287 45      ??      45h  E
00400288 ac      ??      ACh
00400289 35      ??      35h  S
0040028a cd      ??      CDh
0040028b 85      ??      85h
0040028c 1f      ??      1Fh
0040028d 4d      ??      4Dh  M
0040028e a6      ??      A6h
0040028f f6      ??      F6h
00400290 f7      ??      F7h
00400291 07      ??      07h
00400292 96      ??      96h
00400293 c3      ??      C3h
00400294 ef      ??      EFh
00400295 b8      ??      B8h
00400296 ca      ??      CAh
00400297 ff      ??      FFh
//
// .gnu.hash
// SHF_GNU_HASH [0x400298 - 0x4002b3]
// ram: 00400298-004002b3
//
__DT_GNU_HASH                                XREF[2]:  004000ea0(*),
                                                _elfSectionHeaders::00000110(*)
00400298 01 00 00 00      dddw      1h      GNU Hash Table - nbucket
0040029c 01 00 00 00      dddw      1h      GNU Hash Table - symbase
004002a0 01 00 00 00      dddw      1h      GNU Hash Table - bloom_size
```

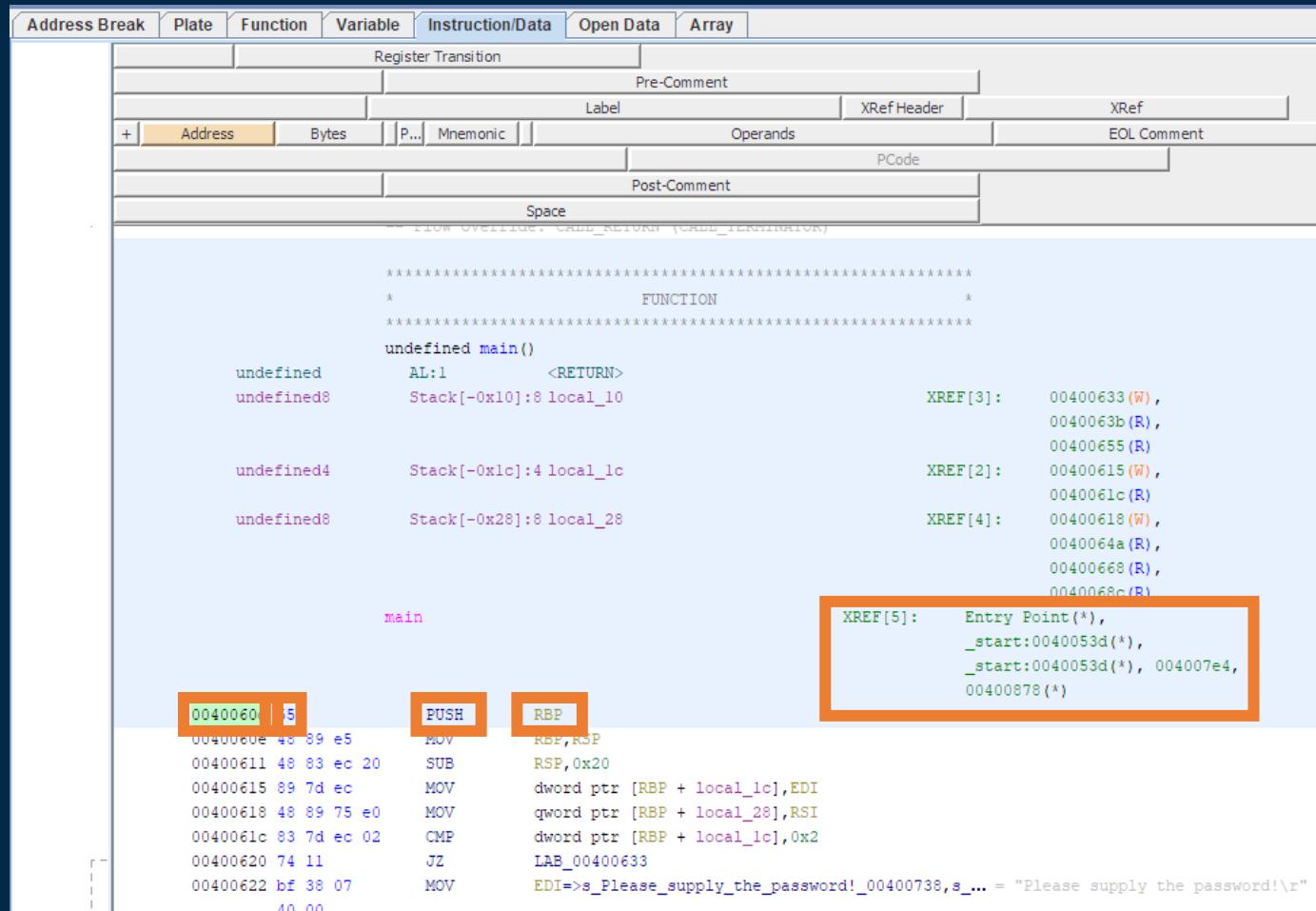


# #Ghidra Nav: Disassembly View

- This is where the resulting assembly code is displayed
- This listing can be edited by clicking the  symbol
- By default this listing contains
  - Address
  - Bytes
  - ASM Instructions (Mnemonics) and operands
  - Comments
  - Xrefs



# #Ghidra Nav: Disassembly View



The screenshot displays the Ghidra Disassembly View for a function named `main`. The interface includes tabs for Address Break, Plate, Function, Variable, Instruction/Data, Open Data, and Array. The main area shows the disassembly of the function, with instructions and their corresponding registers and stack addresses. The `XREFs` section is highlighted, showing references to other code locations.

Registers and Stack Addresses:

- undefined AL:1 <RETURN>
- undefined8 Stack[-0x10]:8 local\_10
- undefined4 Stack[-0x1c]:4 local\_1c
- undefined8 Stack[-0x28]:8 local\_28

Instructions:

- 00400600 48 89 e5 MOV RBP, RSP
- 00400611 48 83 ec 20 SUB RSP, 0x20
- 00400615 89 7d ec MOV dword ptr [RBP + local\_1c], EDI
- 00400618 48 89 75 e0 MOV qword ptr [RBP + local\_28], RSI
- 0040061c 83 7d ec 02 CMP dword ptr [RBP + local\_1c], 0x2
- 00400620 74 11 JZ LAB\_00400633
- 00400622 bf 38 07 MOV EDI => s\_Please\_supply\_the\_password!\_00400738, s\_... = "Please supply the password!\r"

XREFs:

- XREF[3]: 00400633(W), 0040063b(R), 00400655(R)
- XREF[2]: 00400615(W), 0040061c(R)
- XREF[4]: 00400618(W), 0040064a(R), 00400668(R), 0040068c(R)
- XREF[5]: Entry Point(\*), \_start:0040053d(\*), \_start:0040053d(\*), 004007e4, 00400878(\*)

XRefs: These are generated when Ghidra detects other locations or instructions that reference this address



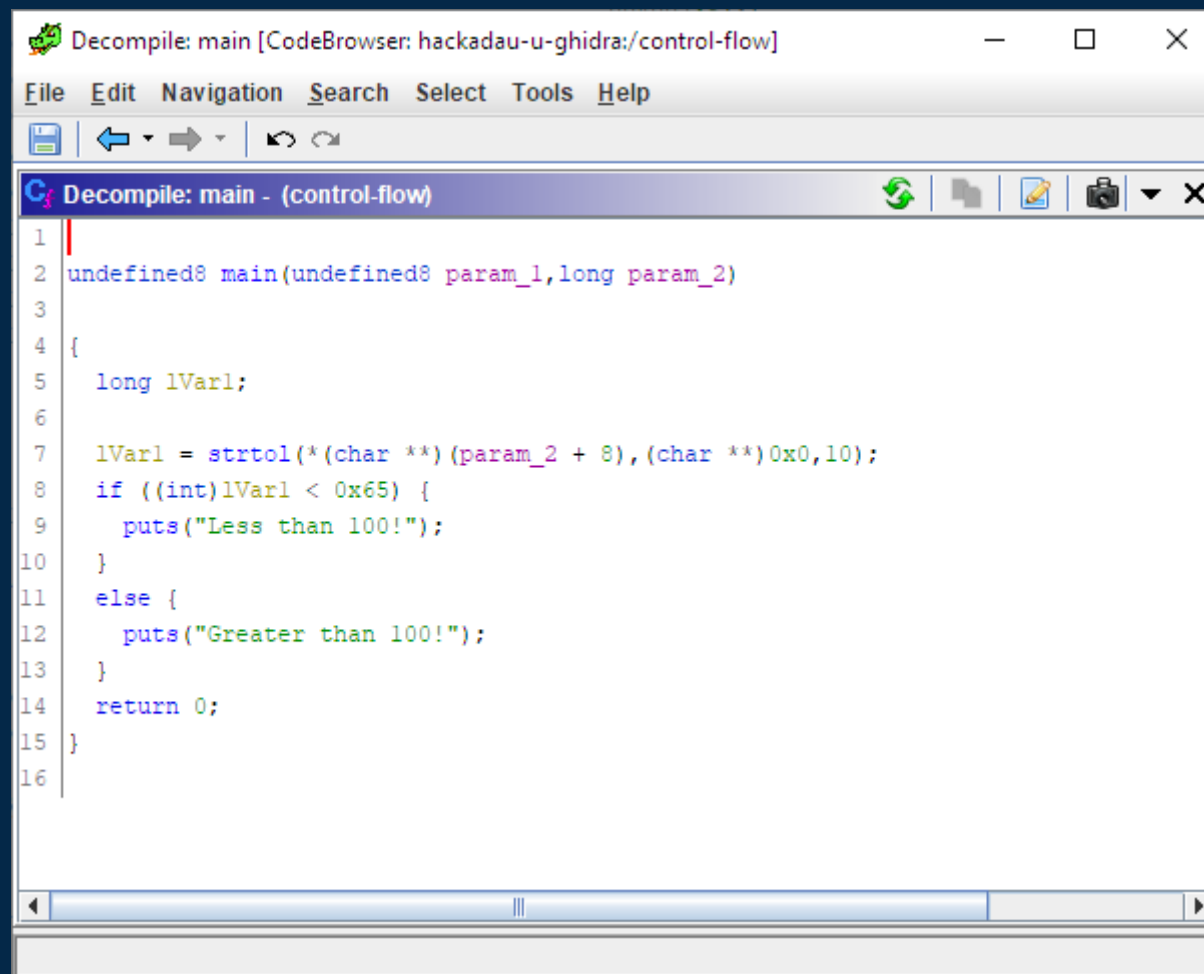


# #Ghidra: Decompiler

- One of Ghidra's most powerful features is the decompiler
  - Implemented utilizing Ghidra's P-Code
  - P-Code abstracts assembly instructions into P-Code operations
  - P-Code is an intermediate language shared across all supported processors
- The decompiler creates C code from the analyzed P-Code
  - All supported processors can utilize the decompiler
  - All processors are created with the SLEIGH language
  - SLEIGH specifies the translation from machine code to P-Code



# #Ghidra Nav: Decompiler View



The screenshot shows the Ghidra decompiler interface. The window title is "Decompile: main [CodeBrowser: hackadau-u-ghidra:/control-flow]". The menu bar includes File, Edit, Navigation, Search, Select, Tools, and Help. The toolbar has icons for file operations and navigation. The code editor displays the following C code:

```
1  
2 undefined8 main(undefined8 param_1,long param_2)  
3  
4 {  
5     long lVar1;  
6  
7     lVar1 = strtol(*(char **) (param_2 + 8), (char **)0x0,10);  
8     if ((int)lVar1 < 0x65) {  
9         puts("Less than 100!");  
10    }  
11    else {  
12        puts("Greater than 100!");  
13    }  
14    return 0;  
15 }  
16
```



# #GHIDRA: Byte View

The screenshot displays the GHIDRA interface with the 'Bytes: control-flow' window open. The assembly view on the left shows instructions from address 0040057d to 004005c5. The central pane shows the corresponding hex dump, with the instruction at 0040057d (PUSH RBP) highlighted. The right pane shows the raw byte sequence for the selected instruction.

**Assembly View:**

Address	Hex	Instruction	Comment
0040057d	55	PUSH	RBP
0040057e	48 89 e5	MOV	RBP, RSP
00400581	48 83 ec 20	SUB	RSP, 0x20
00400585	89 7d ec	MOV	dword ptr [RBP], RDI
00400588	48 89 75 e0	MOV	qword ptr [RBP], RAX
0040058c	48 8b 45 e0	MOV	RAX, qword ptr [RBP+0x10]
00400590	48 83 c0 08	ADD	RAX, 0x8
00400594	48 8b 00	MOV	RAX, qword ptr [RBP+0x14]
00400597	ba 0a 00	MOV	EDX, 0xa
0040059a	00 00		
0040059c	be 00 00	MOV	ESI, 0x0
0040059f	00 00		
004005a1	48 89 c7	MOV	RDI, RAX
004005a4	b8 00 00	MOV	EAX, 0x0
004005a7	00 00		
004005a9	e8 d2 fe ff ff	CALL	strtol
004005ac	48 98	CDQE	
004005b0	48 89 45 f8	MOV	qword ptr [RBP], RAX
004005b4	48 83 7d f8 64	CMP	qword ptr [RBP], RAX
004005b9	7e 0c	JLE	LAB_004005bb
004005bb	bf 64 06 40 00	MOV	EDI=>s_0
004005c0	e8 8b fe ff ff	CALL	puts
004005c5	eb 0a	JMP	LAB_004005c7

**Hex Dump View:**

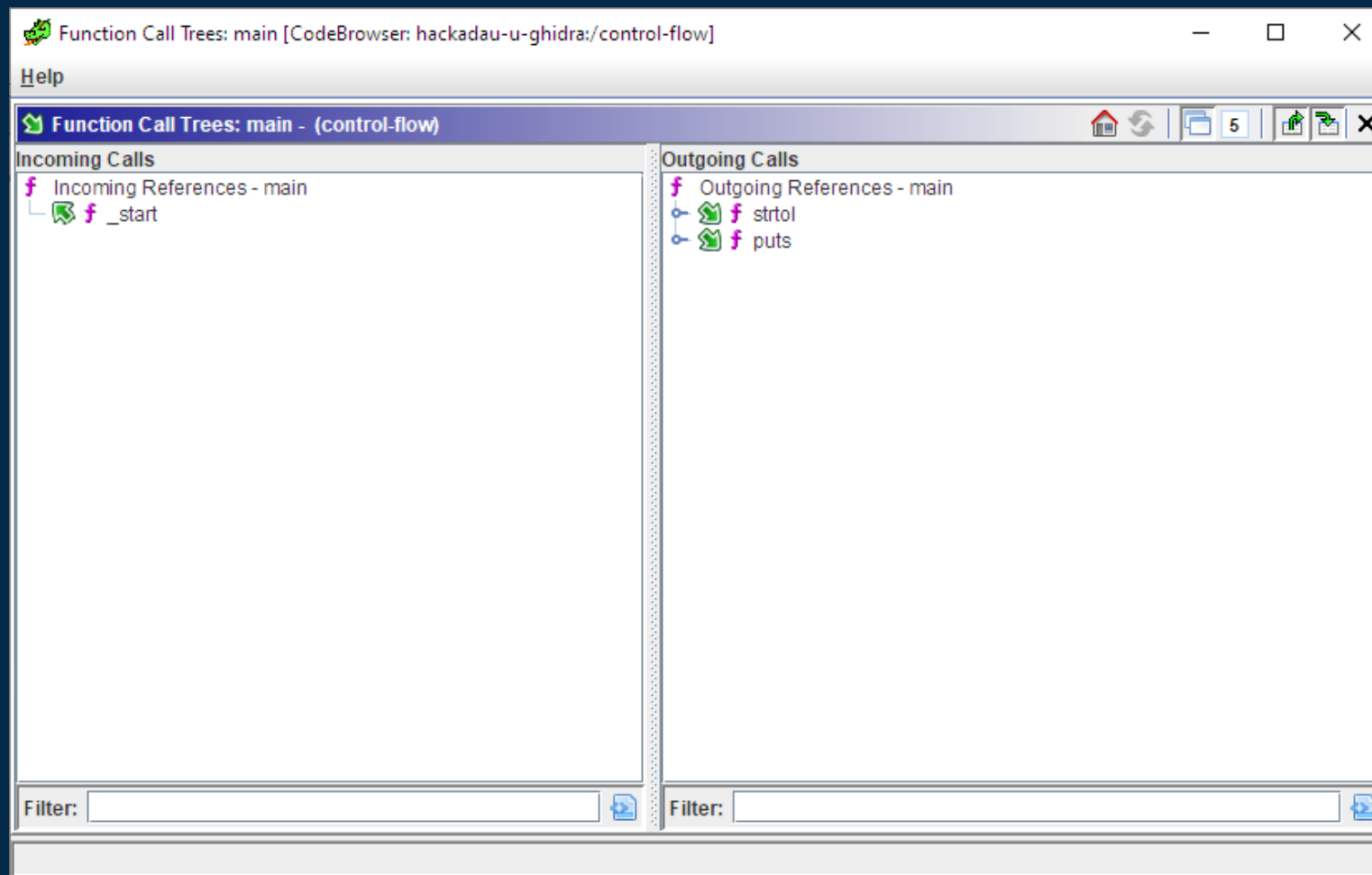
Addresses	Hex
00400470	ff 25 b2 0b 20 00 68 02 00 00 e9 c0 ff ff ff
00400480	ff 25 aa 0b 20 00 68 03 00 00 e9 b0 ff ff ff
00400490	31 ed 49 89 d1 5e 48 89 e2 48 83 e4 f0 50 54 49
004004a0	c7 c0 50 06 40 00 48 c7 c1 e0 05 40 00 48 c7 c7
004004b0	7d 05 40 00 e8 a7 ff ff ff f4 66 0f 1f 44 00 00
004004c0	b8 4f 10 60 00 55 48 2d 48 10 60 00 48 83 f8 0e
004004d0	48 89 e5 77 02 5d c3 b8 00 00 00 48 85 c0 74
004004e0	f4 5d bf 48 10 60 00 ff e0 0f 1f 80 00 00 00 00
004004f0	b8 48 10 60 00 55 48 2d 48 10 60 00 48 c1 f8 03
00400500	48 89 e5 48 89 c2 48 c1 ea 3f 48 01 d0 48 d1 f8
00400510	75 02 5d c3 ba 00 00 00 48 85 d2 74 f4 5d 48
00400520	89 c6 bf 48 10 60 00 ff e2 0f 1f 80 00 00 00 00
00400530	80 3d 11 0b 20 00 00 75 11 55 48 89 e5 e8 7e ff
00400540	ff ff 5d c6 05 fe 0a 20 00 01 f3 c3 0f 1f 40 00
00400550	48 83 3d c8 08 20 00 00 74 1e b8 00 00 00 00 48
00400560	85 c0 74 14 55 bf 20 0e 60 00 48 89 e5 ff 0d 5d
00400570	e9 7b ff ff ff 0f 1f 00 e9 73 ff ff ff 55 48 89
00400580	e5 48 83 ec 20 89 7d ec 48 89 75 e0 48 8b 45 e0
00400590	48 83 c0 08 48 8b 00 ba 0a 00 00 00 be 00 00 00
004005a0	00 48 89 c7 b8 00 00 00 00 e8 d2 fe ff ff 48 98
004005b0	48 89 45 f8 48 83 7d f8 64 7e 0c bf 64 06 40 00
004005c0	e8 8b fe ff ff eb 0a bf 76 06 40 00 e8 7f fe ff
004005d0	ff b8 00 00 00 00 c9 c3 0f 1f 84 00 00 00 00 00
004005e0	41 57 41 89 ff 41 56 49 89 f6 41 55 49 89 d5 41
004005f0	54 4c 8d 25 18 08 20 00 55 48 8d 2d 18 08 20 00
00400600	53 4c 29 e5 31 db 48 c1 fd 03 48 83 ec 08 e8 05
00400610	fe ff ff 48 85 ed 74 1e 0f 1f 84 00 00 00 00 00
00400620	4c 89 ea 4c 89 f6 44 89 ff 41 ff 14 dc 48 83 c3
00400630	01 48 39 eb 75 ea 48 83 c4 08 5b 5d 41 5c 41 5d
00400640	41 5e 41 5f c3 66 66 2e 0f 1f 84 00 00 00 00 00
00400650	f3 c3
00400660	01 00 02 00 47 72 65 61 74 65 72 20 74 68 61 6e
00400670	20 31 30 30 21 00 4c 65 73 73 20 74 68 61 6e 20
00400680	31 30 30 21 00

**Raw Bytes View:**

7b ff ff ff 0f 1f 00 e9 73 ff ff ff 55 48 89  
48 83 ec 20 89 7d ec 48 89 75 e0 48 8b 45 e0  
83 c0 08 48 8b 00 ba 0a 00 00 00 be 00 00 00  
48 89 c7 b8 00 00 00 00 e8 d2 fe ff ff 48 98  
89 45 f8 48 83 7d f8 64 7e 0c bf 64 06 40 00  
8b fe ff ff eb 0a bf 76 06 40 00 e8 7f fe ff  
b8 00 00 00 00 c9 c3 0f 1f 84 00 00 00 00 00  
57 41 89 ff 41 56 49 89 f6 41 55 49 89 d5 41  
4c 8d 25 18 08 20 00 55 48 8d 2d 18 08 20 00  
4c 29 e5 31 db 48 c1 fd 03 48 83 ec 08 e8 05  
ff ff 48 85 ed 74 1e 0f 1f 84 00 00 00 00 00  
89 ea 4c 89 f6 44 89 ff 41 ff 14 dc 48 83 c3  
48 39 eb 75 ea 48 83 c4 08 5b 5d 41 5c 41 5d  
5e 41 5f c3 66 66 2e 0f 1f 84 00 00 00 00 00  
c3  
48 83 ec 08 48 83 c4 08 c3  
00 02 00 47 72 65 61 74 65 72 20 74 68 61 6e  
31 30 30 21 00 4c 65 73 73 20 74 68 61 6e 20  
30 30 21 00  
01 1b 03 3b 34 00 00 00  
00 00 00 b8 fd ff ff 80 00 00 00 08 fe ff ff  
00 00 00 f5 fe ff ff a8 00 00 00 58 ff ff ff  
00 00 00 c8 ff ff ff 10 01 00 00  
00 00 00 00 00 00 01 7a 52 00 01 78 10 01  
0c 07 08 90 01 07 10 14 00 00 00 1c 00 00 00  
5d ff ff 2e 00 00 00 00 00 00 00 00 00 00 00



# #Ghidra: Other Views



# #GHIDRA: Navigation

- The listing view can be navigated in multiple ways
  - Scrolling
  - Arrow keys
  - Using the side scroll bar
- Double clicking on Xrefs will navigate to that location
- Locations can be specified by pressing the 'G' key



# #Ghidra Exercises: Overview

- Multiple challenge binaries have been developed for this course
- These binaries were developed to highlight Ghidra features covered in each lesson
- After each lesson, two additional challenge binaries will be released
  - For review during office hours
- On Wednesday of each session week, an advanced challenge may be released for those interested



# #Ghidra Exercises: c1

- Download the exercises from github:
  - <https://github.com/wrongbaud/hackaday-u>
  - This repository will hold all materials for the course
- Import the C1 challenge binary into Ghidra
  - What is this program doing?



# #Ghidra Exercises: c2

- Load the C2 exercise into Ghidra
- Run the application
  - How is this program different from c1?
  - What is it doing?





# #Session 1: Conclusion

- In this lesson we covered:
  - Basic x86\_86 instructions and features
  - Ghidra features
  - Ghidra navigation and basic usage
- For the next session, review the c3/c4 exercises in the github repository
  - Feel free to bring all questions to Thursday's office hour!



# #Questions

?

