

# Introduction to Software Reverse Engineering with Ghidra Session 2: C to ASM

Hackaday U  
Matthew Alt



# #Outline: C to ASM

- Class Admin
- Ghidra Exercise Tips
- Control Flow Review
- Function Calls
- Loops and Iterators
- Switch Statements
- Local and Global Variables
- Array Accesses and Manipulation



# #Session Goals

- Learn how to identify basic C constructs in assembly language
- Review C constructs and their assembly representation
  - Heap memory
  - Function calls
  - Switch cases
  - Loops / iterations
  - Local and global variables
- How to view / modify these constructs with Ghidra



# #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



# #Program Startup

- You may have noticed in our exercises, that there is additional code outside of our main functions!
  - These additional blocks of code are used to properly launch the binary
- Program startup and behavior is defined by the System V ABI
- Within the ELF header, there is an `e_entry` field, this field points to the `_start()` function
  - This is what eventually calls `main`!



# #Program Startup

- All of the exercises in this course conform to the System V ABI
  - This determines how a program starts and is loaded!
  - Determines how information in the ELF header is parsed
- We can use the information from this ABI to help us when reverse engineering
  - `main()` is our entry point
  - The arguments to main are determined by the ABI!



# #Ghidra Tip: Function Signatures

- Function signatures can be edited in Ghidra, altering:
  - Argument count
  - Argument types
  - Return values
- Fixing the function signature can greatly improve decompiler output
- Right click the function name -> Edit Function Signature



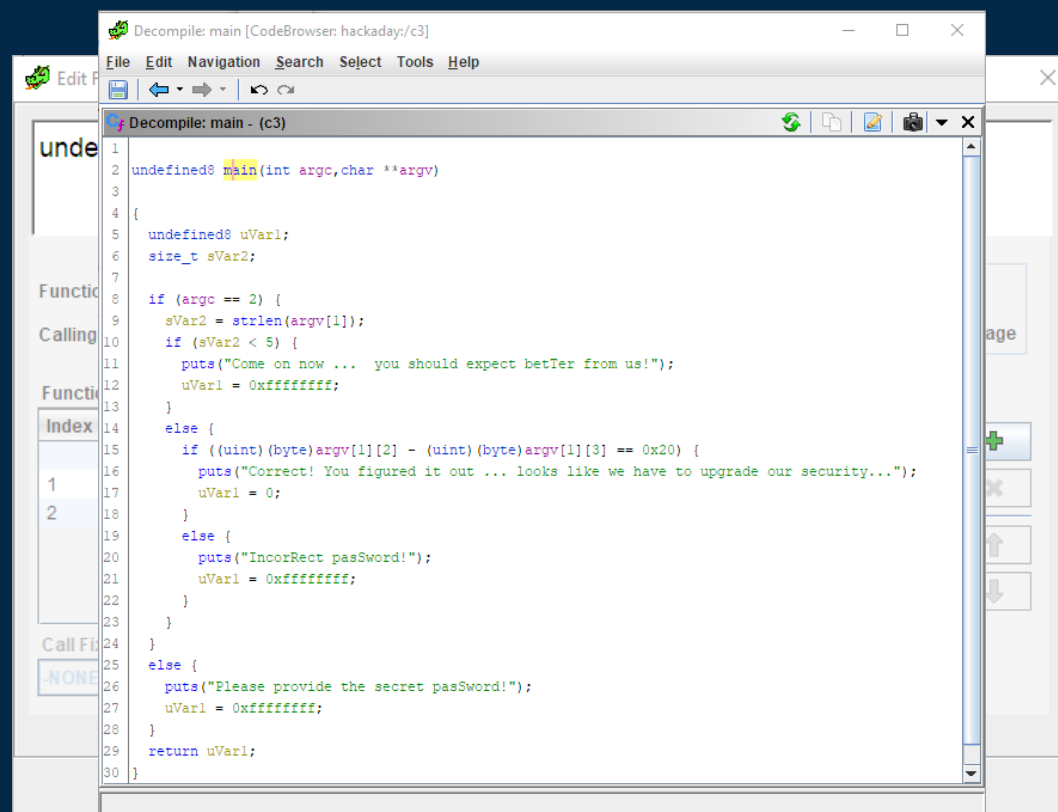
# #Exercise Tip: argc/argv

- The C standard defines the arguments passed to a main function
  - `int argc` = Argument Count
  - `char **argv` = Argument vector
- We can apply these types to our function prototype in Ghidra
  - This makes the function more read-able
- The changes will also propagate to the assembly listing





# #Ghidra Tip: Function Signatures



The screenshot shows the Ghidra decompiler interface. The main window displays the decompiled C code for the `main` function. The code is as follows:

```
1  undefined8 main(int argc, char **argv)
2
3
4  {
5      undefined8 uVar1;
6      size_t sVar2;
7
8      if (argc == 2) {
9          sVar2 = strlen(argv[1]);
10         if (sVar2 < 5) {
11             puts("Come on now ... you should expect better from us!");
12             uVar1 = 0xffffffff;
13         }
14         else {
15             if ((uint)(byte)argv[1][2] - (uint)(byte)argv[1][3] == 0x20) {
16                 puts("Correct! You figured it out ... looks like we have to upgrade our security...");
17                 uVar1 = 0;
18             }
19             else {
20                 puts("Incorrect password!");
21                 uVar1 = 0xffffffff;
22             }
23         }
24     }
25     else {
26         puts("Please provide the secret password!");
27         uVar1 = 0xffffffff;
28     }
29     return uVar1;
30 }
```

On the left side, the 'Function Index' pane shows a list of functions, with 'main' selected. The 'Calling' pane is empty, and the 'Call Fit' pane shows '-NONE-'. The 'Function Signature' pane shows the signature `undefined8 main(int argc, char **argv)`.

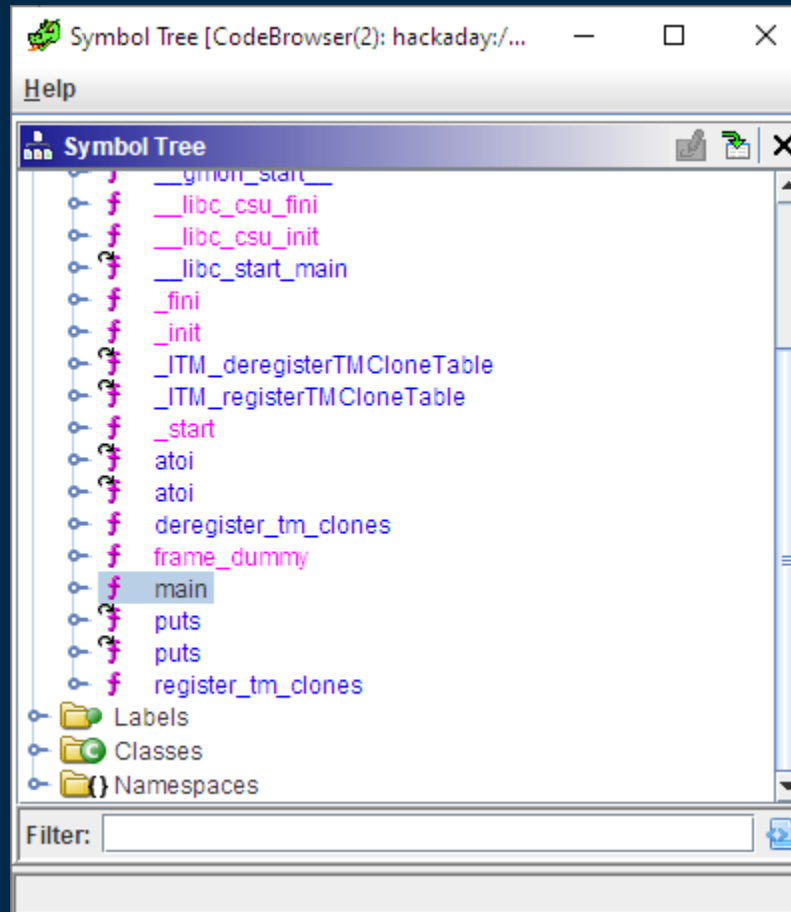


# #Ghidra Tip: Imports / Exports

- Imports and exports can be viewed from the Symbol tree
  - Imports: libraries that are utilized by your target binary
  - Exports: Exposed information about our binary for the operating system loader
- When importing a binary, users can also specify a path for libraries that are imported
- For our challenges, “main” is always a good starting point!
  - Defined by the ABI!
  - If you can’t find main, start with \_start!



# #Ghidra Tip: Imports / Exports



# #Control Flow

- Control flow is the order in which instructions are executed
- Common statements used to manipulate control flow include:
  - if /else
  - goto
  - switch
  - while
- These statements can be reconstructed by analyzing the resulting assembly code



# #Control Flow

- RIP contains the address of the next instruction to execute
- The JMP instruction (and others) can alter RIP
  - JMP ADDR
- JMP can also selectively execute based on the RFLAGS register
  - JE: Jump if equal/zero
  - JNE: Jump if not equal/nonzero
  - JG: Jump if greater (signed)
  - JL: Jump if less (signed)



# #Control Flow: Example

```
000000000040057d <main>:
40057d: 55                push    rbp
40057e: 48 89 e5          mov     rbp,rsi
400581: 48 83 ec 20       sub     rsp,0x20
400585: 89 7d ec          mov     DWORD PTR [rbp-0x14],edi
400588: 48 89 75 e0       mov     QWORD PTR [rbp-0x20],rsi
40058c: 48 8b 45 e0       mov     rax,QWORD PTR [rbp-0x20]
400590: 48 83 c0 08       add     rax,0x8
400594: 48 8b 00          mov     rax,QWORD PTR [rax]
400597: ba 0a 00 00 00    mov     edx,0xa
40059c: be 00 00 00 00    mov     esi,0x0
4005a1: 48 89 c7          mov     rdi,rax
4005a4: b8 00 00 00 00    mov     eax,0x0
4005a9: e8 d2 fe ff ff    call    400480 <strtol@plt>
4005ae: 48 98             cdq     rax
4005b0: 48 89 45 f8       mov     QWORD PTR [rbp-0x8],rax
4005b4: 48 83 7d f8 64    cmp     QWORD PTR [rbp-0x8],0x64
4005b9: 7e 0c             jle     4005c7 <main+0x4a>
4005bb: bf 64 06 40 00    mov     edi,0x400664
4005c0: e8 8b fe ff ff    call    400450 <puts@plt>
4005c5: eb 0a             jmp     4005d1 <main+0x54>
4005c7: bf 76 06 40 00    mov     edi,0x400676
4005cc: e8 7f fe ff ff    call    400450 <puts@plt>
4005d1: b8 00 00 00 00    mov     eax,0x0
4005d6: c9              leave   rbp
4005d7: c3              ret
4005d8: 0f 1f 84 00 00 00 00 00  nop     DWORD PTR [rax+rax*1+0x0]
4005df: 00
```

If it was less than 100 we jump to 0x4005C7, otherwise we continue to 0x4005BB

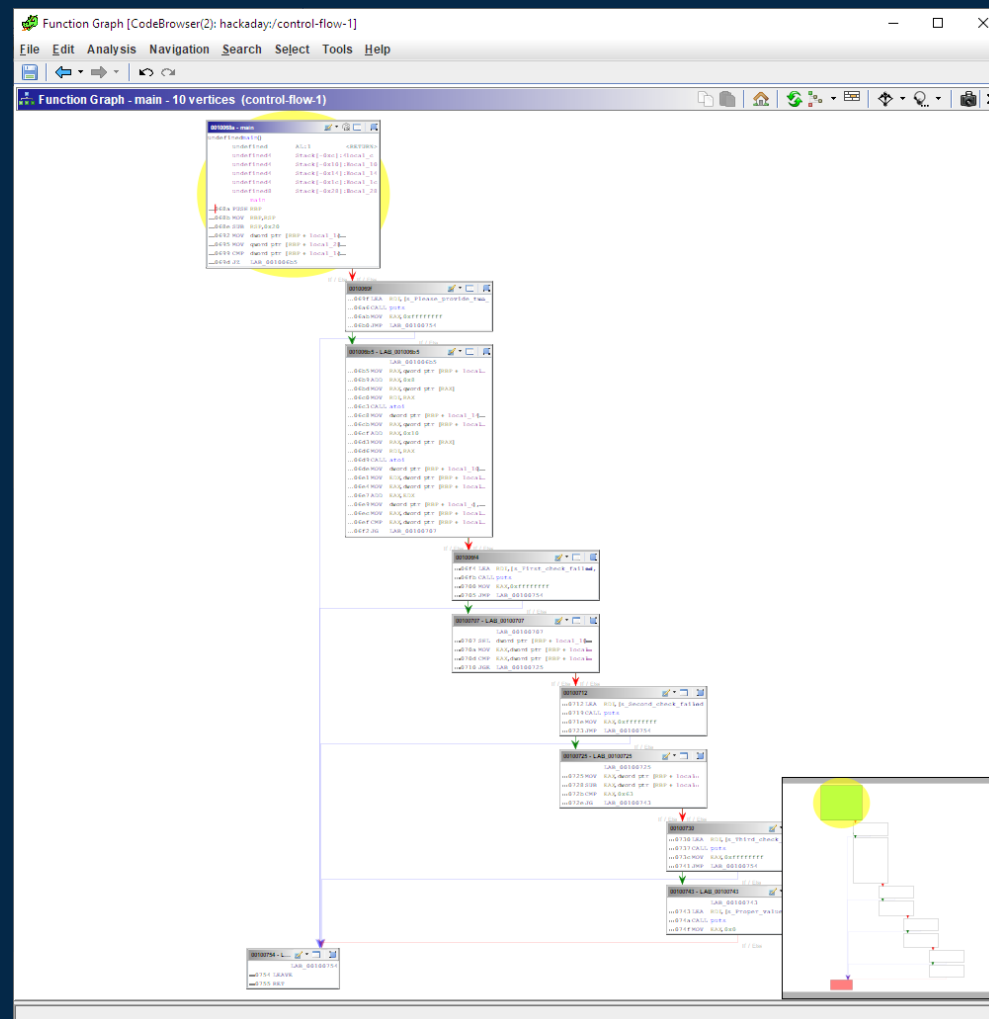


# #Ghidra Tip: Graph View

- When looking at multiple branches, graph view can be helpful
  - Displays multiple blocks of code representing each branch
- Graph view can be useful when determining control flow
- This can be entered through the following:
  - Highlight a function name
  - Window -> Function Graph



# #Ghidra Tip: Graph View





# #Control Flow: Exercise 1

- Load the exercise `session-two/exercises/control-flow-1`
- How many compare statements are in main?
- What are the three values that are being compared?
- Can you pass all three checks?



# #Switch Cases

- Switch cases allow a variable to be compared against a list of values
  - Each value being compared against is a case
- Switch statements can contain any number of cases
- The expression for the case must have the same data type as the variable in the switch
- The switch statement terminates on a break statement



# #Switch Cases: C to ASM

## • C Code

```
int a = atoi(argv[1]);
switch((char)a){
    case 'a':
        a = 1;
        break;
    case 'b':
        a = 2;
        break;
    default:
        a = 3;
        break;
}
return 0;
```

## • Assembly Code

```
6c0: call 560 <atoi@plt>
6c5: mov  DWORD PTR [rbp-0x4],eax
6c8: mov  eax,DWORD PTR [rbp-0x4]
6cb: movsx eax,al
6ce: cmp  eax,0x61
6d1: je   6da <main+0x50>
6d3: cmp  eax,0x62
6d6: je   6e3 <main+0x59>
6d8: jmp  6ec <main+0x62>
6da: mov  DWORD PTR [rbp-0x4],0x1
6e1: jmp  6f4 <main+0x6a>
6e3: mov  DWORD PTR [rbp-0x4],0x2
6ea: jmp  6f4 <main+0x6a>
6ec: mov  DWORD PTR [rbp-0x4],0x3
```



# #Ghidra Tip: Converting Data

- In the listing view, data types can be converted
  - Hexadecimal, decimal, char, etc
- Right click an immediate value and select: “Convert”
  - Multiple representations can be applied
- This can be used to make the decompiler output more readable



# #Ghidra Tip: Converting Data

Decompile: main - (switch-statement)

```

1  MOV
2  MOV
3  MOVSB
4  CMP
5  JZ
6  CMP
7  JZ
8  JMP
9  if (param_1 == 2) {
10     iVar1 = atoi(*(char **) (param_2 + 8));
11     if (iVar1 == 'a') {
12         local_c = 1;
13     }
14     else {
15         if (iVar1 == 'b') {
16             local_c = 2;
17         }
18         else {
19             local_c = 3;
20         }
21     }
22     uVar2 = (ulong)local_c;
23 }
24 else {
25     uVar2 = 0xffffffff;
26 }
27 return uVar2;
28 }
29

```



# #Loops

- Loops allow repeated execution of a block of code
  - One of the most common programming structures
  - Statements in the loop are executed sequentially
- Loops can be implemented in assembly in multiple ways
  - CMP -> JMP
  - LOOP
  - REP
- Loop typically operate under a conditional code
  - This code is used to determine whether the loop should execute



# #Loops: C to ASM – for

## • C Code

```
int count = atoi(argv[1]);  
int sum = 0;  
int x = 0;  
for (x = 0; x < count; x++) {  
    sum += x;  
}
```

## • Assembly Code

```
6ac: call 560 <atoi@plt>  
6b1: mov  DWORD PTR [rbp-0x4],eax  
6b4: mov  DWORD PTR [rbp-0xc],0x0  
6bb: mov  DWORD PTR [rbp-0x8],0x0  
6c2: mov  DWORD PTR [rbp-0x8],0x0  
6c9: jmp  6d5 <main+0x4b>  
6cb: mov  eax,DWORD PTR [rbp-0x8]  
6ce: add  DWORD PTR [rbp-0xc],eax  
6d1: add  DWORD PTR [rbp-0x8],0x1  
6d5: mov  eax,DWORD PTR [rbp-0x8]  
6d8: cmp  eax,DWORD PTR [rbp-0x4]  
6db: jl   6cb <main+0x41>
```



# #Loops: C to ASM – while

## • C Code

```
int count = atoi(argv[1]);  
int sum = 0;  
int x = 0;  
while(x < count){  
    sum += x;  
    x += 1;  
}
```

## • Assembly Code

```
6ac: call 560 <atoi@plt>  
6b1: mov  DWORD PTR [rbp-0x4],eax  
6b4: mov  DWORD PTR [rbp-0xc],0x0  
6bb: mov  DWORD PTR [rbp-0x8],0x0  
6c2: jmp  6ce <main+0x44>  
6c4: mov  eax,DWORD PTR [rbp-0x8]  
6c7: add  DWORD PTR [rbp-0xc],eax  
6ca: add  DWORD PTR [rbp-0x8],0x1  
6ce: mov  eax,DWORD PTR [rbp-0x8]  
6d1: cmp  eax,DWORD PTR [rbp-0x4]  
6d4: jl   6c4 <main+0x3a>
```



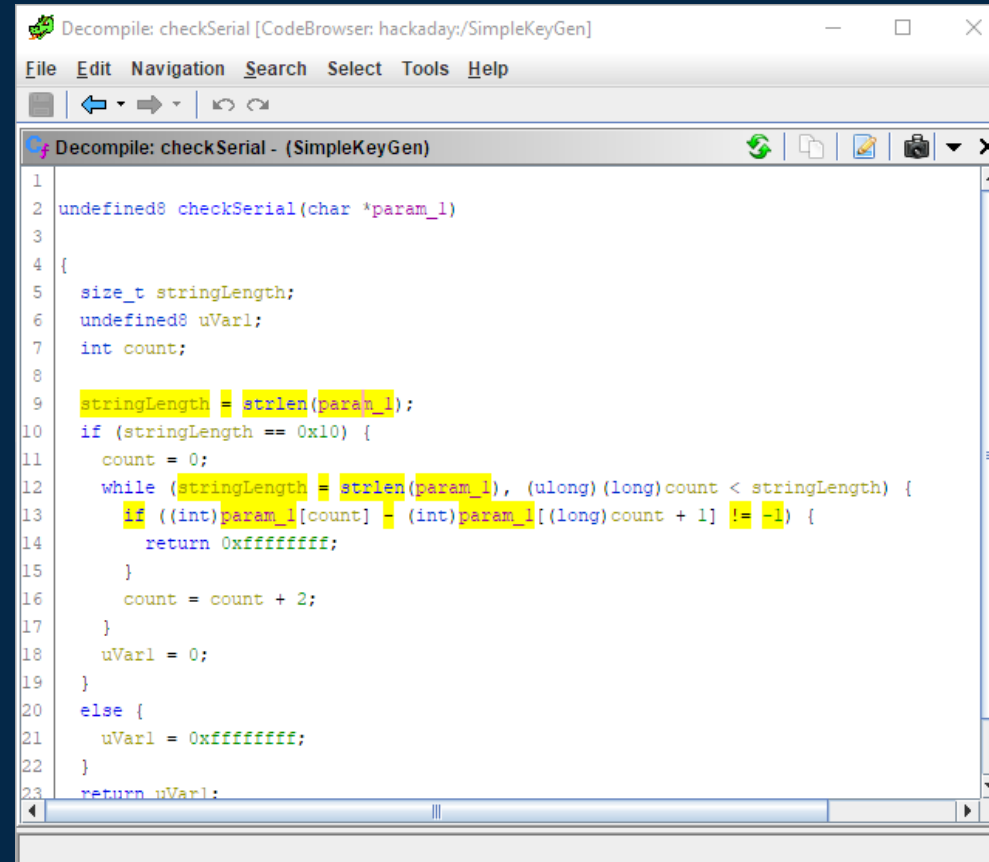


# #Ghidra Tip: Highlighting / Slicing

- When viewing the assembly listing or decompiler view, items can be highlighted
  - Useful for tracking register usage in larger functions
- Slicing can be applied in the decompiler window
  - This will display usage of the selected variable
- Ghidra will attempt to synchronize highlights between disassembly / decompiler views



# #Ghidra Tip: Highlighting / Slicing



```
Decompile: checkSerial [CodeBrowser: hackaday:/SimpleKeyGen]
File Edit Navigation Search Select Tools Help
Decompile: checkSerial - (SimpleKeyGen)
1
2 undefined8 checkSerial(char *param_1)
3
4 {
5     size_t stringLength;
6     undefined8 uVar1;
7     int count;
8
9     stringLength = strlen(param_1);
10    if (stringLength == 0x10) {
11        count = 0;
12        while (stringLength == strlen(param_1), (ulong)(long)count < stringLength) {
13            if ((int)param_1[count] - (int)param_1[(long)count + 1] != -1) {
14                return 0xffffffff;
15            }
16            count = count + 2;
17        }
18        uVar1 = 0;
19    }
20    else {
21        uVar1 = 0xffffffff;
22    }
23    return uVar1;
}
```

Forward Inst Slice  
highlights usages  
moving forward in the  
function



# #Loops and Iterations: Exercise

- Load the exercise `session-two/exercises/loop-example-1`
- How many times does this loop run?
- What is this loop looking for?
  - Do the values used represent anything?
- Can you get access?



# #Variables

- When a variable is declared, it is declared within a particular scope
  - The scope defines how accessible a variable is
  - We will define two types of scope for this course: Local and Global
- Local Variables
  - Defined within a function
  - Only accessible within the function
- Global Variables
  - Declared outside of a function
  - Can be used in all functions



# #Variables: C to ASM

## • C Code

```
int globalVar = 0x15;
```

```
int main(int argc, char *argv[]){  
    int localVar = 0x10;  
    int localVarTwo = 0x11;  
    globalVar += localVar;  
    localVarTwo += globalVar;  
    globalVar = 0;  
    return 0;  
}
```

## • Assembly Code

```
5fa: push  rbp  
5fb: mov   rbp, rsp  
5fe: mov   DWORD PTR [rbp-0x14], edi  
601: mov   QWORD PTR [rbp-0x20], rsi  
605: mov   DWORD PTR [rbp-0x8], 0x10  
60c: mov   DWORD PTR [rbp-0x4], 0x11  
613: mov   edx, DWORD PTR [rip+0x2009f7] # 201010 <globalVar>  
619: mov   eax, DWORD PTR [rbp-0x8]  
61c: add   eax, edx  
61e: mov   DWORD PTR [rip+0x2009ec], eax # 201010 <globalVar>  
624: mov   eax, DWORD PTR [rip+0x2009e6] # 201010 <globalVar>  
62a: add   DWORD PTR [rbp-0x4], eax  
62d: mov   DWORD PTR [rip+0x2009d9], 0x0 # 201010 <globalVar>  
637: mov   eax, 0x0  
63c: pop   rbp  
63d: ret  
63e: xchg  ax, ax
```

Stored in .data section!

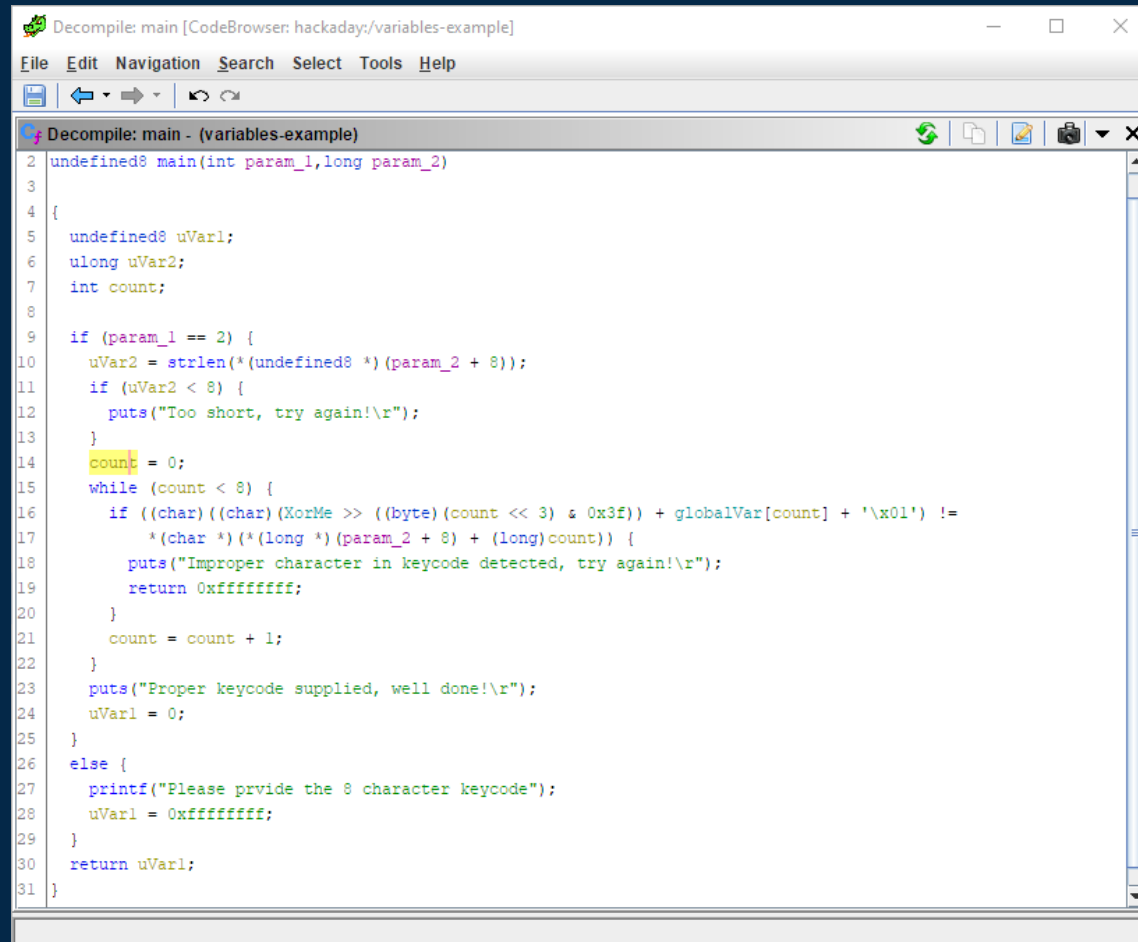


# #Ghidra Tip: Labelling/Renaming

- Variables and offsets can be labelled in Ghidra
  - Useful for labelling memory regions
  - Makes pseudo code more understandable
- Ghidra will attempt to synchronize the variable names between the listing view and decompiler view



# #Ghidra Tip: Labelling/Renaming



```
Decompile: main [CodeBrowser: hackaday/variables-example]
File Edit Navigation Search Select Tools Help
Decompile: main - (variables-example)
2 undefined8 main(int param_1,long param_2)
3
4 {
5     undefined8 uVar1;
6     ulong uVar2;
7     int count;
8
9     if (param_1 == 2) {
10         uVar2 = strlen((undefined8 *) (param_2 + 8));
11         if (uVar2 < 8) {
12             puts("Too short, try again!\r");
13         }
14         count = 0;
15         while (count < 8) {
16             if ((char)((char)(XorMe >> ((byte)(count << 3) & 0x3f)) + globalVar[count] + '\x01') !=
17                 *(char *) ((long *) (param_2 + 8) + (long)count)) {
18                 puts("Improper character in keycode detected, try again!\r");
19                 return 0xffffffff;
20             }
21             count = count + 1;
22         }
23         puts("Proper keycode supplied, well done!\r");
24         uVar1 = 0;
25     }
26     else {
27         printf("Please provide the 8 character keycode");
28         uVar1 = 0xffffffff;
29     }
30     return uVar1;
31 }
```



# #Variables: Example

- Load the exercise `session-two/exercises/variables-example`
- How many global variables are being used?
- How many local variables are in use?
- Can you find the proper keycode?





# #Functions

- Functions are called using the `call` instruction
  - `call` pushes the return address to the stack when called
- The first 6 function parameters are passed in through registers
  - RDI,RSI,RDX,RCX,R8,R9
  - After this parameters are passed through the stack
  - Large parameters /structures passed by value are passed through the stack
- `ret` is used to return from a function
  - The return address is popped off the stack and placed into RIP



# #Functions: Stack Usage

```
0000000000400517 <main>:
400517: 55          push    rbp
400518: 48 89 e5    mov     rbp, rsp
40051b: 48 83 ec 10  sub     rsp, 0x10
40051f: 89 7d fc    mov     DWORD PTR [rbp-0x4], edi
400522: 48 89 75 f0  mov     QWORD PTR [rbp-0x10], rsi
400526: bf 0f 00 00 00  mov     edi, 0xf
40052b: e8 bd ff ff ff  call    4004ed <add and print>
400530: b8 00 00 00 00  mov     eax, 0x0
400535: c9          leave
400536: c3          ret
400537: 66 0f 1f 84 00 00 00  nop     WORD PTR [rax+rax*1+0x0]
40053e: 00 00
```



# #Functions: Calling Conventions

- Calling conventions define how function calls are implemented
  - How arguments are passed to functions
  - How return values are pass back from functions
  - Stack management and register cleanup
- GNU/Linux uses the System V AMD64 ABI
  - ABI = Application Binary Interface
- Calling convention defined the epilogue / prologue for functions



# #Functions: Prologue/Epilogue

- Functions can be thought of as three components:
  - Prologue
  - Body
  - Epilogue
- The prologue reserves space for variables on the stack
- The epilogue cleans up the stack frame and returns it to it's original state



# #Function: C to ASM

## C Code

```
void add_and_print(int x)
{
    int local_var = 10;
    int local_var_two = 14;
    int sum = 0;
    sum += x;
    sum += local_var;
}
```

## Assembly Code

```
4004ed: push rbp
4004ee: mov  rbp, rsp
4004f1: mov  DWORD PTR [rbp-0x14], edi
4004f4: mov  DWORD PTR [rbp-0xc], 0xa
4004fb: mov  DWORD PTR [rbp-0x8], 0xe
400502: mov  DWORD PTR [rbp-0x4], 0x0
400509: mov  eax, DWORD PTR [rbp-0x14]
40050c: add  DWORD PTR [rbp-0x4], eax
40050f: mov  eax, DWORD PTR [rbp-0xc]
400512: add  DWORD PTR [rbp-0x4], eax
400515: pop  rbp
400516: ret
```

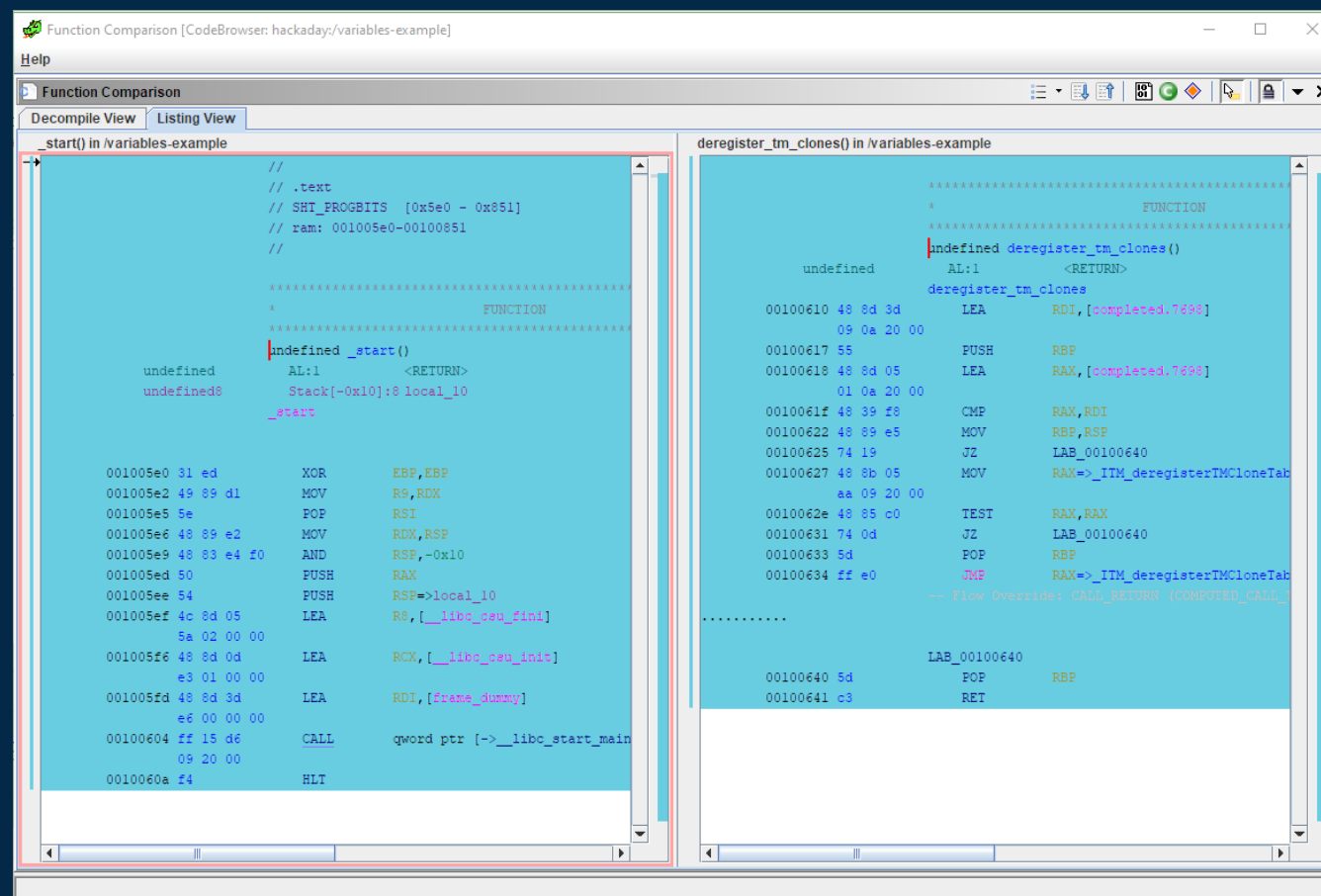


# #Ghidra Tip: Functions

- Ghidra can perform function comparisons
  - This can be useful for doing side by side comparisons of similar functions
- Ghidra will display the two functions side by side
  - Listing view
  - Decompiler view
- To view the function comparison window, highlight two functions, then right click -> Compare Selected Functions



# #Ghidra Tip: Functions



The screenshot shows the Ghidra Function Comparison window. The title bar reads "Function Comparison [CodeBrowser: hackaday/variables-example]". The window has a menu bar with "Help" and a toolbar with various icons. Below the menu bar are two tabs: "Decompile View" (selected) and "Listing View". The main area is split into two panes. The left pane shows the decompiled code for the function `_start() in /variables-example`. The right pane shows the decompiled code for the function `deregister_tm_clones() in /variables-example`. Both panes display assembly instructions with their addresses and hex values on the left, and the corresponding assembly instructions on the right. The left pane also includes comments and labels like `LAB_00100640`. The right pane includes comments like `--- Flow Override: CALL_RETURN (COMPUTED_CALL_...`.

```
//  
// .text  
// SHI_PROGBITS [0x5e0 - 0x851]  
// ram: 001005e0-00100851  
//  
*****  
*  
*****  
FUNCTION  
*****  
undefined _start()  
AL:1 <RETURN>  
Stack[-0x10]:8 local_10  
_start  
  
001005e0 31 ed XOR EBP,EBP  
001005e2 49 89 d1 MOV R9,RDX  
001005e5 5e POP RSI  
001005e6 48 89 e2 MOV RDX,RSP  
001005e9 48 83 e4 f0 AND RSP,-0x10  
001005ed 50 PUSH RAX  
001005ee 54 PUSH RSP=>local_10  
001005ef 4c 8d 05 LEA R8,[__libc_sws_final]  
5a 02 00 00  
001005f6 48 8d 0d LEA RCX,[__libc_sws_init]  
e3 01 00 00  
001005fd 48 8d 3d LEA RDI,[frame_dummy]  
e6 00 00 00  
00100604 ff 15 d6 CALL qword ptr [->__libc_start_main  
09 20 00  
0010060a f4 HLT
```

```
*****  
*  
*****  
FUNCTION  
*****  
undefined deregister_tm_clones()  
AL:1 <RETURN>  
deregister_tm_clones  
  
00100610 48 8d 3d LEA RDI,[completed.7424]  
09 0a 20 00  
00100617 55 PUSH RBP  
00100618 48 8d 05 LEA RAX,[completed.7424]  
01 0a 20 00  
0010061f 48 39 f8 CMP RAX,RDI  
00100622 48 89 e5 MOV RBP,RSP  
00100625 74 19 JZ LAB_00100640  
00100627 48 8b 05 MOV RAX=>_ITM_deregisterTMCloneTab  
aa 09 20 00  
0010062e 48 85 c0 TEST RAX,RAX  
00100631 74 0d JZ LAB_00100640  
00100633 5d POP RBP  
00100634 ff e0 JMP RAX=>_ITM_deregisterTMCloneTab  
--- Flow Override: CALL_RETURN (COMPUTED_CALL_...  
.....  
LAB_00100640  
00100640 5d POP RBP  
00100641 c3 RET
```



# #Functions: Exercise 1

- Load the exercise `session-two/exercises/func-example-1`
- How many functions does the auto analysis discover?
- How many local variables are present in the each function?
  - What are their values?
- Do any of these functions take arguments?
  - If so, what are the arguments?





# #Heap Memory

- The heap is used for dynamic memory allocations
  - Used when the size of a variable can be varied
  - `malloc/calloc` – Used to allocate
- Heap memory is not managed automatically
  - Developers must manage it manually
  - `free(var)` – used to free memory
  - Failure to do so results in “memory leaks”
- Heap variables can be accessed globally



# #Heap Memory: Stack Vs. Heap

- Stack Memory
  - No need to de-allocate
  - Local variables only
  - Limited by stack size (OS dependent)
  - Statically sized variables
- Heap Memory
  - Variables can be accessed globally
  - No limit on size (within reason)
  - Must be managed by user



# #Heap Memory: Exercise 1

- Load the exercise session-two/exercises/heap-example-1
- How much memory is being allocated via malloc?
- How is this program different than the loop example?



# #Array Accesses

- Array accesses often utilize the LEA instruction
- LEA = Load Effective Address
  - Loads the address of the memory calculation into the register
  - Does not dereference the value as MOV would
- LEA EAX, [ EAX + EBX + 1234567 ]
  - Stores the value of EAX+EBX+1234567 into EAX
  - Does not dereference!



# #Array Accesses: C to ASM

## C Code

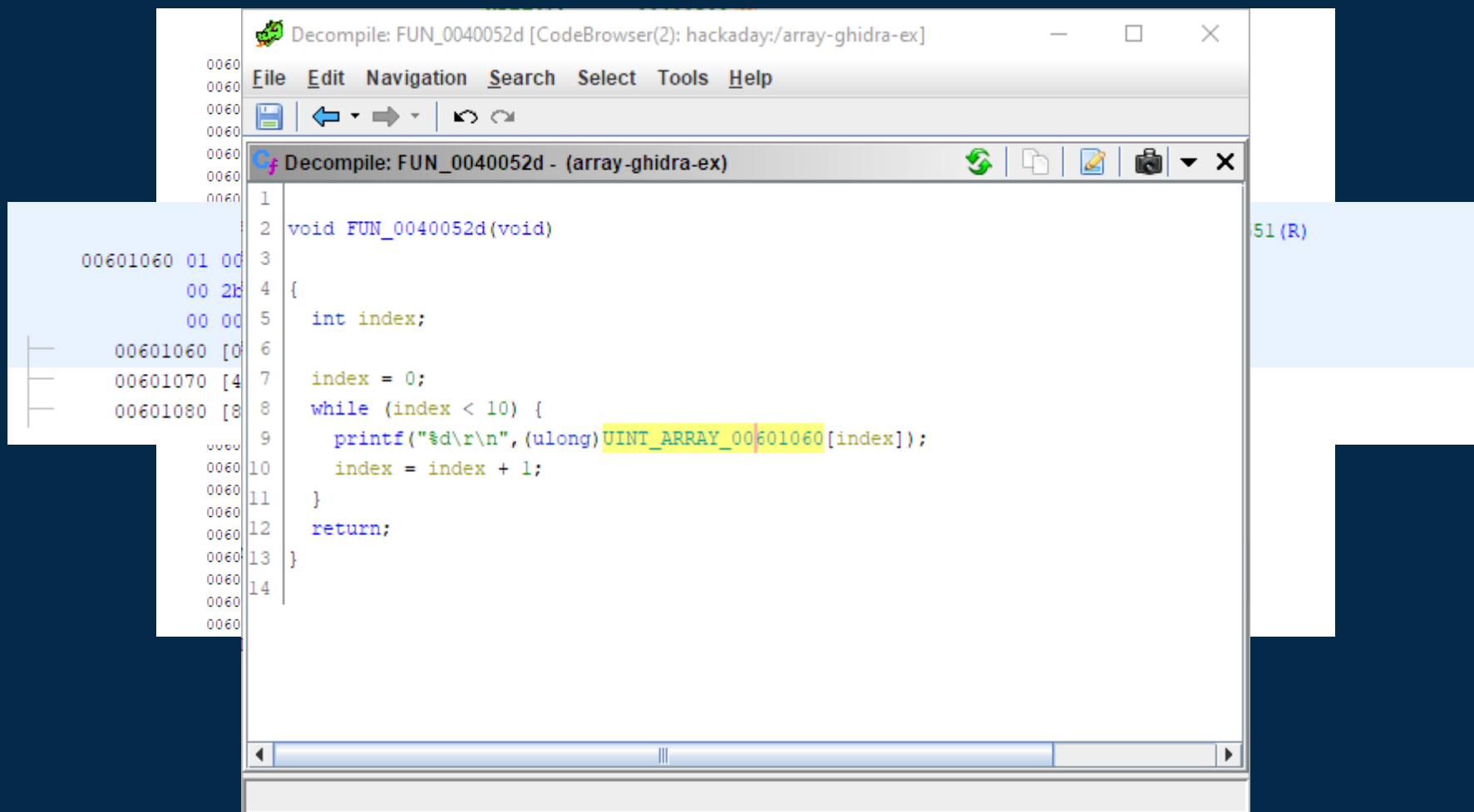
```
int nums[10];  
int main(int argc, char *argv[]){  
    int x;  
    for(x=0;x<10;x++){  
        nums[x] = x;  
    }  
    return 0;  
}
```

## Assembly Code

```
0000000000000633 <main>:  
633: push    rbp  
634: mov     rbp, rsp  
637: mov     DWORD PTR [rbp-0x14], edi  
63a: mov     OWORD PTR [rbp-0x20], rsi  
63e: mov     DWORD PTR [rbp-0x4], 0x0  
645: jmp     665 <main+0x32>  
647: mov     eax, DWORD PTR [rbp-0x4]  
64a: cdq       
64c: lea     rcx, [rax*4+0x0]  
653: 00  
654: lea     rax, [rip+0x2009e5] # 201040 <nums>  
65b: mov     edx, DWORD PTR [rbp-0x4]  
65e: mov     DWORD PTR [rcx+rax*1+0x0], edx  
661: add     DWORD PTR [rbp-0x4], 0x1  
665: cmp     DWORD PTR [rbp-0x4], 0x9  
669: jle     647 <main+0x14>  
66b: mov     eax, 0x0  
670: pop     rbp  
671: ret  
672: nop     WORD PTR cs:[rax+rax*1+0x0]  
679: 00 00 00  
67c: 0f 1f 40 00          nop     DWORD PTR [rax+0x0]
```



# #Ghidra Tip: Creating Arrays



The screenshot shows the Ghidra CodeBrowser window for a decompiled function named FUN\_0040052d. The window has a menu bar (File, Edit, Navigation, Search, Select, Tools, Help) and a toolbar. The code is as follows:

```
1 void FUN_0040052d(void)
2 {
3     int index;
4     index = 0;
5     while (index < 10) {
6         printf("%d\r\n", (ulong)UINT_ARRAY_00601060[index]);
7         index = index + 1;
8     }
9     return;
10 }
```

On the left, a memory map shows addresses 00601060, 00601070, and 00601080. On the right, a register window shows 51 (R). The array access in the code is highlighted in yellow.



# #Array Accesses: Exercise

- Load the exercise session-two/exercises /array-example
- How many different arrays are in use?
  - What is their scope, are they global?
- Can you solve the password for all 4 index values?



# #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

?

