Introduction to Software Reverse Engineering with Ghidra Session 1

Hackaday U

Matthew Alt

#whoami

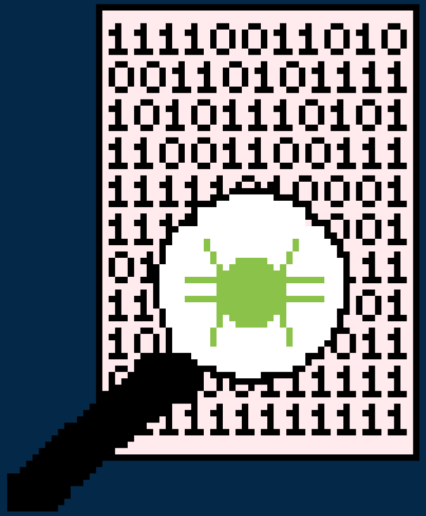
- Reverse engineer focused on embedded systems
- Security researcher for Caesar Creek Software
- Provide training and assessments through VoidStar
  - voidstarsec.com
- @wrongbaud
  - wrongbaud.github.io

#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

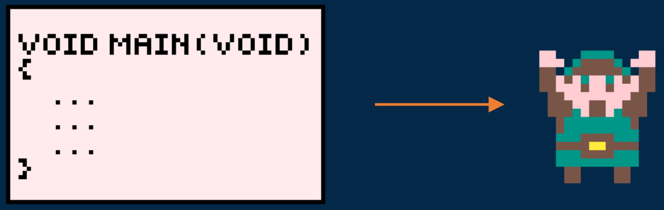


#Software Engineering Review

- 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

```c
#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 0x00000011:
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 0x0000000000400270

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
         filesz 0x0000000000000150 memsz 0x0000000000000150 flags r-x
  INTERP off    0x0000000000000190 vaddr 0x0000000000400190 paddr 0x0000000000400190 align 2**0
         filesz 0x000000000000000f memsz 0x000000000000000f flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
         filesz 0x00000000000002f8 memsz 0x00000000000002f8 flags r-x
    LOAD off    0x00000000000002f8 vaddr 0x00000000006002f8 paddr 0x00000000006002f8 align 2**21
         filesz 0x0000000000000160 memsz 0x0000000000000160 flags rw-
 DYNAMIC off    0x00000000000002f8 vaddr 0x00000000006002f8 paddr 0x00000000006002f8 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                 0x00000000004001a0
  STRTAB               0x00000000004001e8
  SYMTAB               0x00000000004001b8
  STRSZ                0x000000000000001e
  SYMENT               0x0000000000000018
  DEBUG                0x0000000000000000
  PLTGOT               0x0000000000600438
  PLTRELSZ             0x0000000000000018
  PLTREL               0x0000000000000007
  JMPREL               0x0000000000400230
  VERNEED              0x0000000000400210
  VERNEEDNUM           0x0000000000000001
  VERSYM               0x0000000000400206

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 ECX, 10
LABEL:
 INX EAX
 LOOP LABEL
...
...
```

Assemble →

```
31C0B90A
00000000
FFC1E2FC
...
...
...
```

#SE Review: Pixelated Edition

#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

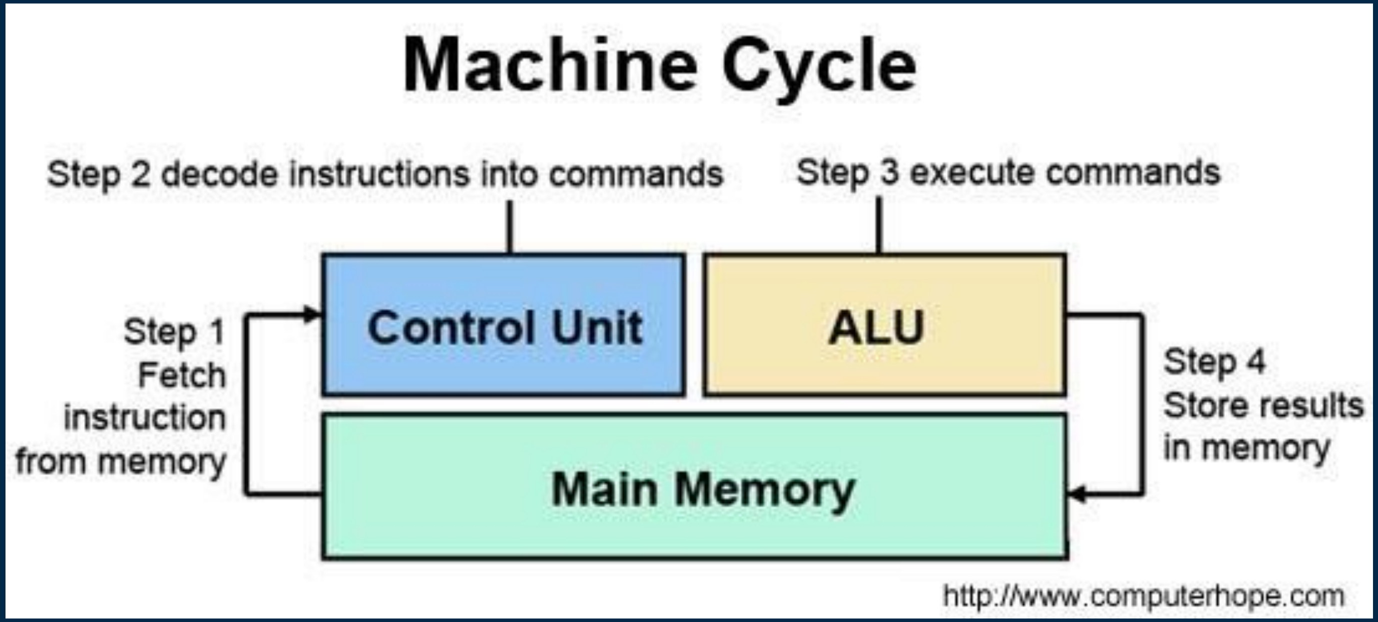
- Our goal is to work backwards from machine code
  - The ELF file will contain machine code
  - This machine code can be converted BACK into assembly language!
  - Machine code -> Assembly Language = Disassembly!

#Computer Architecture 101

- When a program is running, the following must happen:
  1. An instruction is read into memory
  2. The instruction is process 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 info 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
R1	RCX	ECX	CX	CH
R2	RDX	EDX	DX	DH
R3	RBX	EBX	BX	BH
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

- Performs the binary operation AND on the two operands, storing the result in the first
  - `and rax,rax`
    - `rax = rax & rax`

- This syntax is used for other binary operations as well:
  - `xor`
  - `or`

#x86_64: The Stack

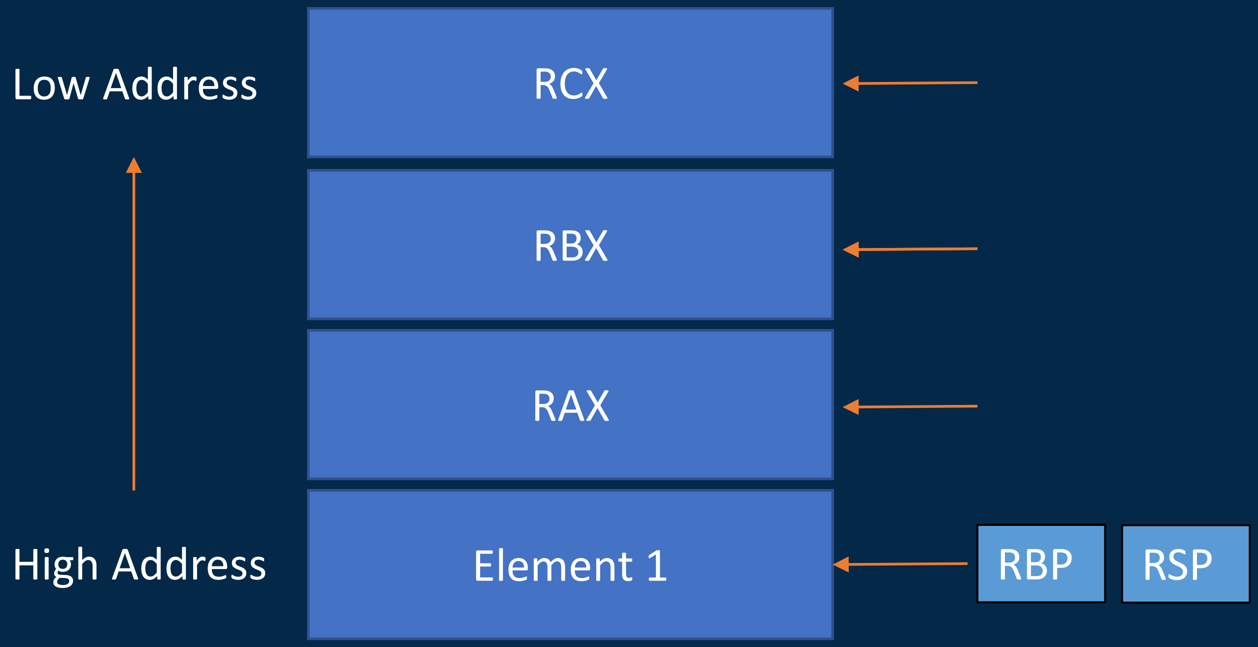
- Data structure containing elements in contiguous memory
  - POP: Reads from stack
  - PUSH: Writes to stack
- Elements are removed in the reverse order that they are added
- Grows high to low
- RSP points to top of stack
- RBP contains base pointer

#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

```
PUSH RAX
PUSH RBX
PUSH RCX
POP RAX
```



#x86_64 Instructions: jmp/call

- `jmp` is used to change what code is being executed
  - Modifies the value in EIP
  - Jmp 0x1000300
    - Set EIP to 0x1000300 and execute the instructions there
- `call` is used to implement function calls
  - Pushes value of `rbp` and `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

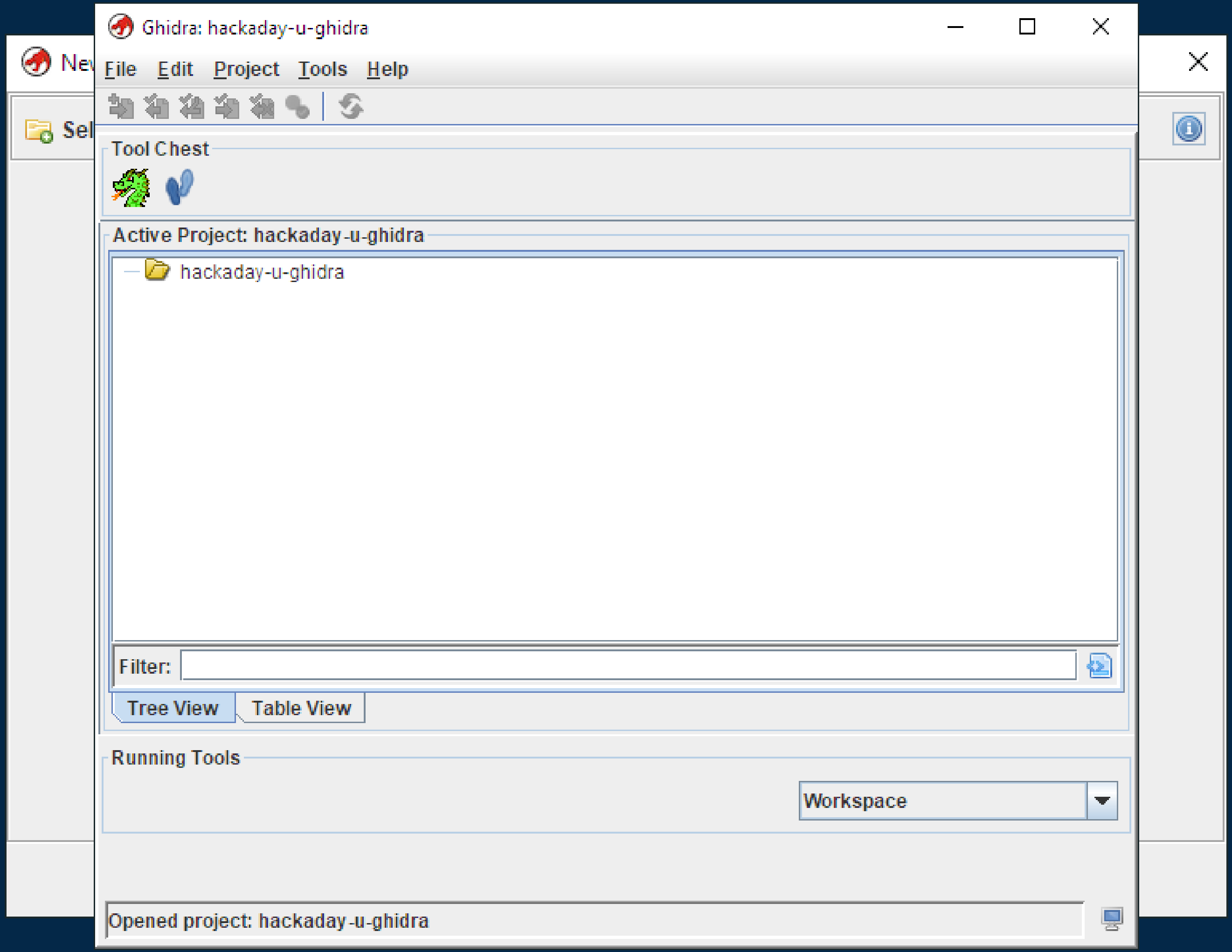
#Ghidra: Creating a Project

- 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



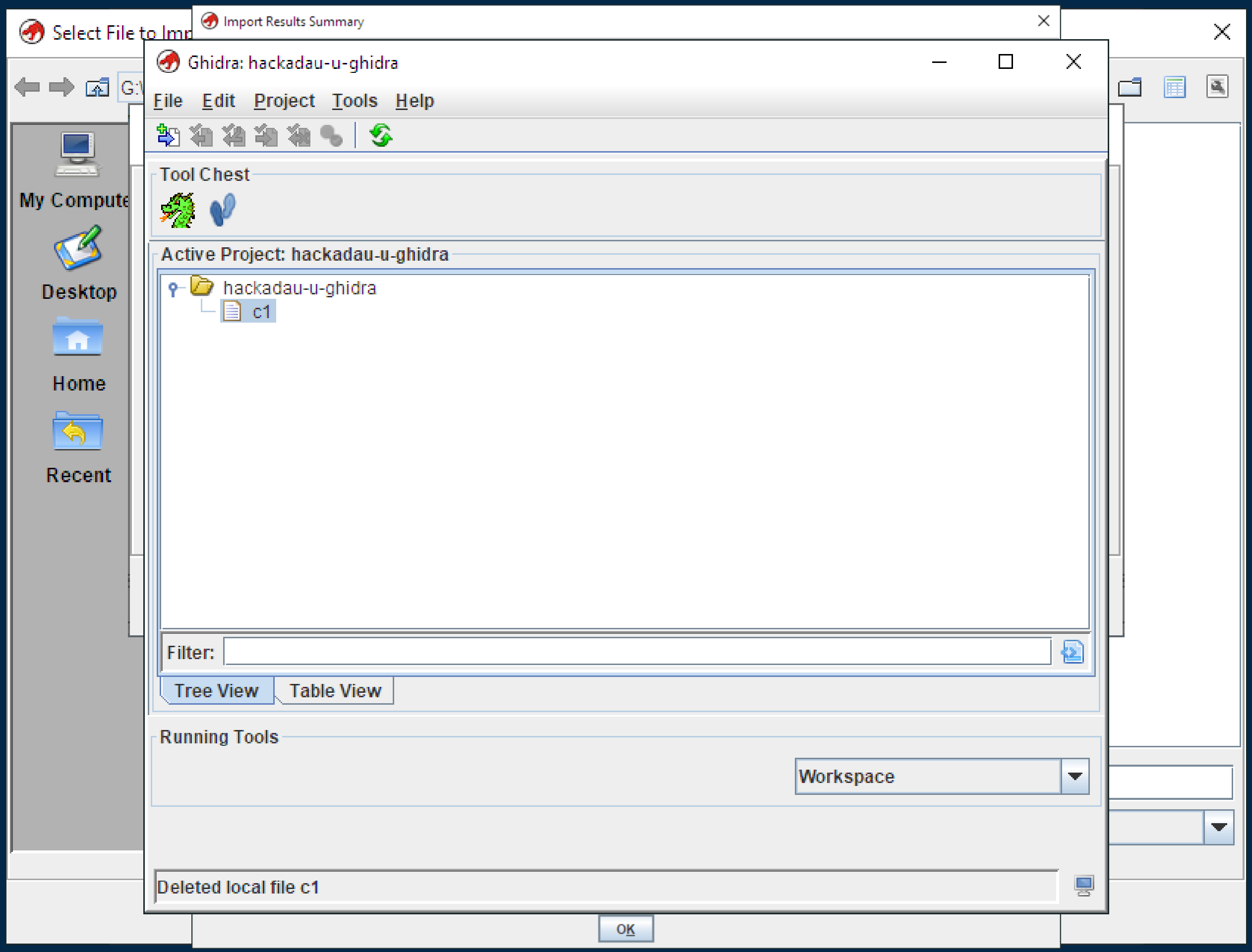
#Ghidra: Loading a Binary

- 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

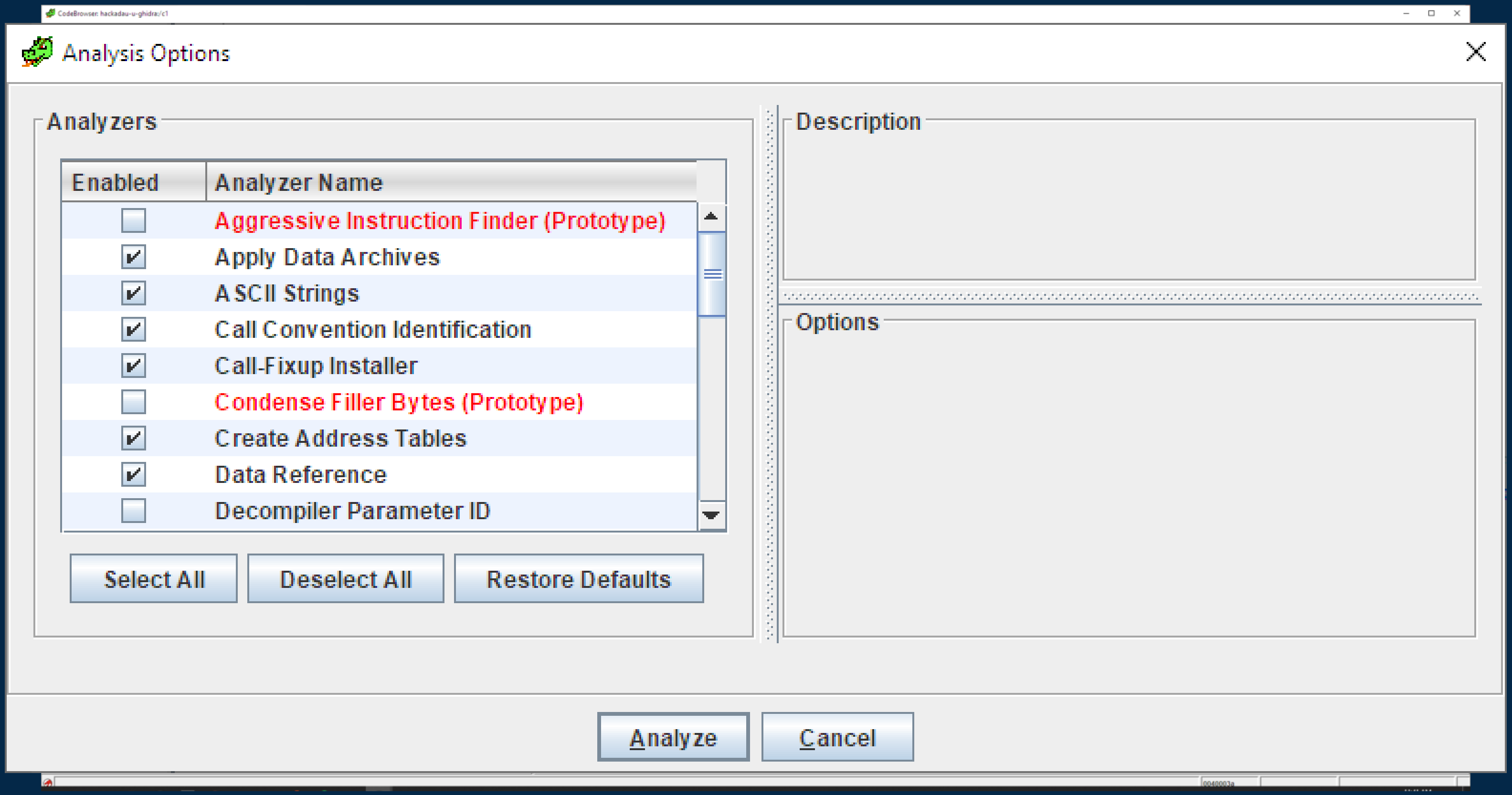
#Ghidra: Loading a Binary



#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)

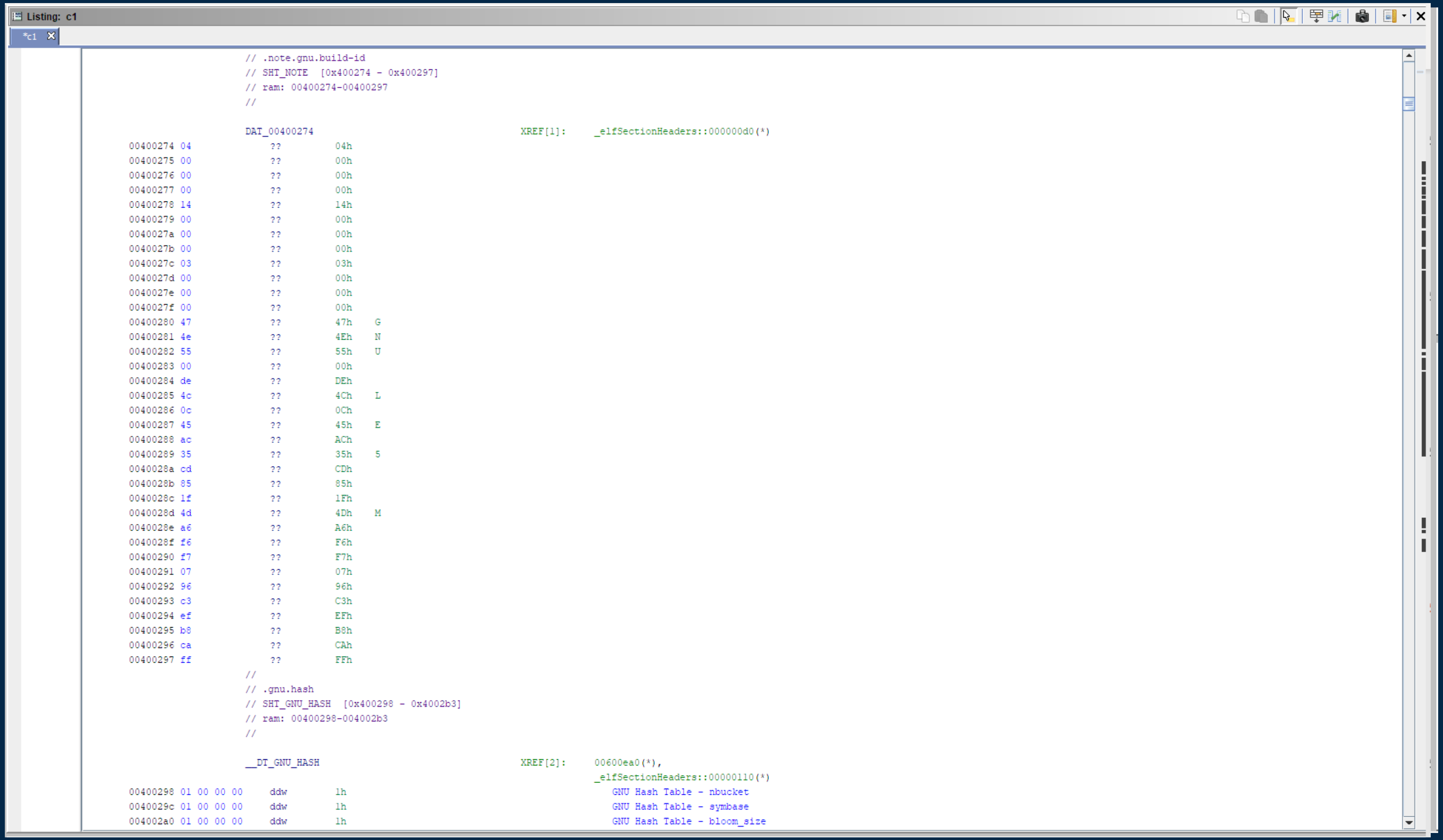
#Ghidra: Initial Analysis



#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

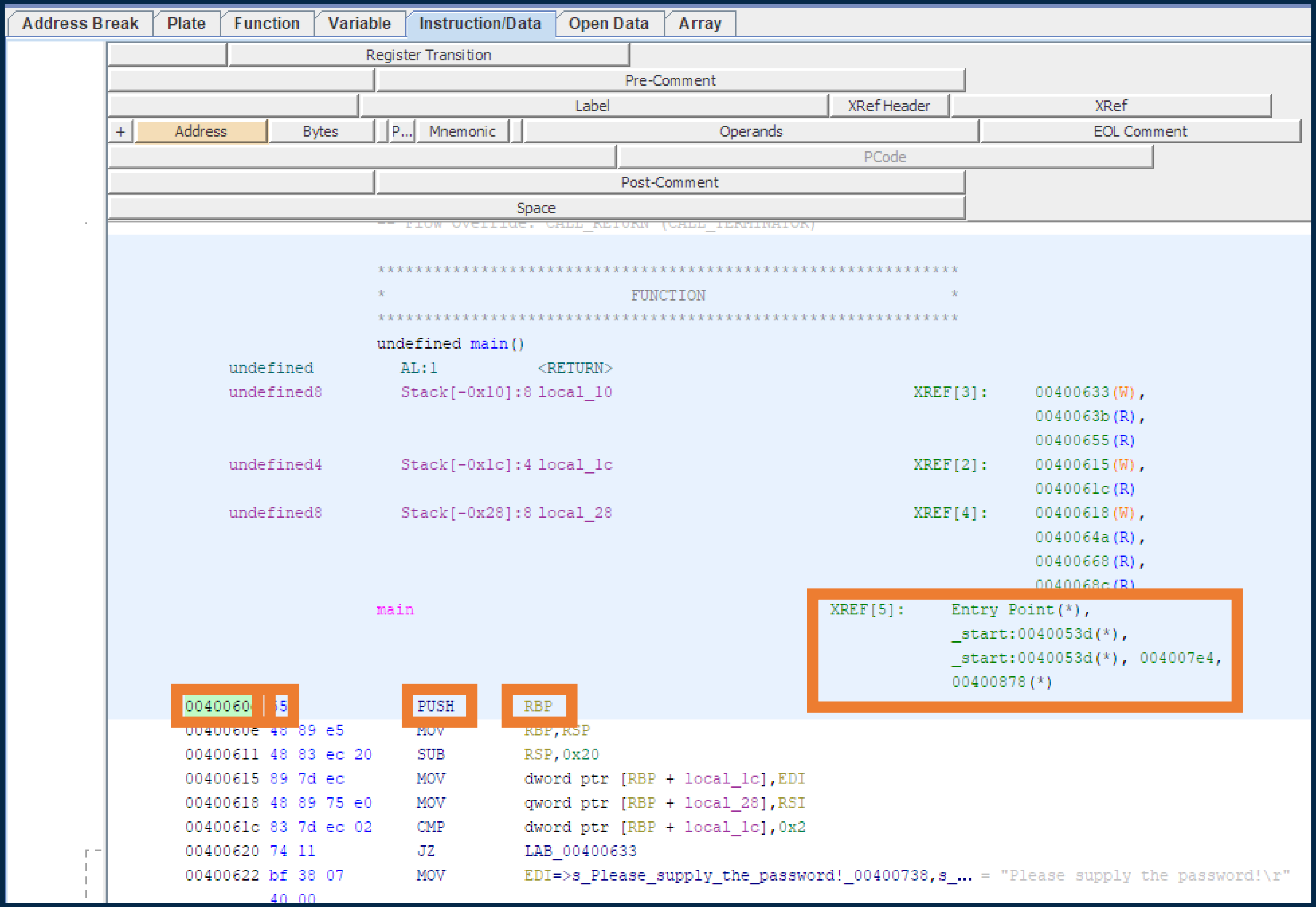
#Ghidra Navigation



#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



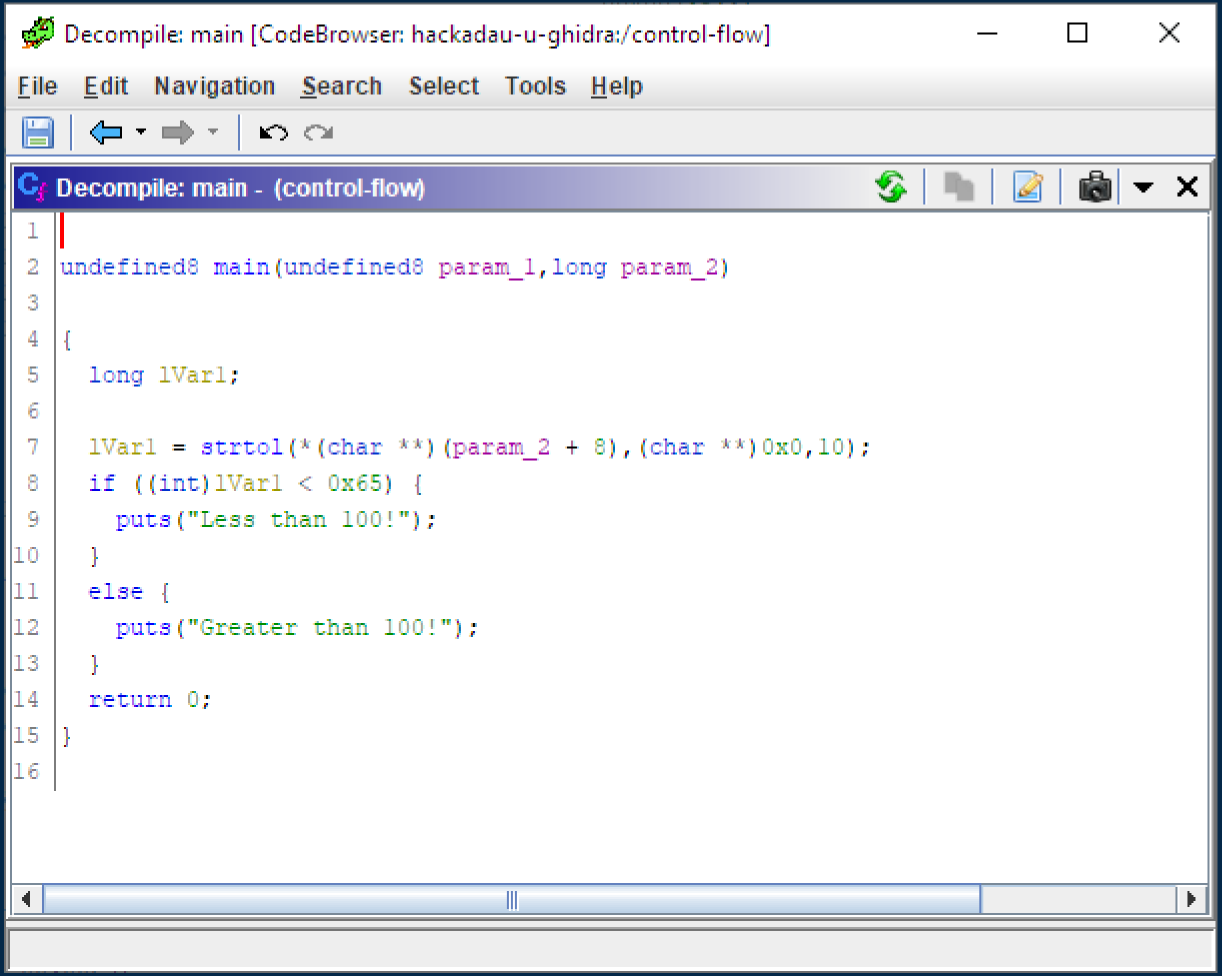
XRefs: These are generated when Ghidra detects other locations or instructions that reference this address

#Ghidra: Decompiler

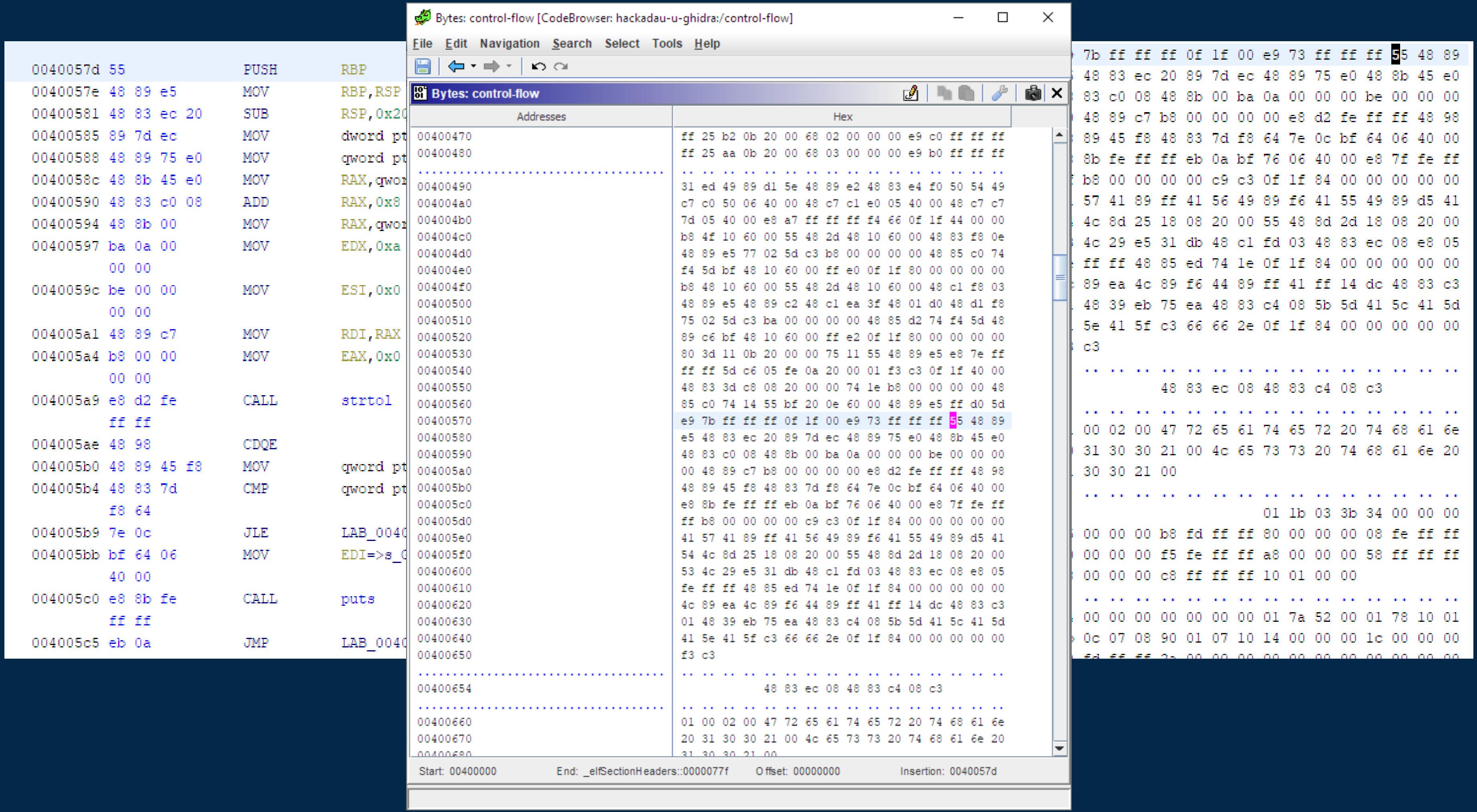
- One of Ghidra’s most powerful features is the decompiler
  - Implemented utilizing Ghidra’s P-Code
  - P-Code abstracts assembly instructions into P-Code operations
  - P-Code is an intermediate language shared across all supported processors

- 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

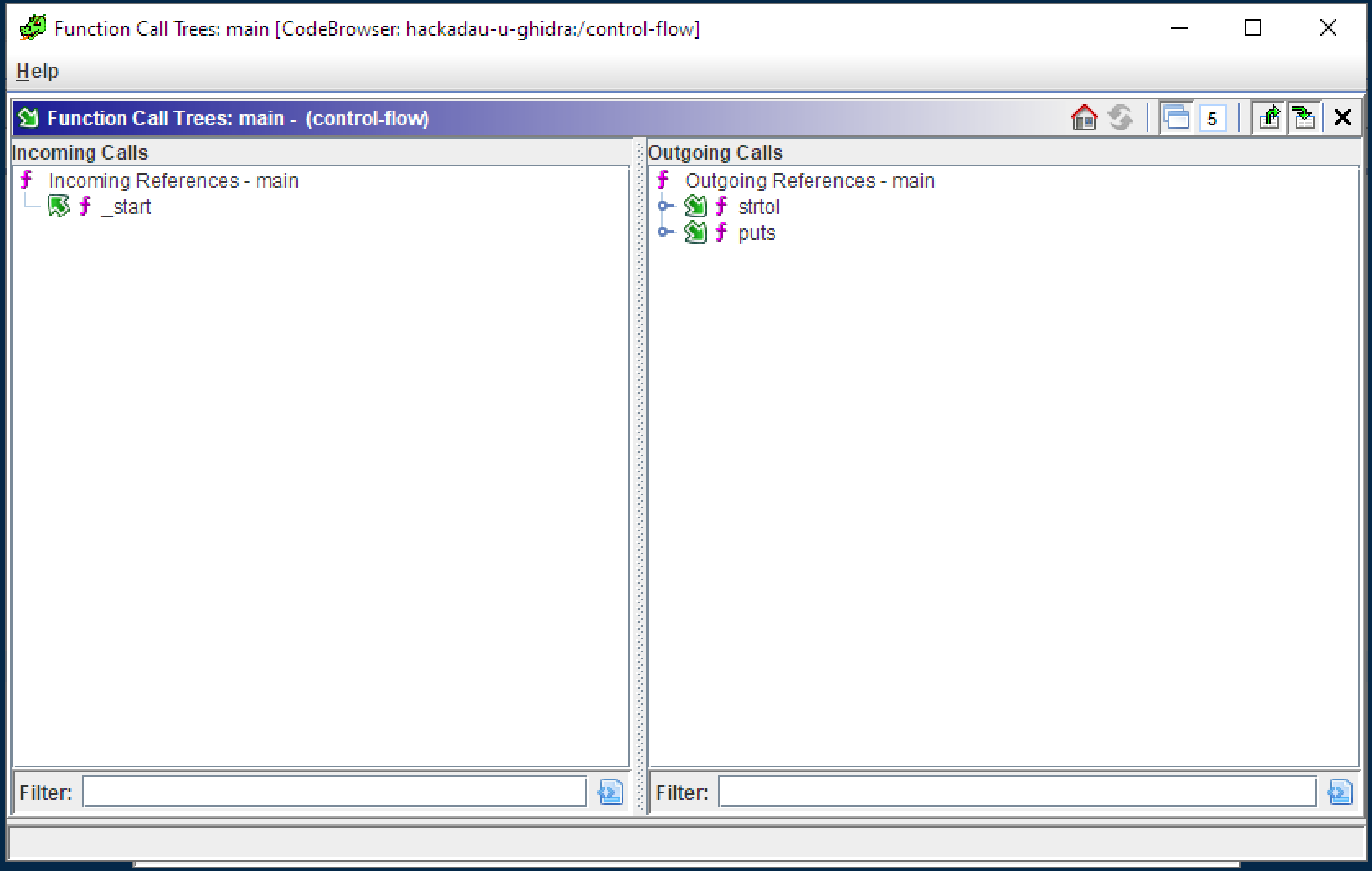
#Ghidra Nav: Decompiler View



#GHIDRA: Byte View



#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

?