

Introduction to Software Reverse Engineering with Ghidra Session 3: C to ASM II

Hackaday U
Matthew Alt



#Outline: C to ASM

- Class Admin
- SRE Tool Landscape Review
- Structures
- Enums
- Pointers
- File Operations
 - Open / Close / Read / Write
- Patching Binaries
- Ghidra Features
 - Bookmarks
 - Searching



#Course Administration

- Office hours will be Thursday at 6:00 ET
- Questions for office hours can be submitted via zoom
- Questions can also be submitted through:
 - Hackaday.io chat room
 - Hackaday messaging



#Session Goals

- Discuss SRE tool landscape
- Review Additional C Constructs / OS Concepts
 - Structures
 - Pointers
 - Enums
 - File operations
 - System calls
- Ghidra Feature Review
 - Binary patching
 - Searching
 - Bookmarks
 - Data type creation



#SRE Tool Landscape

- Many tools exist to perform SRE, below are a few:
 - Radare/r2 - disassembler
 - IDA Pro – disassembler/decompiler
 - Ghidra – disassembler/decompiler
 - Objdump - disassembler
- All of these tools have pros and cons
 - Many people use a combination of these tools



#SRE Tools: A Comparison

- IDA Pro, Pros:
 - Large amount of existing plugins and community tools
 - Handles C++ binaries well
 - PDB Parsing works well
 - In development for over a decade
- IDA Pro, Cons:
 - Costly
 - Documentation can sometimes be lacking (this is improving)
 - Can be difficult to modify core behavior



#SRE Tools: A Comparison

- Radare2, Pros:
 - Open source, regularly updated, active community
 - Forensic features (mount FS, detect partitions, etc)
 - Supports multiple architectures, file formats, Oses, etc
 - Extensible scripting interface
- Radare2, Cons:
 - Steep learning curve (initially)
 - No maintained GUI (3rd party GUIs have been developed)
 - No decompiler



#SRE Tools: Wrap up

- There are plenty of factors to consider when choosing SRE tools:
 - Usability
 - Architecture support
 - File format support
 - Extendibility
 - Price
- As you perform more SRE, experiment with as many tools as possible
 - Using other tools sometimes can help you refine how you approach SRE



#Structures

- User defined data type allowing for combining various data types
- Composite data type defining a grouped list of variables under one name in a block of memory
- Directly reference a contiguous block of memory
 - Padding bytes may be added for alignment purposes



#Structures: Padding

- Elements in structures need to be byte aligned
 - This is compiler dependent (most compilers will do this however)
- Fields that are not byte aligned with the CPU will be padded
 - This is to stop unaligned accesses from occurring
 - Used for compiler optimizations as well
- Ex: A byte sized element will occupy 4 bytes of memory but only use 1



#Structures: C to ASM

• C Code

```
typedef struct {  
    int x;  
    char y;  
    int z;  
}point;
```

```
int main(int argc, char * argv[])  
{  
    point test;  
    test.x = 100;  
    test.y = 'X';  
    test.z = 300;  
    return 0;  
}
```

• Assembly Code

```
<main>:    push    rbp  
<main+1>:  mov     rbp, rsp  
<main+4>:  mov     DWORD PTR [rbp-0x14], edi  
<main+7>:  mov     QWORD PTR [rbp-0x20], rsi  
<main+11>: mov     DWORD PTR [rbp-0x10], 0x64  
<main+18>: mov     BYTE  PTR [rbp-0xc], 0x58  
<main+22>: mov     DWORD PTR [rbp-0x8], 0x12C  
<main+29>: mov     eax, 0x0  
<main+34>: pop     rbp  
<main+35>: ret
```

Stack Addr	Value
rbp-0x8	0x12C
rbp-0xc	0x58
rbp-0x10	0x64
rbp-0x14	RSI
rbp-0x20	EDI



#Structures: Ghidra

- Structures can be added to Ghidra in multiple ways
 - Import header files
 - Manual creation in struct creator
- To import a c file do the following:
 - File -> Import C Source
- Manual struct creation can be done from the data types manager



#Ghidra Tip: Structure Creation

Remember, the struct took up 12 bytes total on the stack

Structure Editor - struct (struct-ex-2) [CodeBrowser(2): hackaday:/struct-ex-2]

Help

Offset	Length	Mnemonic	DataType	Name	Comment
0	4	int	int	X	
4	1	db	byte	Y	
8	4	int	int	Z	

Search:

Byte Offset: 11 10 9 8 7 6 5 4 3 2 1

Component Bits: Z Y X

Name: struct

Description:

Category: struct-ex-2/

Size: 12 Alignment: 4

align(minimu... pack(maxi...

☒ none ☒ none

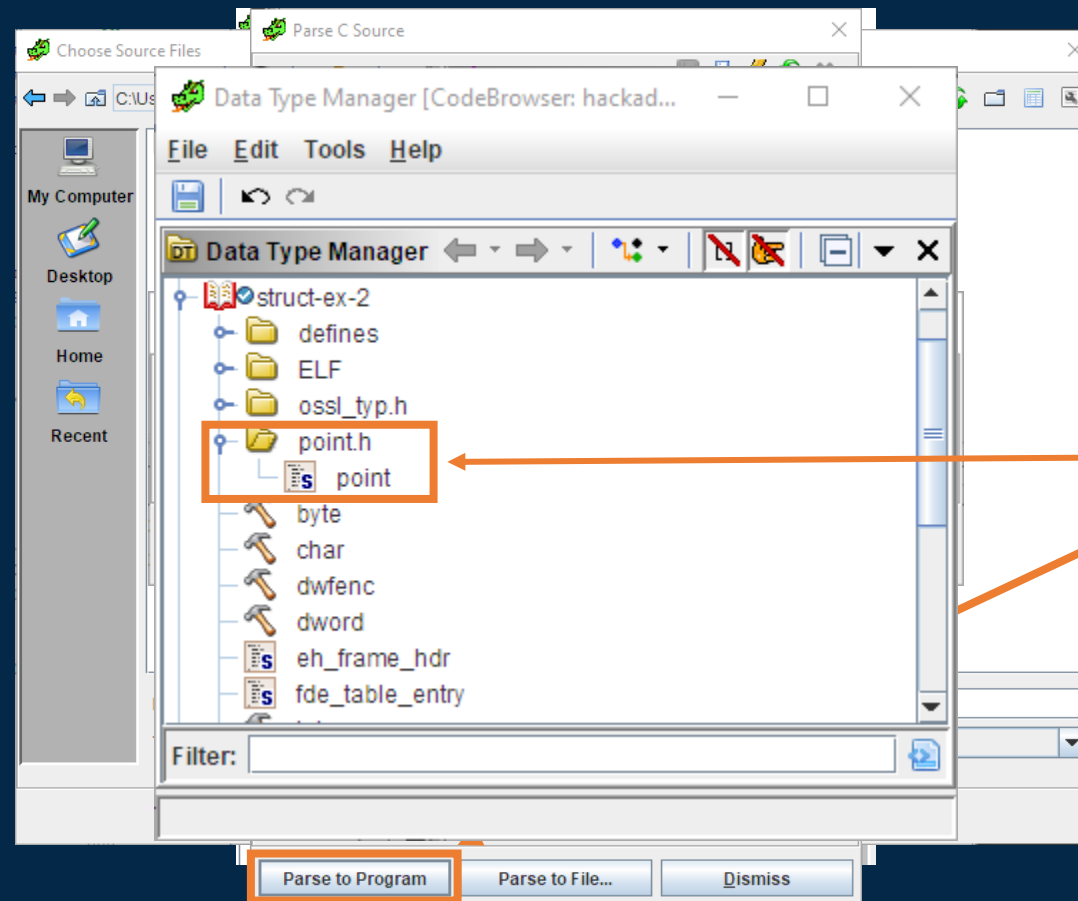
☐ machine ☐

☒ Align

Select this option to fix the alignment!



#Ghidra Tip: Structure Creation

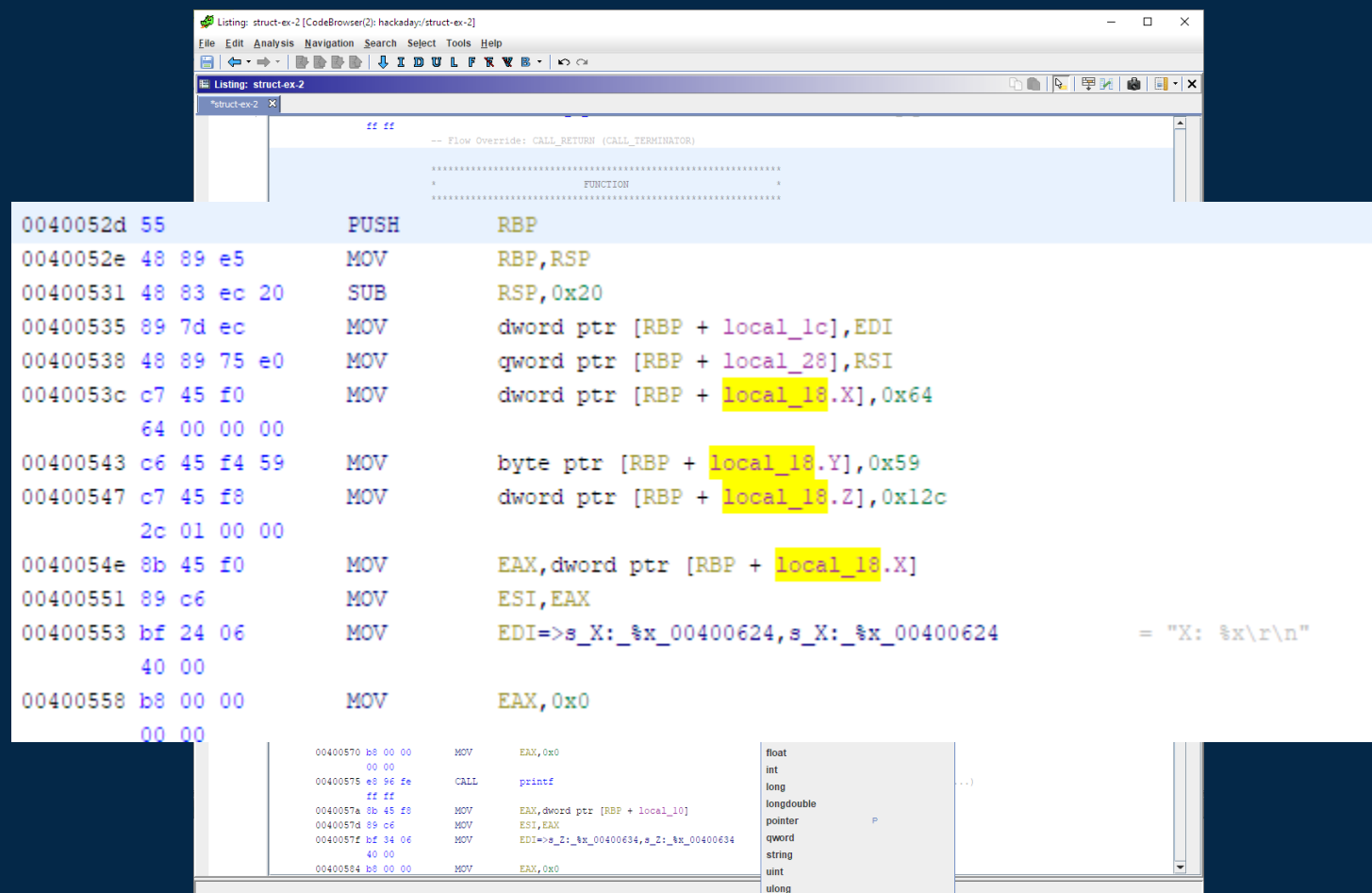


Click here to add additional header files!

The struct has now been added to the program for use in the file to the program



#Ghidra Tip: Applying Structs



```
Listing: struct-ex-2 [CodeBrowser(2): hackaday/struct-ex-2]
File Edit Analysis Navigation Search Select Tools Help
Listing: struct-ex-2
*struct-ex-2
ff ff
-- Flow Override: CALL_RETURN (CALL_TERMINATOR)
*****
* FUNCTION
*****
0040052d 55      PUSH     RBP
0040052e 48 89 e5  MOV     RBP,RSP
00400531 48 83 ec 20  SUB     RSP,0x20
00400535 89 7d ec    MOV     dword ptr [RBP + local_1c],EDI
00400538 48 89 75 e0  MOV     qword ptr [RBP + local_28],RSI
0040053c c7 45 f0    MOV     dword ptr [RBP + local_18.X],0x64
00400543 c6 45 f4 59  MOV     byte ptr [RBP + local_18.Y],0x59
00400547 c7 45 f8    MOV     dword ptr [RBP + local_18.Z],0x12c
0040054e 8b 45 f0    MOV     EAX,dword ptr [RBP + local_18.X]
00400551 89 c6      MOV     ESI,EAX
00400553 bf 24 06    MOV     EDI=>s_X: %x_00400624,s_X: %x_00400624 = "X: %x\r\n"
00400558 b8 00 00    MOV     EAX,0x0
00400559 00 00
00400570 b8 00 00    MOV     EAX,0x0
00400575 e8 96 fe    CALL    printf
0040057a 8b 45 f8    MOV     EAX,dword ptr [RBP + local_10]
0040057d 89 c6      MOV     ESI,EAX
0040057f bf 34 06    MOV     EDI=>s_Z: %x_00400634,s_Z: %x_00400634
00400584 b8 00 00    MOV     EAX,0x0
float
int
long
longdouble
pointer
qword
string
uint
ulong
```



#Structures: Exercise

- Load the exercise from `session-three/exercises/structs`
- How many members does this struct have?
- What do the various members represent?
- Can you re-create the struct in Ghidra?




```
/myProgram `python -c 'print "a"'`  
/myProgram `python -c 'print "a"'`  
/myProgram `python -c 'print "a"'`
```

#Tip: Non Ascii from CMD line

- Thus far, the challenges for this course have worked on ASCII inputs/outputs
- Python can be used to provide non-ASCII input from the command line
- Example, if you wanted to provide the hex values 0x210010, you would perform the following:
 - `./exercise `python -c 'print "\x21\x00\x10"'``



#Pointers

- A pointer is a variable whose value is the address of another variable
 - Direct address of a memory location
- Pointers provide an indirect method of accessing variables
- Pointers are always the same size, despite the size of the data that they point to
 - This will be compiler / architecture dependent



#Pointeres: C to ASM

• C Code

```
void swap(int *x, int *y) {  
    int tmp;  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}  
int main() {  
    int a = 2, b = 3;  
    swap(&a, &b);  
    return 0;  
}
```

• Assembly Code: main

```
push    rbp  
mov     rbp, rsp  
sub     rsp, 0x10  
mov     DWORD PTR [rbp-0x8], 0x2  
mov     DWORD PTR [rbp-0x4], 0x3  
lea     rdx, [rbp-0x4]  
lea     rax, [rbp-0x8]  
mov     rsi, rdx  
mov     rdi, rax  
call    4004ed <swap>  
mov     eax, 0x0  
leave  
ret
```



#Pointeres: C to ASM

• C Code

```
void swap(int *x, int *y) {  
    int tmp;  
    tmp = *x;  
    *x = *y;  
    *y = tmp;  
}  
  
int main() {  
    int a = 2, b = 3;  
    swap(&a, &b);  
    return 0;  
}
```

• Assembly Code: swap

```
push    rbp  
mov     rbp, rsp  
mov     QWORD PTR [rbp-0x18], rdi  
mov     QWORD PTR [rbp-0x20], rsi  
  
mov     rax, QWORD PTR [rbp-0x18]  
mov     eax, DWORD PTR [rax]  
mov     DWORD PTR [rbp-0x4], eax  
  
mov     rax, QWORD PTR [rbp-0x20]  
mov     edx, DWORD PTR [rax]  
mov     rax, QWORD PTR [rbp-0x18]  
mov     DWORD PTR [rax], edx  
  
mov     rax, QWORD PTR [rbp-0x20]  
mov     edx, DWORD PTR [rbp-0x4]  
mov     DWORD PTR [rax], edx  
  
pop     rbp  
ret
```



#Pointers: Ghidra

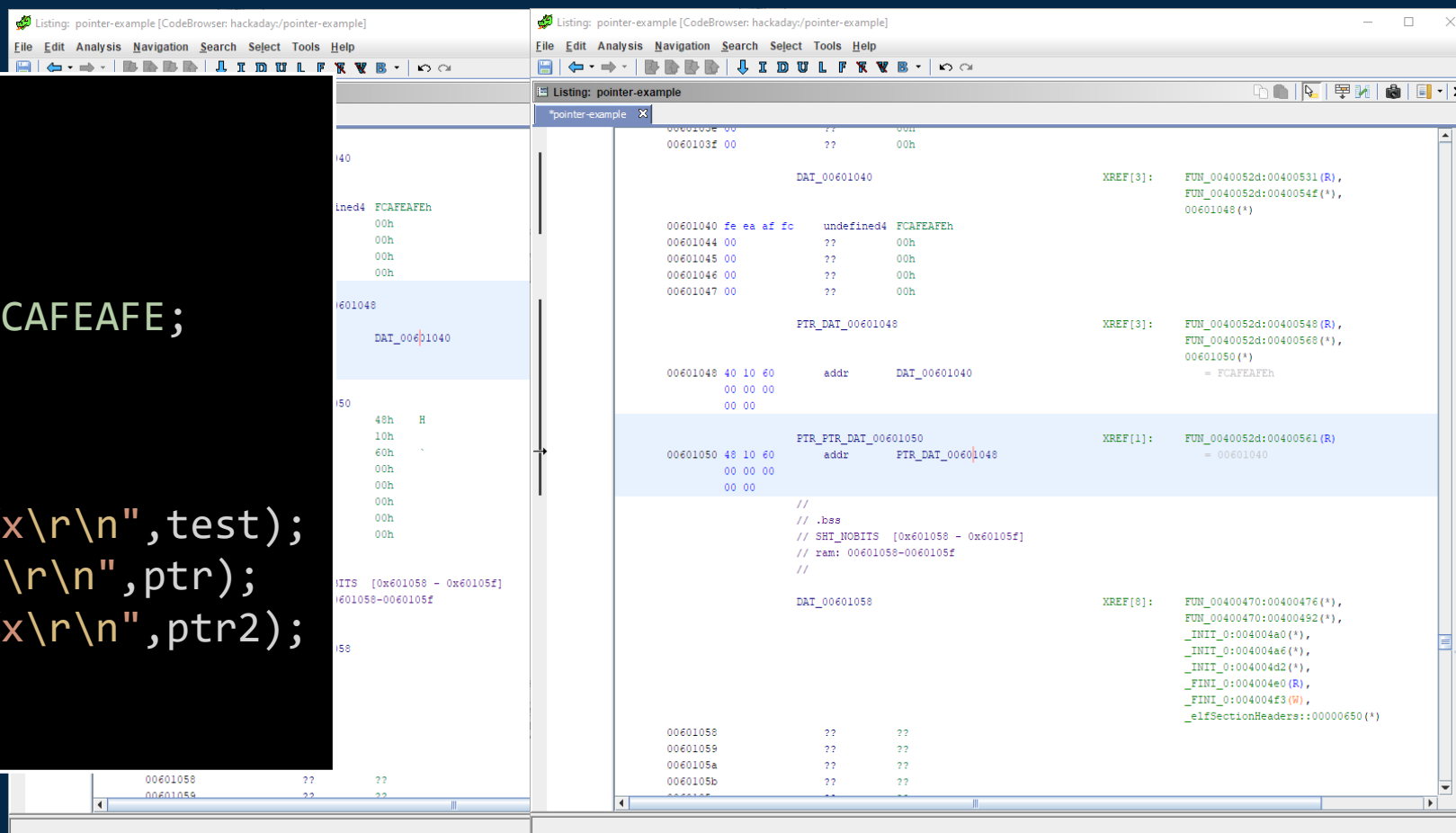
- Pointer to memory locations can be made in Ghidra using the “Pointer” datatype
- The unary operator can also be applied to any data type in the data types manager
 - For example, a pointer to the previously defined structure would be: point*



#Pointers: Ghidra

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int test = 0xDEADBEEFCAFEAFE;
int *ptr = &test;
int * ptr2 = &ptr;
int main() {
    printf("test: 0x%x\r\n",test);
    printf("ptr: 0x%x\r\n",ptr);
    printf("ptr2: 0x%x\r\n",ptr2);
}
```



#Pointers: Exercise

- Load the exercise from `session-three/exercises/pointers`
- How many members does this struct have?
 - HINT: It will look familiar to the previous struct
- What do the various members represent?
- How are pointers being used here?



#Enums

- Enums are a user defined data type in C
- Enums assign names to integral constructs
 - Ex; `enum test { pass = 1, fail = 0};`
- Often utilized to increase readability of code
- If values are not explicitly assigned to enum names, the compiler starts assignments at 0

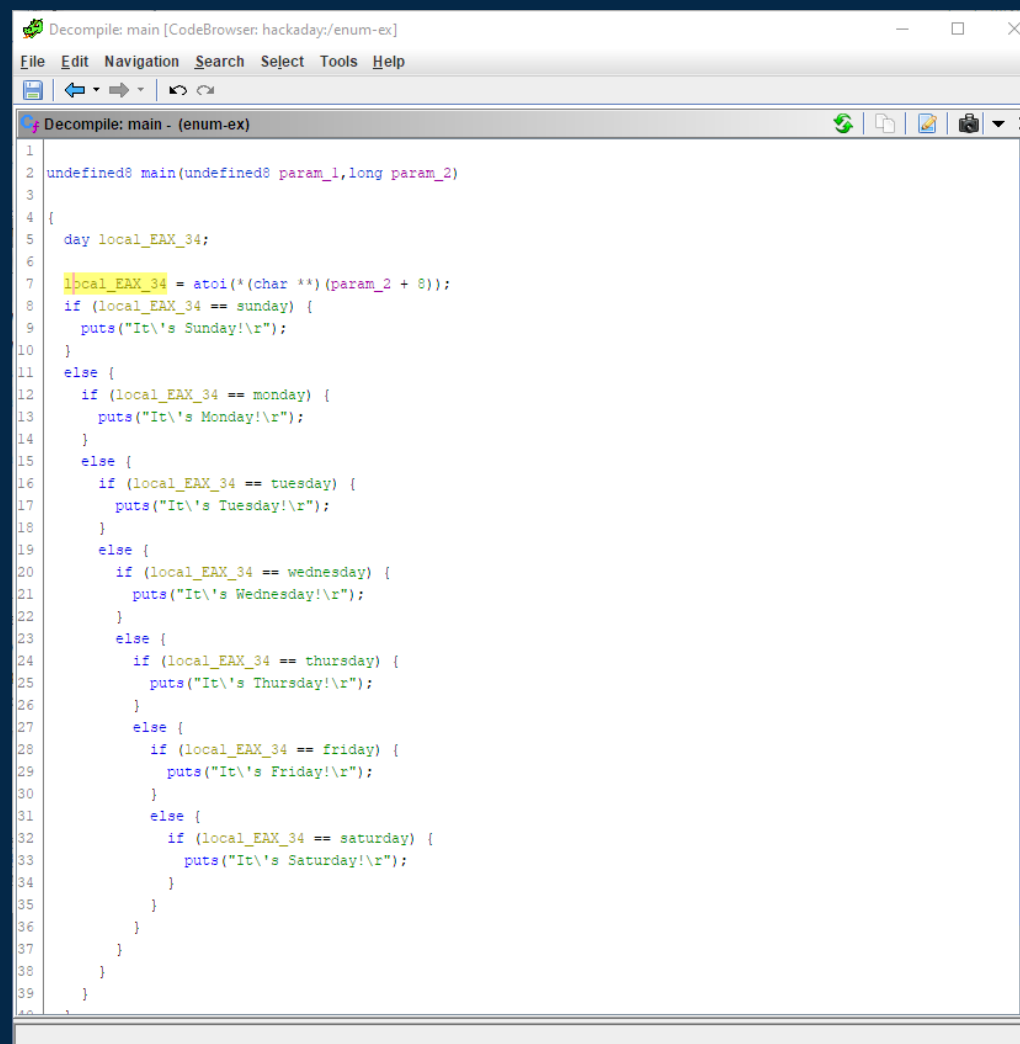


#Enums: Ghidra

- Similar to structures, enums can be imported through C files or defined manually
- These imported/created enums can be applied to scalars in the disassembly listing
- Variables in the decompiler view can also be re-typed to enums that you have defined

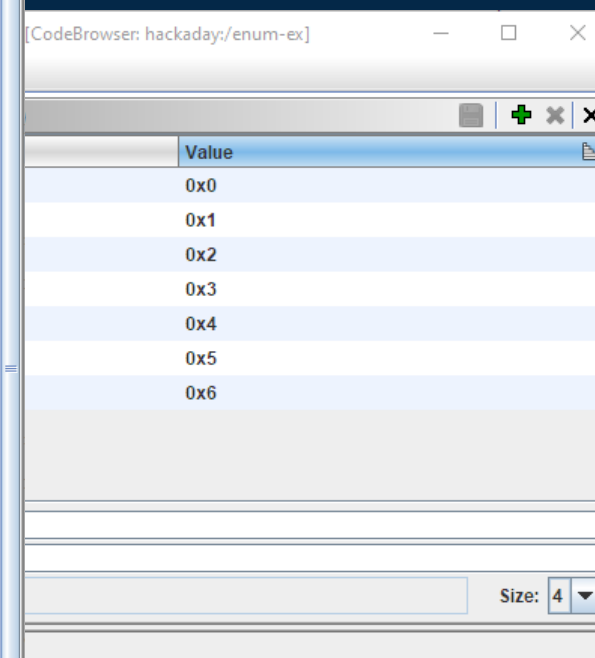


#Enums : Ghidra



The image shows a Ghidra decompiler window titled "Decompile: main [CodeBrowser: hackaday:/enum-ex]". The code is a C function named `main` that takes two parameters, `param_1` and `param_2`. It declares a local variable `local_EAX_34` of type `day`. The code then assigns `local_EAX_34` the value of `atoi((char *) (param_2 + 8))`. Following this, it enters a series of nested `if` and `else` statements to check the value of `local_EAX_34` against constants representing days of the week: `sunday`, `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, and `saturday`. For each day, it calls `puts` to print a string indicating the day. The code ends with a closing brace for the `main` function.

```
1
2 undefined8 main(undefined8 param_1,long param_2)
3
4 {
5     day local_EAX_34;
6
7     local_EAX_34 = atoi((char *) (param_2 + 8));
8     if (local_EAX_34 == sunday) {
9         puts("It's Sunday!\r");
10    }
11    else {
12        if (local_EAX_34 == monday) {
13            puts("It's Monday!\r");
14        }
15        else {
16            if (local_EAX_34 == tuesday) {
17                puts("It's Tuesday!\r");
18            }
19            else {
20                if (local_EAX_34 == wednesday) {
21                    puts("It's Wednesday!\r");
22                }
23                else {
24                    if (local_EAX_34 == thursday) {
25                        puts("It's Thursday!\r");
26                    }
27                    else {
28                        if (local_EAX_34 == friday) {
29                            puts("It's Friday!\r");
30                        }
31                        else {
32                            if (local_EAX_34 == saturday) {
33                                puts("It's Saturday!\r");
34                            }
35                        }
36                    }
37                }
38            }
39        }
40    }
```



The image shows a "Value" window from the Ghidra interface. It contains a table with a single column labeled "Value". The table lists seven hexadecimal values: `0x0`, `0x1`, `0x2`, `0x3`, `0x4`, `0x5`, and `0x6`. The window has a title bar that says "[CodeBrowser: hackaday:/enum-ex]" and a "Size" dropdown menu at the bottom right set to "4".

Value
0x0
0x1
0x2
0x3
0x4
0x5
0x6



#x86_64: System Calls

- System calls are used to interface with the operating system
 - Used for operations which require privileged access
 - This is done through the `syscall` instruction
- The operating system implements a system call handler to catch when this instruction occurs
- Information about the requested syscall is passed through registers



#x86_64: System Calls

Register	Purpose
RAX	System Call Number
RCX	Return Address
R11	Saved RFLAGS
RDI	Argument 0
RSI	Argument 1
RDX	Argument 2
R10	Argument 3
R8	Argument 4
R9	Argument 5



#System Calls: ASM

```
section .rodata  
msg: db 'hackaday-u',10;
```

```
section .text  
global _start  
_start:
```

```
mov rdi, 1  
mov rsi, msg  
mov rdx, 10  
mov rax, 1  
syscall
```

```
mov rdi, 0  
mov rax, 60  
syscall
```

Register	Purpose	Val
RAX	System Call Number	60
RDI	Argument 0	0
RSI	Argument 1	NA
RDX	Argument 2	NA



System Calls: Exercises

- Load the exercise from `session-three/exercises/syscall-exercise`
- How many system calls are performed?
 - What syscalls are performed
- What does this program do?
 - Can you figure it out without running it?
- What is the entry point of this program?



#File Operations: C to ASM

- While system calls are used at a low level to perform file operations, you will often see higher level implementations
 - libc provides these for you
 - open, write, close, etc
- For the purposes of our exercises, these will just look like regular function calls!
 - Libc utilizes the system call functionality to provide these to you!



#File Operations: C to ASM

• C Code

```
int main(int argc, char * argv[])
{
    int fd;
    char * output = "write to file";
    fd = open("testfile.txt", O_CREAT | O_WRONLY);
    int written = write(fd, output, 13);
    int x = close(fd);
    return 0;
}
```

• Assembly Code

```
mov     DWORD PTR [rbp-0x14],eax
push    rbp
mov     DWORD PTR [rbp-0x10],eax
mov     rbp, rbp
mov     eax, DWORD PTR [rbp-0x14]
mov     edx, eax
call    400480 <open@plt>
mov     OWORD PTR [rbp-0x30],rsi
mov     edi, eax
mov     OWORD PTR [rbp-0x8],0x4006a4
call    400480 <write@plt>
```

Contents of section .data:

4006a0 01000200 77726974 6520746f 2066696cwrite to fil

4006b0 65007465 78746669 60652e72 74000000 e.testfile.txt.



#File Operations: Exercises

- Load the exercise from session-three/exercises/files
 - This exercise builds off the struct / pointer exercise
- What is different about this exercise?
- What file operations are being performed?
- Can you solve for the password?



#Ghidra Tips

- During this next section, we will review various Ghidra features that may assist you with the challenges
 - Patching
 - Bookmarks
 - Searching
 - Checksumming
 - Adding additional Libraries

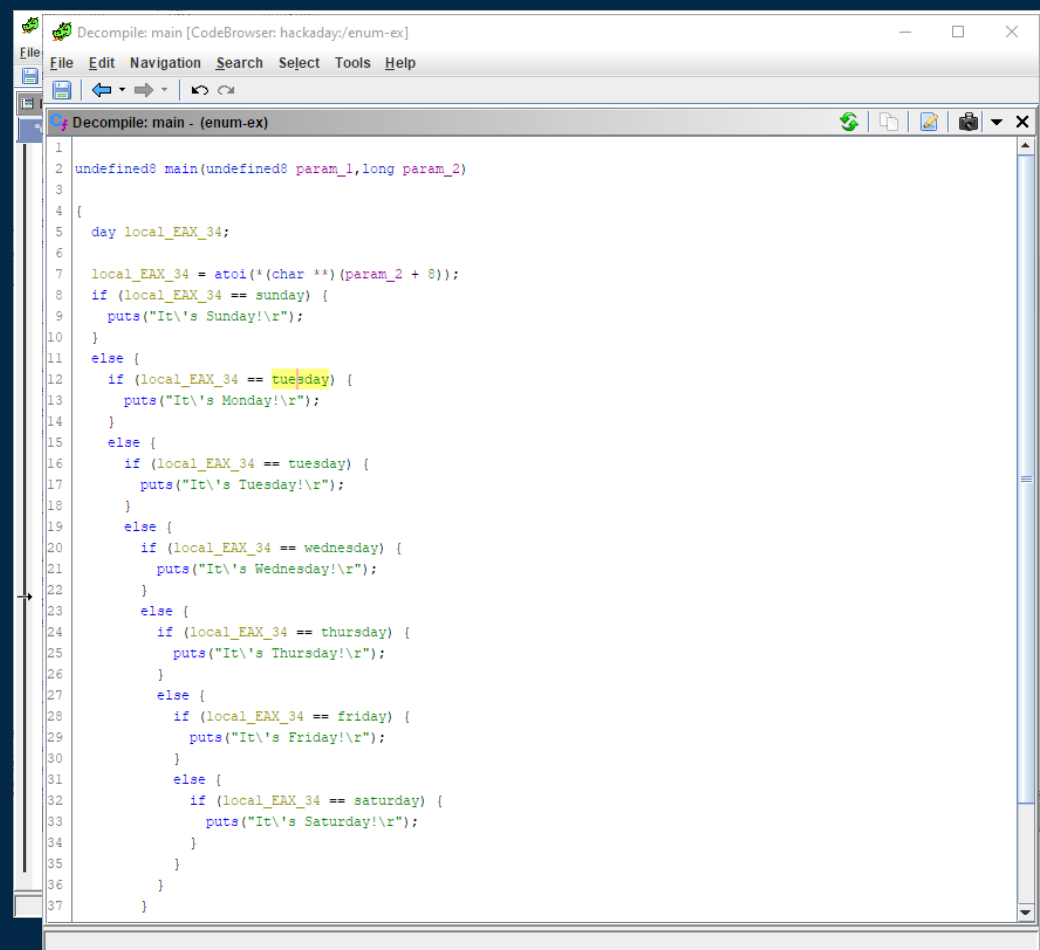


#Ghidra Tip: Binary Patching

- Ghidra can be used to patch binary files
 - Includes an assembler to allow use of actual instructions
 - Not just binary patching with hex values!
- The export functionality is used to dump all initialized memory blocks within Ghidra
 - The result is not a valid ELF file!
- Note that this will not work on files such as ELF/EXE currently
 - This feature CAN be utilized on flat firmware images however



#Ghidra Tip: Binary Patching



The screenshot shows the Ghidra decompiler interface. The title bar reads "Decompile: main [CodeBrowser: hackaday:/enum-ex]". The menu bar includes "File", "Edit", "Navigation", "Search", "Select", "Tools", and "Help". The toolbar contains icons for file operations and navigation. The main window displays the decompiled C code for a function named "main". The code uses a local variable "local_EAX_34" to store the result of "atoi(param_2 + 8)". It then uses a series of if-else statements to check the value of "local_EAX_34" against constants representing days of the week (sunday, tuesday, wednesday, thursday, friday, saturday) and prints a message for each case. The code is as follows:

```
1
2 undefined0 main(undefined0 param_1,long param_2)
3
4 {
5     day local_EAX_34;
6
7     local_EAX_34 = atoi((char *) (param_2 + 8));
8     if (local_EAX_34 == sunday) {
9         puts("It's Sunday!\r");
10    }
11    else {
12        if (local_EAX_34 == tuesday) {
13            puts("It's Monday!\r");
14        }
15        else {
16            if (local_EAX_34 == tuesday) {
17                puts("It's Tuesday!\r");
18            }
19            else {
20                if (local_EAX_34 == wednesday) {
21                    puts("It's Wednesday!\r");
22                }
23                else {
24                    if (local_EAX_34 == thursday) {
25                        puts("It's Thursday!\r");
26                    }
27                    else {
28                        if (local_EAX_34 == friday) {
29                            puts("It's Friday!\r");
30                        }
31                        else {
32                            if (local_EAX_34 == saturday) {
33                                puts("It's Saturday!\r");
34                            }
35                        }
36                    }
37                }
38            }
39        }
40    }
```

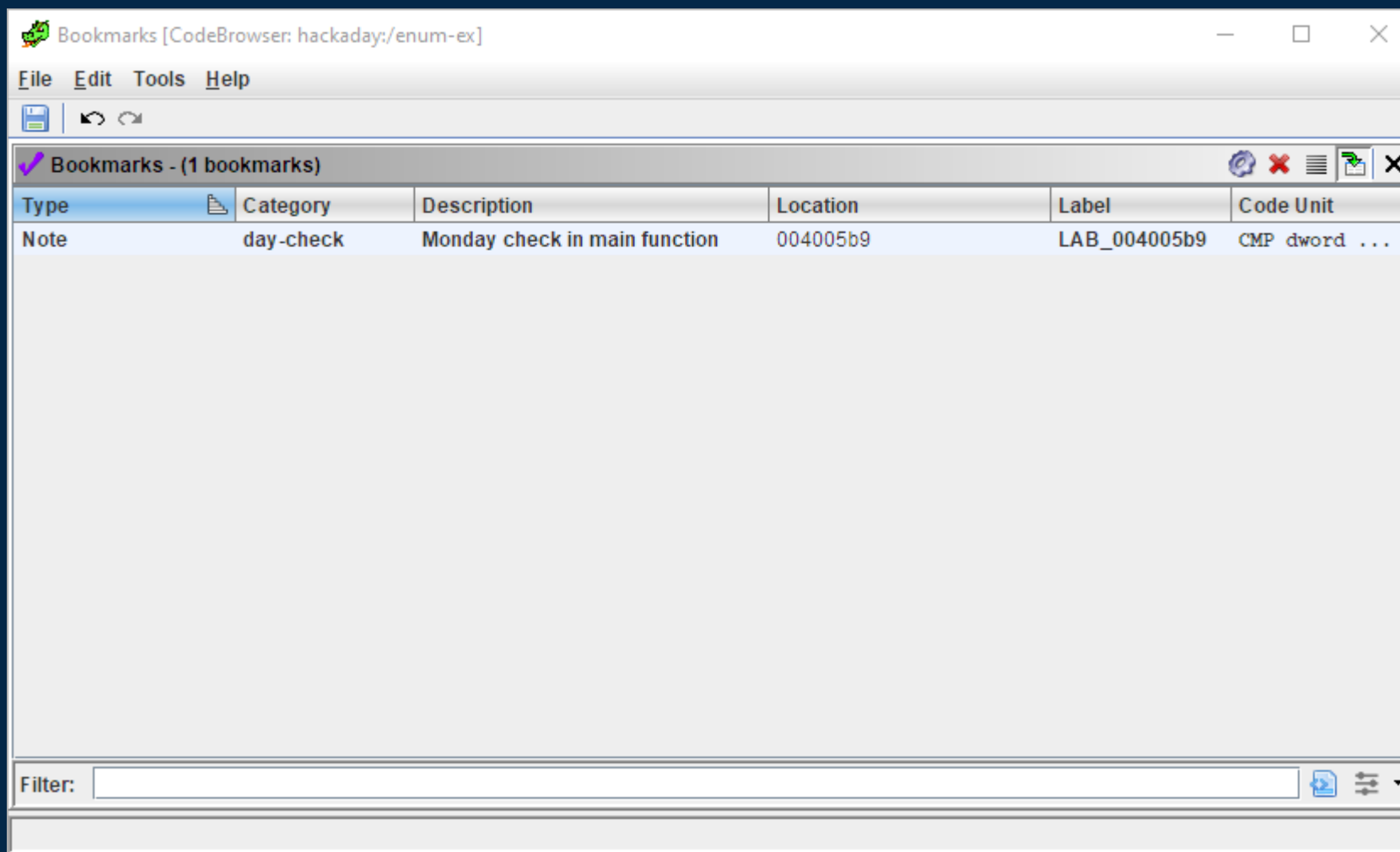


#Ghidra Tip: Bookmarks

- When using Ghidra, locations of interest can be bookmarked
- Bookmarks can be viewed and searched within the bookmarks window
 - Window -> Bookmarks
- Bookmarks are tied to addresses, and can be categorized with custom fields



#Ghidra Tip: Bookmarks



#Ghidra Tip: Searching

- When analyzing a program in Ghidra, you can search for various data types/patterns
 - Instructions
 - Scalars (immediate values)
 - Direct references
 - Instruction patterns
 - Strings
- Both program text and memory can be searched
 - Program Text = Program database or Listing display
 - Memory = entire address space

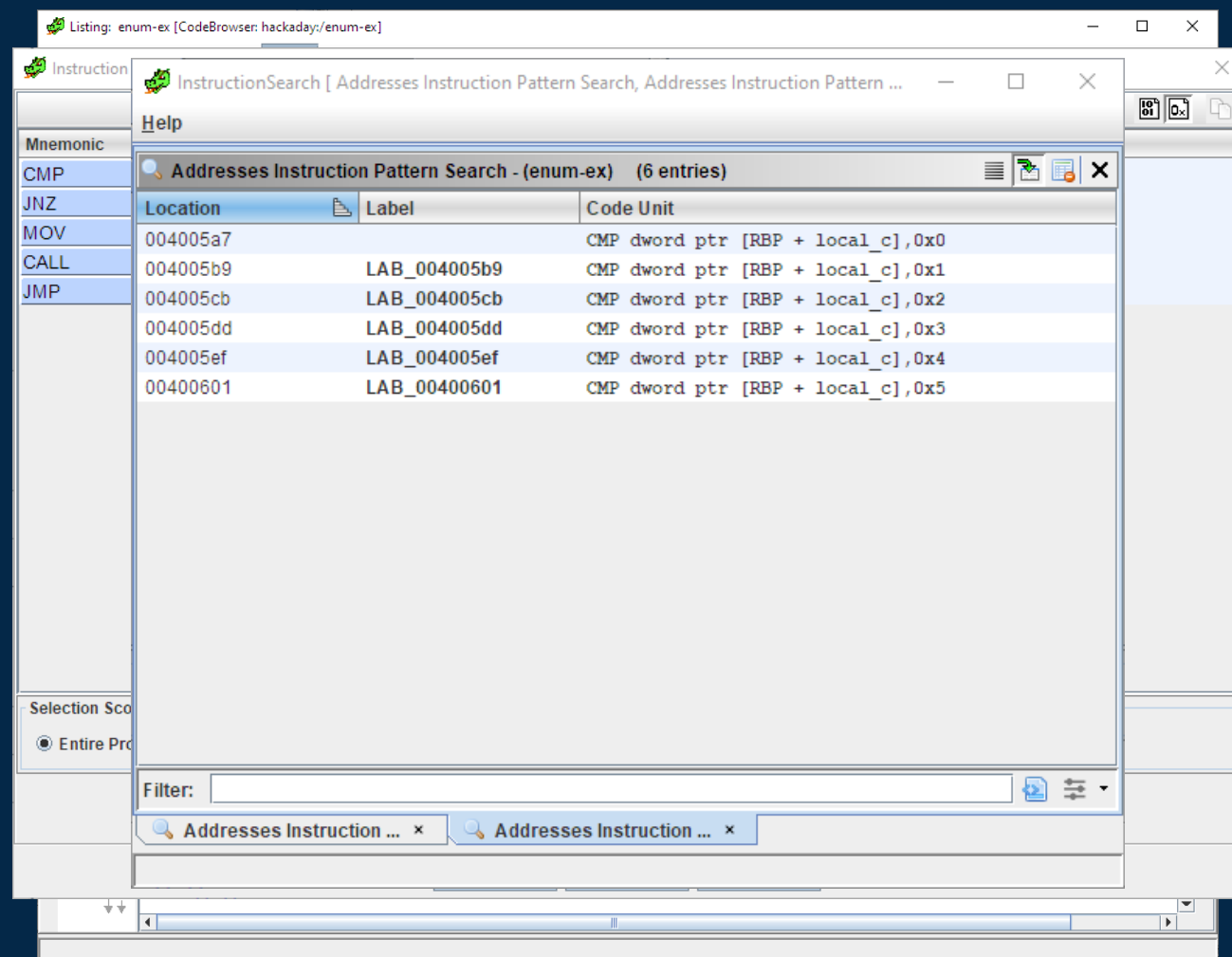


#Ghidra Tip: Searching

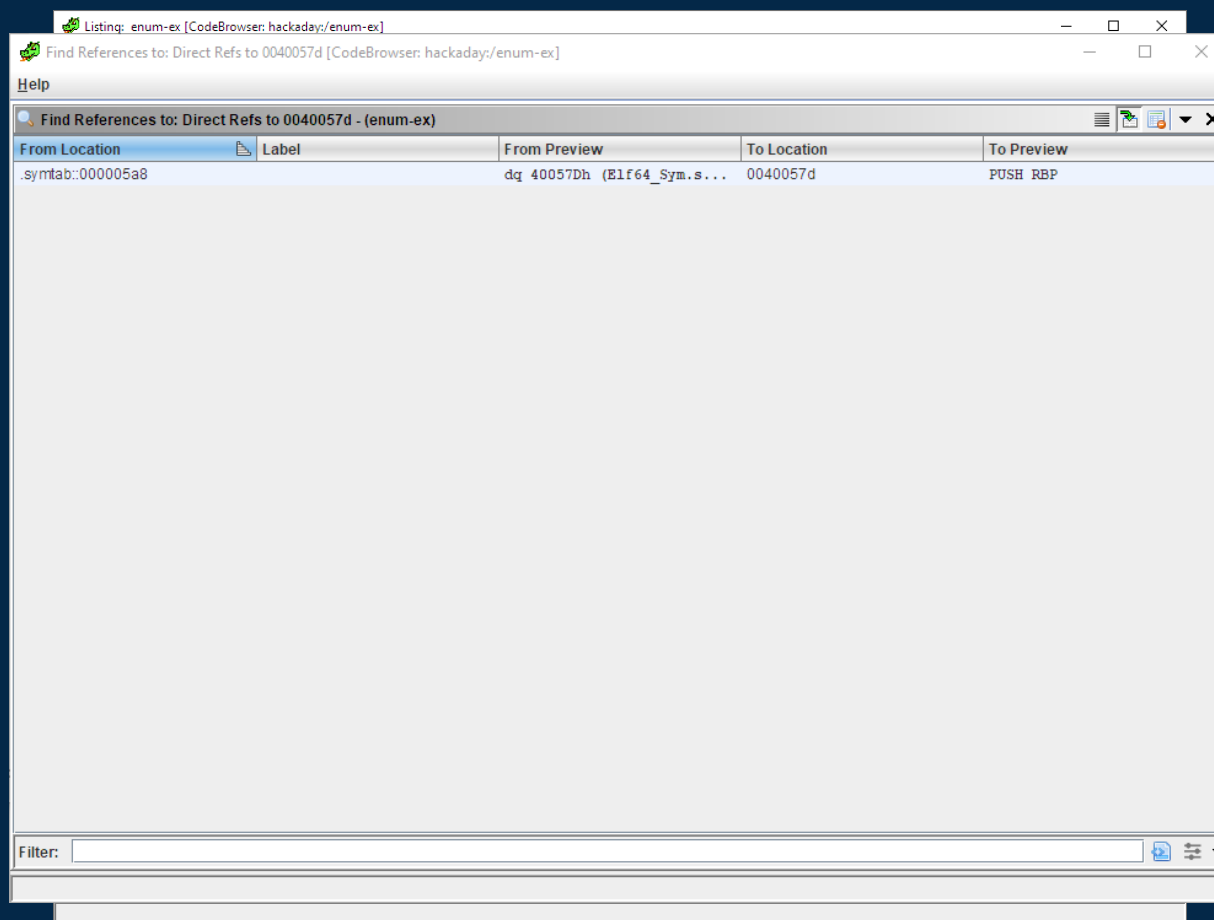
- Instructions
 - Allows the user to search for uses of a particular instruction
- Direct References
 - Searches the program for references to that particular address / location
- Instruction Patterns
 - Searches for user specified sequences of instructions in program
- Scalars
 - Search for immediate value
- Strings
 - Search for defined strings



#Search: Instruction Patterns



#Search: Direct References

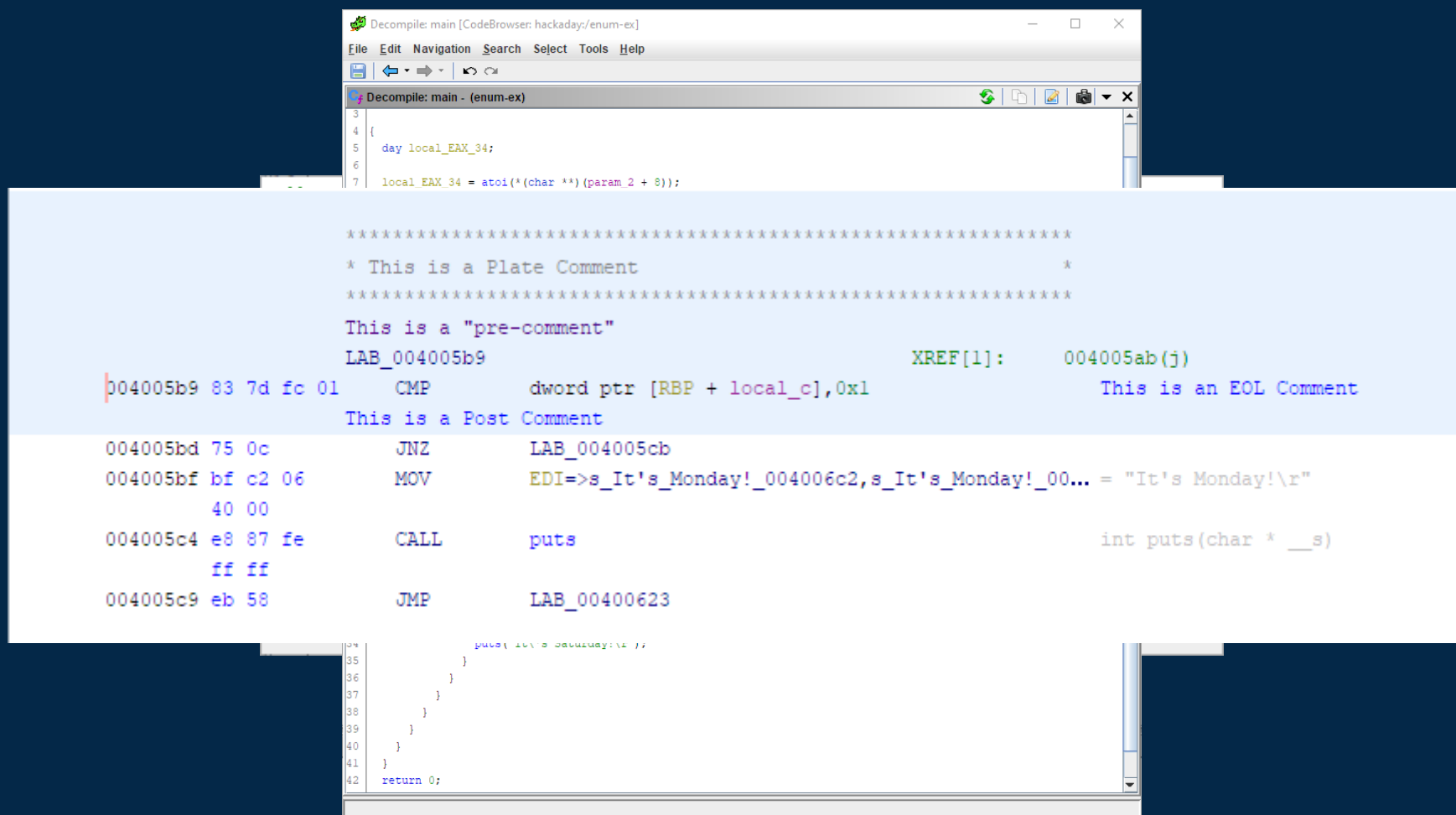


#Ghidra Tip: Comments

- Comments can be added to both the listing view and the decompiler view
- There are various types of comments that can be added:
 - Post
 - Plate
 - EOL
 - Pre
- To add a comment, right click and select “Comment” or press “;”



#Ghidra Tip: Comments



The screenshot displays the Ghidra decompiler interface for a function named 'main'. The decompiled C code at the top shows a local variable 'local_EAX_34' being assigned the value of 'atoi((char *) (param_2 + 8))'. Below this, the assembly view shows several instructions with associated comments:

- A block comment: `*****
* This is a Plate Comment
*****`
- A pre-comment: `This is a "pre-comment"`
- An instruction with a post-comment: `004005b9 83 7d fc 01 CMP dword ptr [RBP + local_c],0x1
This is a Post Comment`
- An instruction with an EOL comment: `004005bd 75 0c JNZ LAB_004005cb
004005bf bf c2 06 MOV EDI=>s_It's_Monday!_004006c2,s_It's_Monday!_00... = "It's Monday!\r"`
- An instruction with a function signature comment: `004005c4 e8 87 fe CALL puts
ff ff
int puts(char * __s)`
- A jump instruction: `004005c9 eb 58 JMP LAB_00400623`

The bottom of the window shows the original assembly code with its corresponding comments, such as `puts("It's Monday!\r");`.



#Wrap Up

- Today we reviewed how to identify various C constructs in C
 - Even though you have a decompiler, it is important to be able to recognize these constructs!
- We also reviewed multiple Ghidra tips to make the reversing process more streamlined
- The exercises for this course are available on the github page!



#Questions

?



#Ghidra Tip: Checksum Tool



#External Libraries



#External Libraries: Ghidra

