

# Internet of Things Reverse Engineering with Ghidra

Troy Shurtleff (Author)

Copyright 2019 BAE Systems

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<https://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

# Background and Introduction

# What is Software Reverse Engineering?

- In 1990 Elliot Chikofsky and Dr. James Cross defined reverse engineering in the context of software as follows [1]:

Reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

- In a security context, reverse engineering is often performed to
  - identify or characterize vulnerabilities and/or
  - develop exploits for those vulnerabilities
- As with almost anything in the security domain intentions can either be malicious or non-malicious
- Here we examine the whys and hows of software reverse engineering...

# Why Reverse Engineer Software?

- Malware
  - Malware authors may reverse engineer software to identify vulnerabilities and develop exploits
    - Current generations of IoT malware move beyond using default credentials and are now exploiting other software vulnerabilities in devices [2]
  - Defenders may reverse engineer malware to understand the purpose of the malware, understand exploitation techniques used and develop solutions to detect and prevent infections from the same or similar malware

# Why Reverse Engineer Software?

- Digital Rights Management
  - Content providers often implement cryptologic controls to protect their intellectual property
  - Malicious reverse engineers make seek flaws in the cryptographic algorithms or, more commonly, their implementations
    - In 2010 the fail0verflow group recovered Sony's private ECDSA signing key because of a flawed signing process [3] [4]
    - This allowed others to sign alternative software so it would run on the PS3
    - Members of the fail0verflow group faced severe legal penalties, mostly stemming from the way they irresponsibly disclosed their findings

# Why Reverse Engineer Software?

- Security Auditing
  - Device and software vendors often enlist white-hat reverse engineers to identify vulnerabilities and develop proof-of-concept exploits
    - These white-hats use the same tools and techniques as malicious black-hats
  - *Responsibly* disclosing vulnerabilities to vendors is critically important
    - Responsible disclosure allows vendors to patch vulnerabilities to protect consumers
    - Irresponsible disclosure creates opportunities for malicious attackers to develop and deliver exploits
      - This can have severe adverse impacts to consumers, vendors, and other members of the public who may be indirectly affected

# Types of Code

- You will encounter different types of “code” when reverse engineering software
- It is important to be specific about what type of code you are referring to when communicating
- **Source Code**
  - Written in a high-level programming language with no direct association with a specific processor architecture
- **Assembly Code**
  - Written in a low-level programming language directly associated with a specific processor architecture
- **Machine Code**
  - Binary representation of instructions that can be executed directly on a specific processor architecture

# Source Code

- Most modern software is written by human programmers in the form of **source code**
- There are a variety of programming languages available, each with different features suitable for different types of applications (e.g. - C, C++, Java, JavaScript, Python)
- Source code is not associated with specific processor architectures
- Source code is expressed in a text-based form that is readily understood by humans
  - There may be comments that aid in understanding
  - Programming languages have a variety of constructs that are intuitively understood (e.g. loops, conditional statements)

# Assembly Code

- **Assembly code** is written in a lower level programming language associated with a specific processor architecture (e.g. x86, ARM, MIPS)
- Typically, software is written in a source code language and the assembly code is generated by a compiler
- Assembly code does contain elements to make it more understandable to humans
  - Mnemonics to represent lower level machine language opcodes, which are part of instructions
  - Instructions may also include operands
  - Assembly code often includes labels and assembler directives for clarity
    - For example, labels in handwritten assembly code help avoid the need to keep modifying specific references to offsets elsewhere in the code as instructions are added, removed, or modified

# Machine Code

- **Machine code** is a binary representation of instructions to be executed by a processor with a specific architecture (e.g. x86, ARM, MIPS)
- Machine code is not intended to be readily understood by humans
- The translation between assembly code instructions and machine code instructions is 1-to-1
  - The “forward engineering” process involves a translation from assembly code to machine code
- Because there is a 1-to-1 relationship between assembly code instructions and machine code instructions, there are a variety of algorithms that do a reasonable job disassembling machine code (i.e. machine code to assembly code)
  - For example, linear sweep and recursive descent are two primary disassembly algorithms
  - As a reverse engineer, you should understand that these algorithms are not perfect and there are anti-reversing techniques specifically intended to make these translations fail

# Building Software: From Source to Executable

## Preprocessing

## Compilation

## Assembly

## Linking

**Processes directives embedded in the original source code.**

Directives may include contents of source files, conditionally include or exclude blocks of source code, or perform substitutions in the source code.

**Translates preprocessed source code into assembly code.**

Several types of optimizations can be applied (e.g. speed, size, ease of debugging) so this translation from source to assembly code is not 1-to-1.

**Translates assembly code into machine code instructions.**

The result is a relocatable object file that itself is not executable on the target system.

Assembly code is a low level symbolic representation of machine code and this translation from assembly to machine code is 1-to-1

**Combines a set of relocatable object files to produce either an executable object file or a shared object file.**

Executable object files contain programs that can be executed directly. Shared object files can be statically linked with other object files or can be dynamically linked at process load-time or during run-time

# Reversing Software

Preprocessing

Compilation

Assembly

Linking

Decompilation

Disassembly

**Translates assembly code into a source code representation that reasonably resembles the original source code.**

Because of preprocessing and optimization during the forward process, the result of decompilation rarely matches the original source exactly.

**Translates machine code instructions into assembly code.**

The input file likely contains both instructions and data. Different disassemblers and algorithms vary in their ability to correctly distinguish the two in all cases. Reverse engineers can aid the process, using tools, to improve the outcome

# Reverse Engineering Demonstration

hello, world

# Reverse Engineering Demonstration

## ● Forward Process

- Goal
  - Given a simple expression of the program requirements and design, produce an executable program that meets the requirements and is aligned with the design intent
- Approach:
  - The source code is written in C
  - Target system
    - Operating System: Linux
    - Architecture: ARM
      - We use an ARM cross compiler (arm-linux-gnueabi-gcc) to build the software

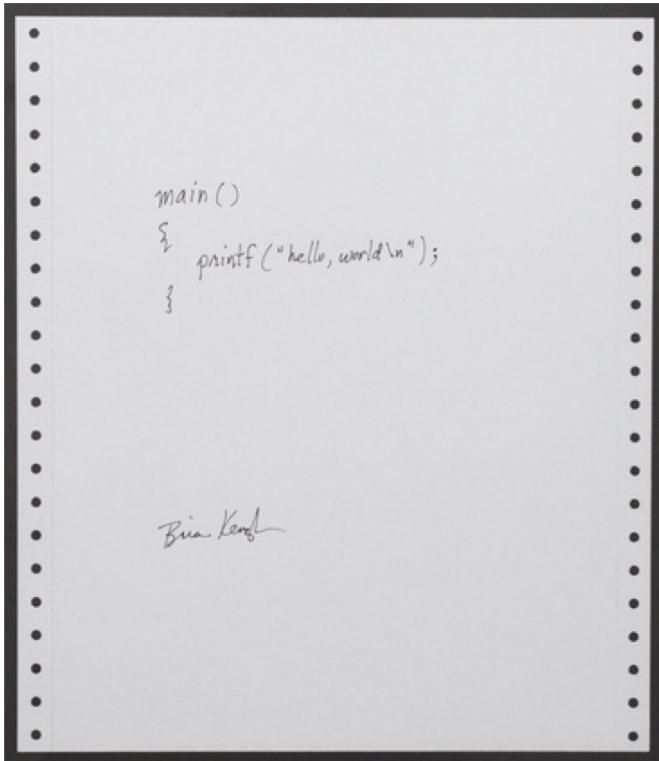
## ● Reverse Process

- Goal:
  - Given only an executable program, understand the original program requirements and design
- Approach:
  - Use Ghidra to reverse the executable program to the point where we have a reasonable representation of the original source code

# Reverse Engineering Demonstration

The Forward Process

# Where to Begin

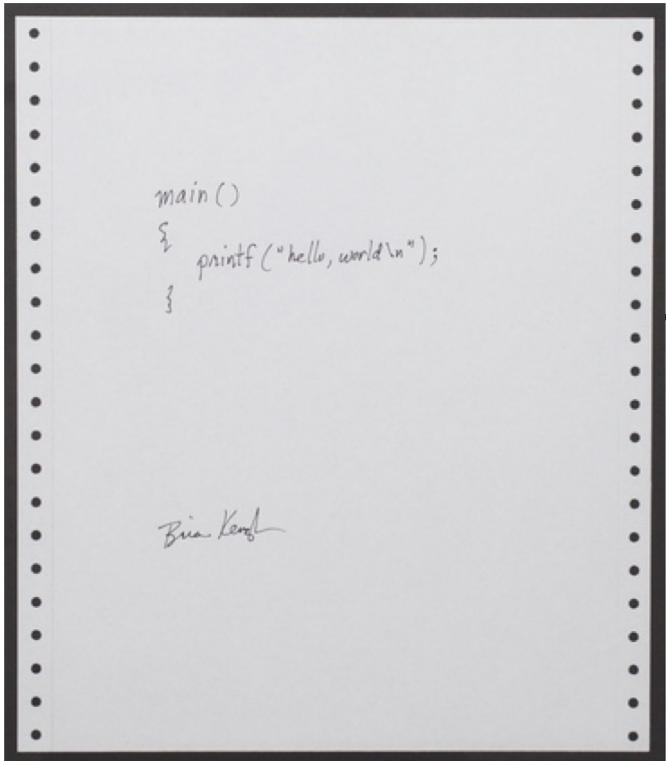


Original Expression of Requirements  
and Design

- For this demonstration, we focus on the forward process steps that span the authoring of source code to the production of an executable program
- Before reaching the point of authoring source code, engineers capture its intended capabilities and design in a variety of artifacts
  - Requirements documents, UML diagrams (use case, sequence, state machine)
- As reverse engineers, we may seek to understand the requirements and design or we may simply want to find flaws in the implementation
  - In either case, these steps we examine here are where we will likely focus most of our efforts

Brian Kernighan, 1978, Public Domain  
[https://commons.wikimedia.org/wiki/File:Hello\\_World\\_Brian\\_Kernighan\\_1978.jpg](https://commons.wikimedia.org/wiki/File:Hello_World_Brian_Kernighan_1978.jpg)

# Writing the Original Source Code



Original Expression of Requirements  
and Design

Initially, programmers translate some expression of the intended design into source code

// This program prints "hello,  
world"

#define NAME "world"

```
int main( ) {
    char * name = NAME;
    printf("hello, %s\n",
name);
}
```

Original Source Code  
(helloworld.c)

Brian Kernighan, 1978, Public Domain

[https://commons.wikimedia.org/wiki/File:Hello\\_World\\_Brian\\_Kernighan\\_1978.jpg](https://commons.wikimedia.org/wiki/File:Hello_World_Brian_Kernighan_1978.jpg)

# Original Source Code Observations

A comment to aid understanding

```
// This program prints "hello,  
world"
```

A preprocessor directive to perform source code substitution prior to compilation

```
#define NAME "world"
```

A local variable

```
int main( ) {  
    char * name = NAME;  
    printf("hello, %s\n",  
           name);  
}
```

A library function call

Original Source Code  
(helloworld.c)

This source code contains elements intended to help understand what the code is doing and aid maintainability

# Preprocessing

Preprocessing

Compilation

Assembly

Linking

```
// This program prints "hello,  
world"
```

```
#define NAME "world"  
  
int main( ) {  
    char * name = NAME;  
    printf("hello, %s\n",  
name);  
}
```

Original Source Code  
(helloworld.c)

The C preprocessor removes  
comments and applies the macros

```
int main( ) {  
    char * name = "world";  
    printf("hello, %s\n", name);  
}
```

Preprocessed Source Code  
(helloworld.i)

```
$ arm-linux-gnueabi-gcc -E helloworld.c -o
```

# Compilation

Preprocessing

Compilation

Assembly

Linking

The compiler translates the preprocessed source code into assembly code

```
int main( ) {  
    char * name = "world";  
    printf("hello, %s\n", name);  
}
```

Preprocessed Source Code  
(helloworld.i)



```
.LC1:  
    .ascii  "hello, %s\012\000"  
.LC0:  
    .ascii  "world\000"  
main:  
    push   {fp, lr}  
    add    fp, sp, #4  
    sub    sp, sp, #8  
    ldr    r3, .L3  
    str    r3, [fp, #-8]
```

<snip>  
Assembler Code  
(helloworld.s)\*  
\* Partially shown

```
$ arm-linux-gnueabi-gcc -S helloworld.i -o
```

# Assembler Code Observations

An assembler directive(begins with a '.')

The '.ascii' directive tells the assembler to replace each character with its ASCII byte value in the output file

A label(ends with a ':')

Labels represent the current location in the output file and can be used as an instruction operand

An ARM assembly instruction

This instruction subtracts the value 8 (decimal) from the value stored in the 'sp' register (stack pointer) and stores the result in the 'sp' register

An ARM assembly instruction with a label reference  
This instruction loads the immediate offset represented by the '.L3' label into the 'r3' register.

.LC1:

.ascii "hello, %s\012\000"

.LC0:

.ascii "world\000"

main:

push {fp, lr}

add fp, sp, #4

sub sp, sp, #8

ldr r3, .L3

str r3, [fp, #-8]

<snip>

Assembler Code

(helloworld.s)\*

\* Partially shown

**This assembler code contains elements intended aid human**

# Assembly

Preprocessing

Compilation

Assembly

Linking

```
.LC1:  
    .ascii  "hello, %s\012\000"  
.LC0:  
    .ascii  "world\000"  
main:  
    push   {fp, lr}  
    add    fp, sp, #4  
    sub    sp, sp, #8  
    ldr    r3, .L3  
    str    r3, [fp, #-8]  
  
<snip>
```

Assembler Code  
(helloworld.s)\*

*\* Partially shown*

Offset	Byte Values (Hexadecimal)
-----	-----
0x0074	68 65 6c 6c 6f 2c 20 25 73
0a 00	
0x006c	77 6f 72 6c 64 00
0x0034	00 48 2d e9
0x0038	04 b0 8d e2
0x003C	08 d0 4d e2
0x0040	1c 30 9f e5
0x0044	08 30 0b e5

Machine Code  
(helloworld.o)\*

*\* Partially shown*

```
$ arm-linux-gnueabi-gcc -c helloworld.s -o
```

# Relocatable Object File Observations

Preprocessing

Compilation

Assembly

Linking

- The assembler translated the **data and assembly code instructions** in the assembler file (helloworld.s) into **data and machine code instructions** in the output file (helloworld.o)
- The output file is a relocatable object file that must be linked with other object files to create either an executable file or a shared object file

```
$ file helloworld.o
helloworld.o: ELF 32-bit LSB relocatable, ARM, EABI5 version 1
(SYSV), not stripped
```

# Relocatable Object File Observations

Preprocessing

Compilation

Assembly

Linking

```
$ hexdump -C helloworld.o
00000000 7f 45 4c 46 01 01 01 01 00 00 00 00 00 00 00 00 |.ELF.....
00000010 01 00 28 00 01 00 00 00 00 00 00 00 00 00 00 00 |..(.....
00000020 58 02 00 00 00 00 00 05 34 00 00 00 00 00 28 00 |X.....4.....
00000030 0c 00 0b 00 00 48 2d e9 04 b0 8d e2 08 d0 4d e2 |....H-.....M.
00000040 10 9f ec 35 08 30 0b e5 08 10 1b e5 14 00 9f e5 |0...0....
00000050 fe ff ff eb 00 30 a0 e3 03 00 a0 e1 04 d0 4b e2 |.....0....K.
00000060 00 88 bd e8 00 00 00 00 08 00 00 00 77 6f 72 6c |.....worl
00000070 64 00 00 00 68 65 6c 6c 6f 2c 20 25 73 0a 00 00 |d..hello, %s..
00000080 47 43 43 3a 20 28 55 62 75 6e 74 75 2f 4c 69 6e |GCC: (Ubuntu/Lin
00000090 61 72 6f 20 37 2e 34 2e 30 2d 31 75 62 75 6e 74 |aro 7.4.0-1ubunt
000000a0 75 31 7e 31 38 2e 30 34 2e 31 29 20 37 2e 34 2e |u1~18.04.1) 7.4.
000000b0 30 00 41 29 00 00 00 61 65 61 62 69 00 01 1f 00 |0.A)...aeabi...
000000c0 00 00 05 35 54 00 06 03 08 01 09 01 12 04 14 01 |...5T.....
000000d0 15 01 17 03 18 01 19 01 1a 02 1e 06 00 00 00 00 |.....
000000e0 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 |.....
000000f0 00 00 00 00 00 00 00 00 04 00 f1 ff 00 00 00 00 |.....
00000100 00 00 00 00 00 00 00 00 03 00 01 00 00 00 00 00 |.....
00000110 00 00 00 00 00 00 00 00 03 00 03 00 00 00 00 00 |.....
00000120 00 00 00 00 00 00 00 00 03 00 04 00 00 00 00 00 |.....
00000130 00 00 00 00 00 00 00 00 03 00 05 00 0e 00 00 00 |.....
00000140 00 00 00 00 00 00 00 00 00 00 05 00 11 00 00 00 |.....
00000150 00 00 00 00 00 00 00 00 00 00 01 00 0e 00 00 00 |.....
00000160 30 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 |0.....
00000170 00 00 00 00 00 00 00 00 03 00 07 00 00 00 00 00 |.....
00000180 00 00 00 00 00 00 00 00 03 00 06 00 00 00 00 00 |.....
00000190 00 00 00 00 00 00 00 00 03 00 08 00 14 00 00 00 |.....
000001a0 00 00 00 00 38 00 00 00 12 00 01 00 19 00 00 00 |....8.....
000001b0 00 00 00 00 00 00 00 00 10 00 00 00 68 65 6c |.....hel
000001c0 6c 6f 77 6f 72 6c 64 2e 63 00 24 64 00 24 61 00 |loworld.c.$d.$a.
000001d0 6d 61 69 6e 00 70 72 69 6e 74 66 00 1c 00 00 00 |main.printf....
000001e0 1c 0d 00 00 30 00 00 00 02 05 00 00 34 00 00 00 |....0.....4...
000001f0 02 05 00 00 00 2e 73 79 6d 74 61 62 00 2e 73 74 |.....symtab.st
00000200 72 74 61 62 00 2e 73 68 73 74 72 74 61 62 00 2e |rtab..shstrtab..
00000210 72 65 6c 2e 74 65 78 74 00 2e 64 61 74 61 00 2e |rel.text..data..
00000220 62 73 73 00 2e 72 6f 64 61 74 61 00 2e 63 6f 6d |bss..rodata..com
00000230 6d 65 6e 74 00 2e 6e 6f 74 65 2e 47 4e 55 2d 73 |ment..note.GNU-s
00000240 74 61 63 6b 00 2e 41 52 4d 2e 61 74 74 72 69 62 |tack..ARM.attrib
00000250 75 74 65 73 00 00 00 00 00 00 00 00 00 00 00 00 |utes.....
```

```
00000260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|*
00000280 1f 00 00 00 00 01 00 00 00 06 00 00 00 00 00 00 |.....|
00000290 34 00 00 00 38 00 00 00 00 00 00 00 00 00 00 00 |4...8.....
000002a0 04 00 00 00 00 00 00 00 00 1b 00 00 00 09 00 00 |.....|
000002b0 40 00 00 00 00 00 00 00 00 dc 01 00 00 18 00 00 |@.....
000002c0 09 00 00 00 01 00 00 00 04 00 00 00 08 00 00 00 |.....|
000002d0 25 00 00 00 01 00 00 00 03 00 00 00 00 00 00 00 |%.....
000002e0 6c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |1.....
000002f0 01 00 00 00 00 00 00 00 2b 00 00 00 08 00 00 00 |.....+.....
00000300 03 00 00 00 00 00 00 00 6c 00 00 00 00 00 00 00 |.....1.....
00000310 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 |.....|
00000320 30 00 00 00 01 00 00 00 02 00 00 00 00 00 00 00 |0.....
00000330 6c 00 00 00 13 00 00 00 00 00 00 00 00 00 00 00 |1.....
00000340 04 00 00 00 00 00 00 00 38 00 00 00 01 00 00 00 |.....8.....
00000350 30 00 00 00 00 00 00 00 7f 00 00 00 33 00 00 00 |0.....3...
00000360 00 00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 |.....|
00000370 41 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 |A.....
00000380 b2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000390 01 00 00 00 00 00 00 00 51 00 00 00 03 00 00 70 |.....Q....p
000003a0 00 00 00 00 00 00 00 00 b2 00 00 00 2a 00 00 00 |.....*.....
000003b0 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 |.....|
000003c0 01 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 |.....|
000003d0 dc 00 00 00 e0 00 00 00 0a 00 00 00 0c 00 00 00 |.....|
000003e0 04 00 00 00 10 00 00 00 09 00 00 00 03 00 00 00 |.....|
000003f0 00 00 00 00 00 00 00 00 bc 01 00 00 20 00 00 00 |.....|
00000400 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 |.....|
00000410 11 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000420 f4 01 00 00 61 00 00 00 00 00 00 00 00 00 00 00 |.....a.....
00000430 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|24
00000438
```

# Linking

Preprocessing

Compilation

Assembly

Linking

```
$ readelf -h helloworld.o
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: REL (Relocatable file)
  Machine: ARM
  Version: 0x1
  Entry point address: 0x0
  Start of program headers: 0 (bytes into file)
  Start of section headers: 600 (bytes into file)
  Flags: 0x50000000, Version5 EABI
  Size of this header: 52 (bytes)
  Size of program headers: 0 (bytes)
  Number of program headers: 0
  Size of section headers: 40 (bytes)
  Number of section headers: 12
  Section header string table index: 11
```

Machine Code  
(helloworld.o)\*  
*\* Partially shown*

```
$ readelf -h helloworld
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: ARM
  Version: 0x1
  Entry point address: 0x10310
  Start of program headers: 52 (bytes into file)
  Start of section headers: 6884 (bytes into file)
  Flags: 0x5000200, Version5 EABI, soft-float ABI
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 9
  Size of section headers: 40 (bytes)
  Number of section headers: 29
  Section header string table index: 28
```

Machine Code  
(helloworld)\*  
*\* Partially shown*

```
$ arm-linux-gnueabi-gcc helloworld.o -o helloworld
```

# Executable File Observations

Preprocessing

Compilation

Assembly

Linking

- The linker linked our relocatable object file with other object files to create an ELF executable file

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
linked, interpreter /lib/ld-, for GNU/Linux 3.2.0,
BuildID[sha1]=0c4102eecea9b56c0a57e3c4cdb2d6c2667bc8a4, not stripped
```

- Because we built the executable for the ARM architecture we need to use the qemu-arm emulator to run it on our x86\_64 system

```
$ qemu-arm -L /usr/arm-linux-gnueabi helloworld
hello, world
$
```

# Reverse Engineering Demonstration

The Reverse Process

# Where to Begin

- We may only have a piece of executable software to begin our reverse engineering efforts
  - This is likely to be the case for a malware sample
- If we are reverse engineering an IoT device, we may have a fully running system with associated configuration files
- For this demonstration, we focus reversing the helloworld executable we just built to see how close we can get to the original source code
  - Often we aren't able to validate our results because we don't actually have the source code
  - We can probably gain some confidence in our conclusions with empirical evidence
- Reversing is a very broad domain and here we focus specifically on the software code
  - For example, we may also reverse engineer protocols or system configurations

# Disassembly

Decompilation

Disassembly

We load our helloworld ELF executable into Ghidra and get this result in the Listing view

Shown here is the main() function

Ghidra's Disassembly (helloworld)

Listing: helloworld [CodeBrowser: hello\_world;/helloworld]

File Edit Analysis Navigation Search Select Tools Help

I D U L F T V B

Listing: helloworld

\* FUNCTION \*

undefined main()

    r0:1 <RETURN>

    Stack[-0xc]:4 local\_c

XREF[2]: 00010410(W), 00010414(R)

main

XREF[3]: Entry Point(\*), \_start:00010330(\*), 00010344(\*)

00010400 00 48 2d e9 stmdb sp!,{ r11 lr }

00010404 04 b0 8d e2 add r11,sp,#0x4

00010408 08 d0 4d e2 sub sp,sp,#0x8

0001040c 1c 30 9f e5 ldr r3=>s\_world\_000104a8,[PTR\_s\_world\_00010430] = "world"

00010410 08 30 0b e5 str r3=>s\_world\_000104a8,[r11,#local\_c] = "world"

00010414 08 10 1b e5 ldr r1=>s\_world\_000104a8,[r11,#local\_c] = "world"

00010418 14 00 9f e5 ldr r0=>s\_hello,\_%s\_000104b0,[PTR\_s\_hello,\_%s\_0001... = 000104b0 = "hello, %s\n"

0001041c af ff ff eb bl printf int printf(char \* \_\_format, ...)

00010420 00 30 a0 e3 mov r3,#0x0

00010424 03 00 a0 e1 cpq r0,r3

00010428 04 d0 4b e2 sub sp,r11,#0x4

0001042c 00 88 bd e8 ldmia sp!,{ r11 pc }

PTR\_s\_world\_00010430 XREF[1]: main:0001040c(R) = "world"

00010430 a8 04 01 00 addr s\_world\_000104a8

PTR\_s\_hello,\_%s\_00010434 XREF[1]: main:00010418(R) = "hello, %s\n"

00010434 b0 04 01 00 addr s\_hello,\_%s\_000104b0

29

# Disassembly

Decompilation

Disassembly

ARM Machine Code  
Instructions and  
Data

Directly from the  
helloworld ELF  
executable

The screenshot shows the Ghidra interface with the title "Listing: helloworld [CodeBrowser: hello\_world:helloworld]". The menu bar includes File, Edit, Analysis, Navigation, Search, Select, Tools, and Help. The toolbar has icons for file operations and analysis. The main window displays the assembly listing for the "main" function. The assembly code is color-coded, and comments explain the C code it corresponds to. A callout box highlights the printf instruction at address 00010418.

```
***** FUNCTION *****
undefined main()
    r0:1 <RETURN>
    Stack[-0xc]:4 local_c
    sp!,{ r11 lr }
    r11,sp,#0x4
    sp,sp,#0x8
    r3=>s_world_000104a8,[PTR_s_world_00010430] = "world"
    r3=>s_world_000104a8,[r11,#local_c] = 000104a8
    r1=>s_world_000104a8,[r11,#local_c] = "world"
    r0=>s_hello,_%s_000104b0,[PTR_s_hello,_%s_000104b0] = 000104b0
    int printf(char * __format, ...)

    stmdb sp!,{ r11 lr }
    add r11,sp,#0x4
    sub sp,sp,#0x8
    ldr r3,[r3]
    ldr r1,[r1]
    ldr r0,[r0]

    str r3,[r3]
    ldr r1,[r1]
    ldr r0,[r0]

    bl printf
    mov r3,#0x0
    mov r0,r3
    sub sp,r11,#0x4
    ldmia sp!,{ r11 pc }

    PTR_s_world_00010430 XREF[1]: main:000104c(R) = "world"
    PTR_s_world_000104a8 XREF[1]: main:00010418(R) = "hello, %s\n"

    add r3,[r3]
    add r1,[r1]
    add r0,[r0]

    PTR_s_hello,_%s_00010434 XREF[1]: main:00010418(R) = "hello, %s\n"
    PTR_s_hello,_%s_000104b0 XREF[1]: main:00010418(R) = "hello, %s\n"
```

# Disassembly

Decompilation

Disassembly

Memory Offsets

```
Listing: helloworld [CodeBrowser: hello_world:/helloworld]
File Edit Analysis Navigation Search Select Tools Help
I D U L F R V B
Listing: helloworld
helloworld x

***** FUNCTION *****
*
undefined undefined4
undefined main()
    r0:1             <RETURN>
    Stack[-0xc]:4 local_c

main
00010400 00 48 2d e9  stmdb   sp!,{ r11 lr }
00010404 04 b0 8d e2  add     r11,sp,#0x4
00010408 08 d0 4d e2  sub    sp,sp,#0x8
0001040c 1c 30 9f e5  ldr    r3=>s_world_000104a8,[PTR_s_world_00010430] = "world"
00010410 08 30 0b e5  str    r3=>s_world_000104a8,[r11,#local_c] = "world"
00010414 08 10 1b e5  ldr    r11=>s_world_000104a8,[r11,#local_c] = "world"
00010418 14 00 9f e5  ldr    r0=>s_hello,_%s_000104b0,[PTR_s_hello,_%s_000104b0] = 000104b0
0001041c af ff ff eb  bl     printf = "hello, %s\n"
00010420 00 30 a0 e3  mov    r3,#0x0
00010424 08 00 a0 e1  cpy    r0,r3
00010428 04 d0 4b e2  sub    sp,r11,#0x4
0001042c 00 88 bd e8  ldmia  sp!,{ r11 pc }

PTR_s_world_00010430
00010430 a8 04 01 00  addr   s_world_000104a8 XREF[1]: main:0001040c(R) = "world"

PTR_s_hello,_%s_00010434
00010434 b0 04 01 00  addr   s_hello,_%s_000104b0 XREF[1]: main:00010418(R) = "hello, %s\n"

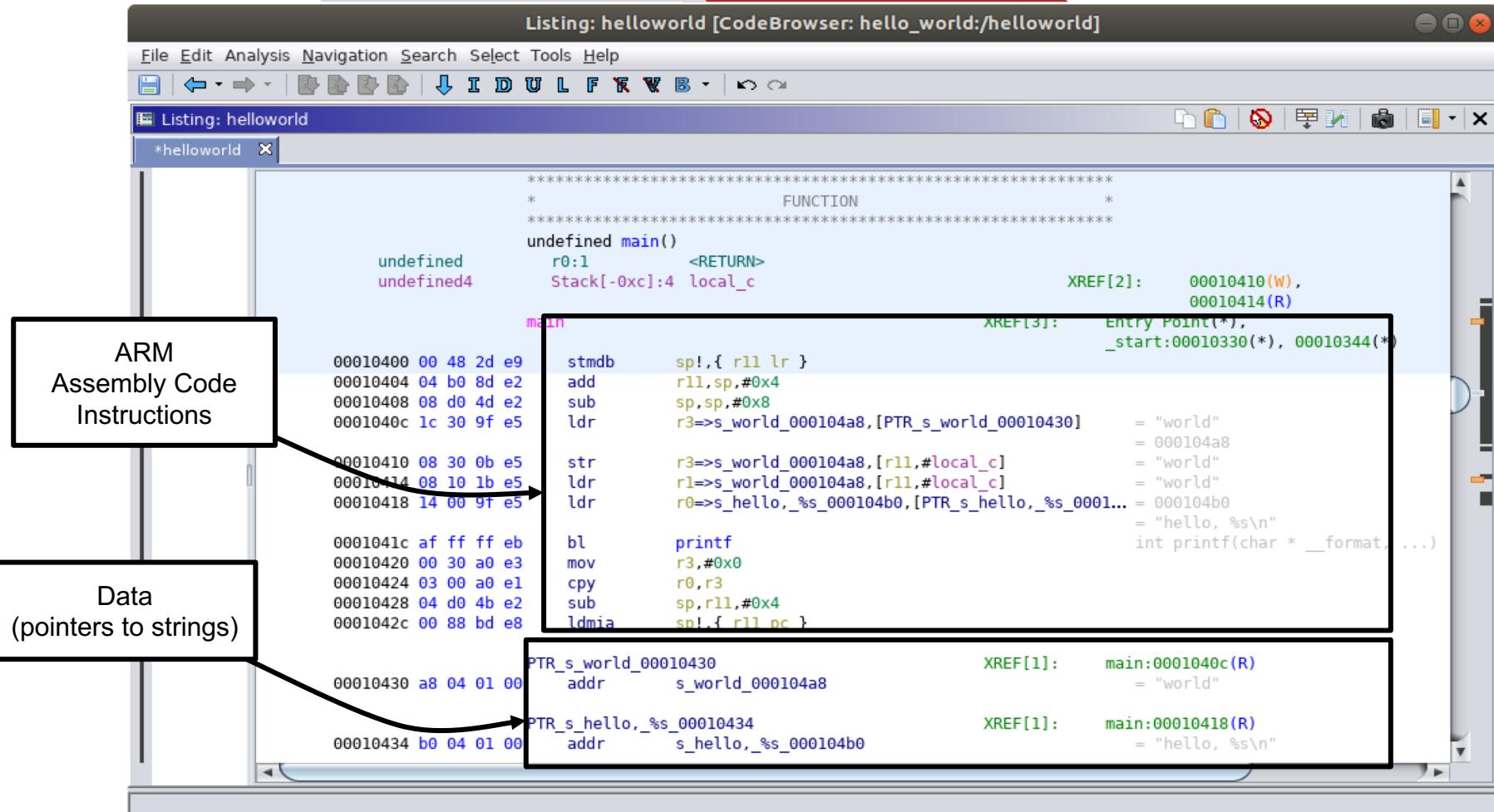
***** XREFS *****
XREF[2]: 00010410(W), 00010414(R)
XREF[3]: Entry Point(*), _start:00010330(*), 00010344(*)
```

Ghidra's Disassembly  
(helloworld)

# Disassembly

Decompilation

Disassembly



Ghidra's Disassembly  
(helloworld)

# Disassembly Observations

```
push    {fp, lr}
add     fp, sp, #4
sub     sp, sp, #8
ldr     r3, .L3
str     r3, [fp, #-8]
ldr     r1, [fp, #-8]
ldr     r0, .L3+4
bl      printf
mov     r3, #0
mov     r0, r3
sub     sp, fp, #4
pop    {fp, pc}
```

Assembler Code  
(helloworld.s)\*

\* Only the assembly code for the main function is shown

In reality, we wouldn't have access to the original assembler code (helloworld.s), but we can see Ghidra's disassembly of the main() function is equivalent to the original assembler code from the forward process demonstration

```
stmdb   sp!,{ r11 lr }
add    r11,sp,#0x4
sub    sp,sp,#0x8
ldr    r3=>s_world_000104a8,[PTR_s_world_00010430]      = "world"
       = 000104a8
str    r3=>s_world_000104a8,[r11,#local_c]           = "world"
ldr    r1=>s_world_000104a8,[r11,#local_c]           = "world"
ldr    r0=>s_hello,_%s_000104b0,[PTR_s_hello,_%s_0001... = 000104b0
       = "hello, %s\n"
bl     printf
       int printf(char * __format, ...)
mov    r3,#0x0
cpy    r0,r3
sub    sp,r11,#0x4
ldmia sp!,{ r11 pc }
```

Ghidra's Disassembly  
(helloworld)\*

\* Only the (dis)assembly code for the main function is shown

Note:

- “`stmdb sp!,{ r11 lr }`” is synonymous with “`push {fp, lr}`”
- “`ldmia sp!,{ r11 pc }`” is synonymous with “`pop {fp, pc}`”

# Disassembly Analysis

- We can analyze the disassembly to begin to understand how the program works
- In the analysis that follows we don't follow the code "in order"
- We begin by identifying the most recognizable constructs first
- As we progress, we examine the more application-specific portions of our target sample

# Disassembly of the main() Function

## Identifying the function prologue and epilogue

Function Prologue

```
stmdb    sp!,{ r11 lr }
add      r11,sp,#0x4
sub      sp,sp,#0x8
ldr      r3=>s_world_000104a8,[PTR_s_world_00010430]      = "world"
         = 000104a8
str      r3=>s_world_000104a8,[r11,#local_c]           = "world"
ldr      r1=>s_world_000104a8,[r11,#local_c]           = "world"
ldr      r0=>s_hello,%s_000104b0,[PTR_s_hello,%s_0001... = 000104b0
         = "hello, %s\n"
bl       printf                                         int printf(char * __format, ...)
mov      r3,#0x0
cpy      r0,r3
sub      sp,r11,#0x4
ldmia   sp!,{ r11 pc }
```

Function Epilogue

# Disassembly of the main() Function

## Populating the return value

```
stmdb    sp!,{ r11 lr }
add      r11,sp,#0x4
sub      sp,sp,#0x8
ldr      r3=>s_world_000104a8,[PTR_s_world_00010430]      = "world"
         = 000104a8
str      r3=>s_world_000104a8,[r11,#local_c]          = "world"
ldr      r1=>s_world_000104a8,[r11,#local_c]          = "world"
ldr      r0=>s_hello,_%s_000104b0,[PTR_s_hello,_%s_0001... = 000104b0
         = "hello, %s\n"
bl       printf                                         int printf(char * __format, ...)
mov      r3,#0x0
cpy      r0,r3
sub      sp,r11,#0x4
ldmia   sp!,{ r11 pc }
```

Moving the value '0' into the 'r0' register  
The 'r0' register holds the function return value

# Disassembly of the main() Function

Storing a pointer to the C string 'world' on the stack

## Setting a local variable

```
stmdb    sp!,{ r11 lr }
add      r11,sp,#0x4
sub      sp,sp,#0x8
ldr      r3=>s_world_000104a8,[PTR_s_world_00010430]      = "world"
         = 000104a8
str      r3=>s_world_000104a8,[r11,#local_c]           = "world"
ldr      r1=>s_world_000104a8,[r11,#local_c]           = "world"
ldr      r0=>s_hello,_%s_000104b0,[PTR_s_hello,_%s_0001... = 000104b0
         = "hello, %s\n"
bl       printf
int printf(char * _format, ...)
mov      r3,#0x0
cpy      r0,r3
sub      sp,r11,#0x4
ldmia   sp!,{ r11 pc }
```

# Disassembly of the main() Function

## Calling a function

```
stmdb    sp!,{ r11 lr }
add      r11,sp,#0x4
sub      sp,sp,#0x8
ldr      r3=>s_world_000104a8,[PTR_s_world_00010430]      = "world"
         = 000104a8
str      r3=>s_world_000104a8,[r11,#local_c]           = "world"
ldr      r1=>s_world_000104a8,[r11,#local_c]           = "world"
ldr      r0=>s_hello,_%s_000104b0,[PTR_s_hello,_%s_0001... = 000104b0
         = "hello, %s\n"
bl       printf                                         int printf(char * __format, ...)
mov      r3,#0x0
cpy      r0,r3
sub      sp,r11,#0x4
ldmia   sp!,{ r11 pc }
```

Load a pointer to the C string 'world' into register 'r1'

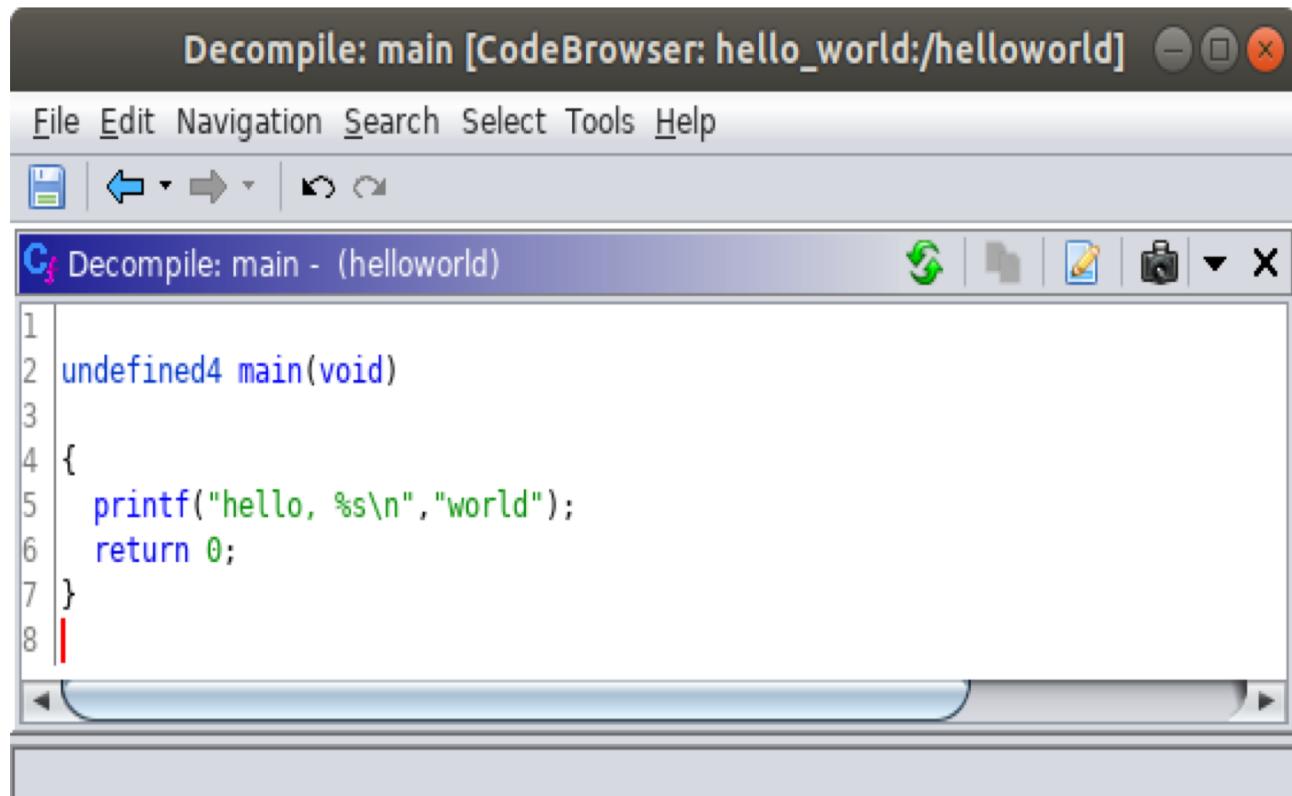
Load a pointer to the C string 'hello. %s\n' into register 'r0'

Call the 'printf' function with the first argument held in register 'r0' and  
the second argument stored in register 'r1'

# Decompilation

Decompilation

Disassembly



Ghidra also provides a This is Ghidra's decompilation of the main() function

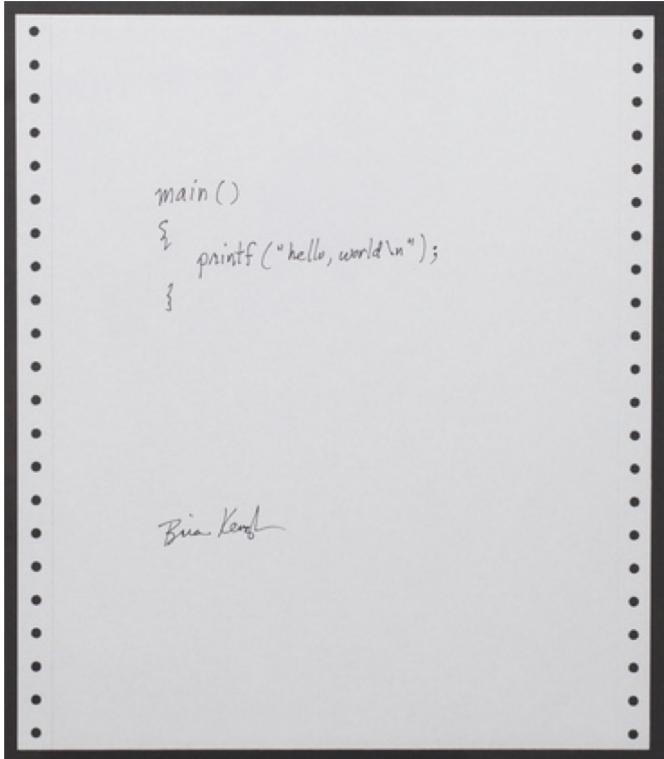
The screenshot shows the Ghidra interface with the title bar "Decompile: main [CodeBrowser: hello\_world:/helloworld]". The menu bar includes File, Edit, Navigation, Search, Select, Tools, and Help. Below the menu is a toolbar with standard file operations like Save, Open, and Close. The main window displays the decompiled code for the main function:

```
1
2 undefined4 main(void)
3
4 {
5     printf("hello, %s\n", "world");
6     return 0;
7 }
8
```

The code is color-coded: "undefined4" and "main" are blue, while the printf call and its argument are green. A red cursor is positioned at the end of the closing brace on line 7.

Ghidra's Decompilation of the main function

# Decompilation Observations



Original Expression of Requirements  
and Design

In this case, Ghidra did a reasonable job creating a higher level representation of the original source code from the helloworld ELF executable.

```
// This program prints "hello, world"

#define NAME "world"

int main( ) {
    char * name = NAME;
    printf("hello, %s\n", name);
}
```

Original Source Code  
(helloworld.c)

```
Decompile: main [CodeBrowser: hello_world:/helloworld]
File Edit Navigation Search Select Tools Help
Decompile: main - (helloworld)
1 undefined4 main(void)
2
3 {
4     printf("hello, %s\n","world");
5     return 0;
6 }
7
8
```

Ghidra's Decompilation of the main function

Brian Kernighan, 1978, Public Domain  
[https://commons.wikimedia.org/wiki/File:Hello\\_World\\_Brian\\_Kernighan\\_1978.jpg](https://commons.wikimedia.org/wiki/File:Hello_World_Brian_Kernighan_1978.jpg)

# Examining Ghidra

# A Deeper Dive into the Ghidra Tool

- Ghidra is a software reverse engineering framework first released by the National Security Agency (NSA) in March, 2019. Primary capabilities include:
- Disassembly
  - Results are highly annotated
  - Functions identified
  - Arguments and variables are identified
  - Cross references
  - Strings are shown
- Decompilation
  - Provides a C language representation of the program
  - Reconstructs logical and mathematical expressions
  - Identifies arguments, variables, and function calls

# Other Ghidra Features

- Several integrated views of the program
  - Navigate easily between disassembly, decompilation, function listing, strings listing, memory map, etc.
- Edit function signatures, argument and variable names
- Add comments and annotations during manual analysis
- Rich filtering interfaces
- Scripting support to extend capabilities and automate tasks

# Ghidra Supported Architectures

- Ghidra supports a variety of architectures including:
  - X86
  - ARM
  - PowerPC
  - MIPS
  - Java bytecode

# Ghidra Definitions

- **Program**

- A unit of software or data that can be viewed and analyzed within Ghidra
- Supported formats include
  - Executable and Linkable Format (ELF)
  - Portable Executable (PE)
  - Raw binary files

- **Project**

- A collection programs and associated configuration information

- **Plugin**

- A building block that adds a unit of functionality to Ghidra.
- Can be combined with other plugins to create higher level functionality
- Ghidra comes with a set of plugins, but more can be added

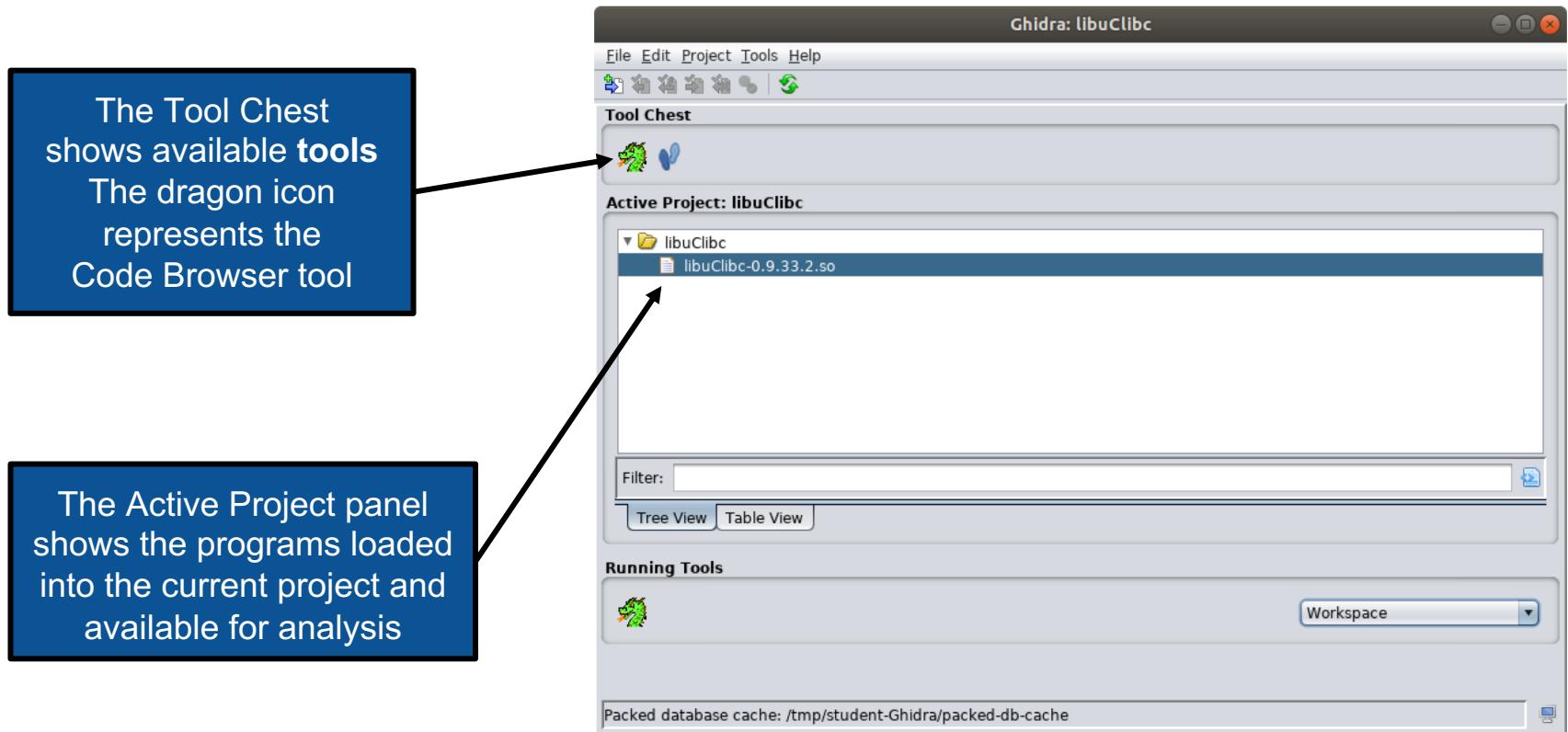
- **Tool**

- A collection of plugins that together provide a GUI for performing reverse engineering tasks
- Ghidra's Code Browser tool is available by default and has all the Core plugins included
- Additional tools can be defined

Refer to the glossary in the Ghidra Help for additional definitions

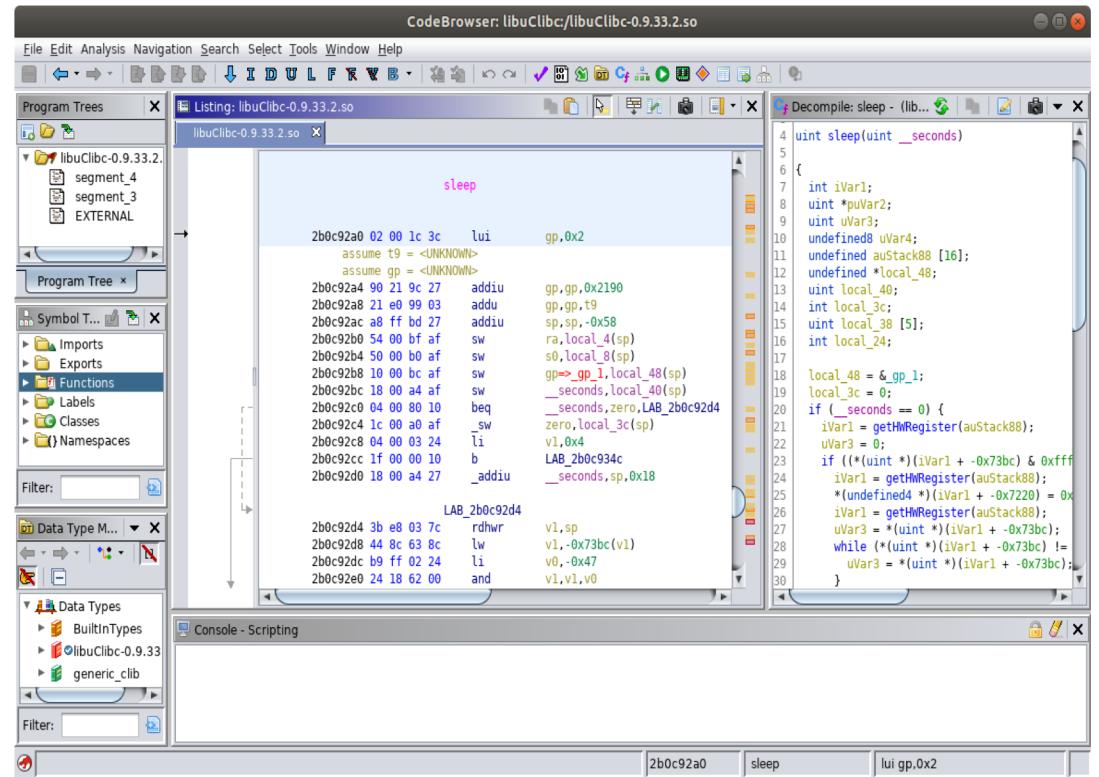
# Ghidra Project Window

- The first window you encounter after launching Ghidra is the Project Window
- The Project Window is the interface for creating and managing **projects**

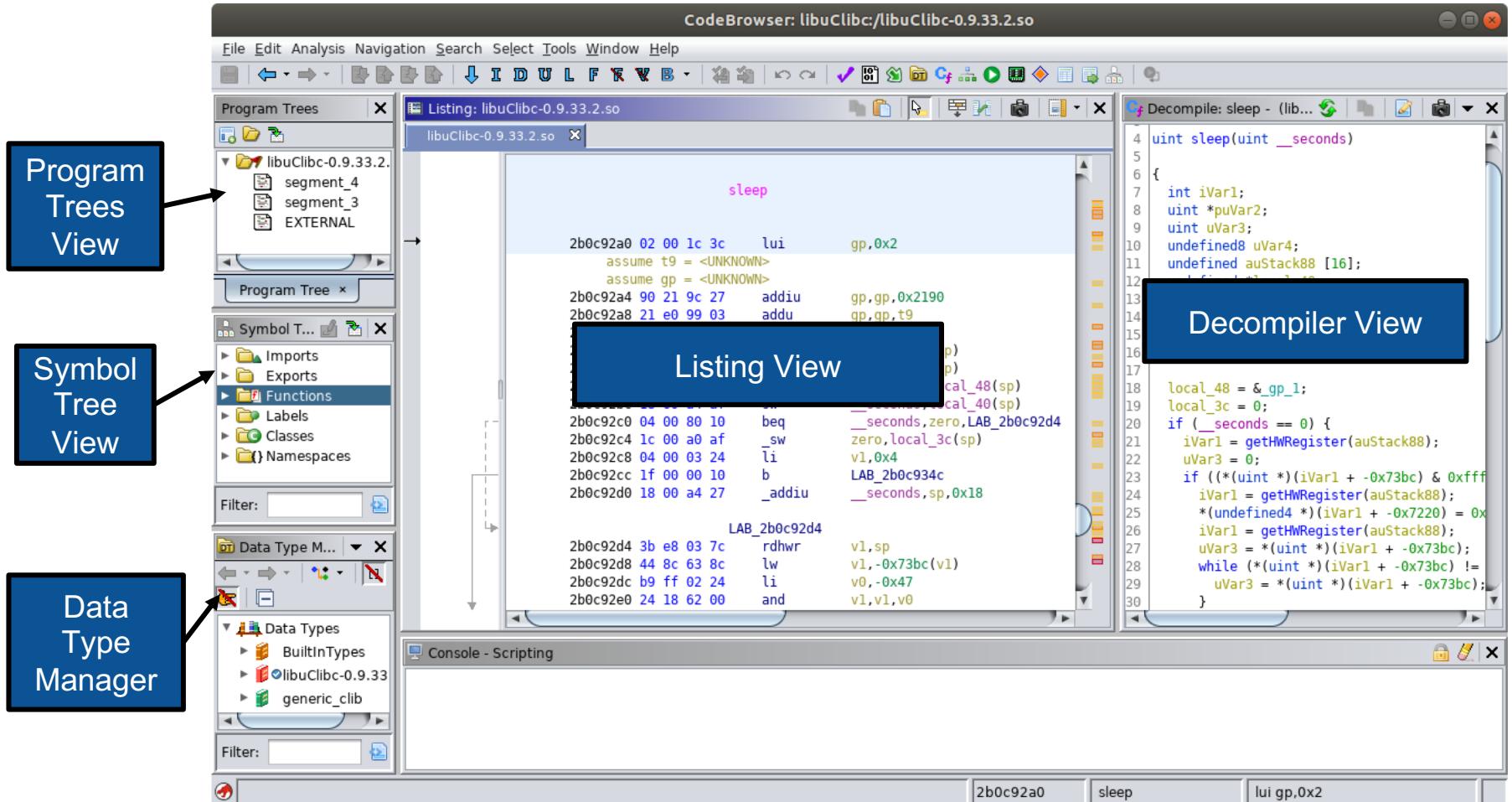


# Code Browser Tool

- The Code Browser tool is the primary GUI interface you will use
- It contains a variety of visible views by default
  - Views can be moved by clicking the title bar then dragging the tool
  - Views can be moved within the main Code Browser window or fully detached
- Additional views can be added by selecting them from the Window menu



# Code Browser Tool: Default Views



# Listing View

- The main window for examining a program's instructions and data
- Includes address offsets, machine code instructions, assembly code instructions
- Branching and function calls are illustrated with the arrows on the left
- Annotations for symbol names, cross references, and strings are included

```
Listing: libuClibc-0.9.33.2.so [CodeBrowser: libuClibc;/libuClibc-0.9.33.2.so]
File Edit Analysis Navigation Search Select Tools Help
I D U L F R V B
Listing: libuClibc-0.9.33.2.so | libuClibc-0.9.33.2.so x
2b0c930c 10 00 43 34 ori v1,v0,0x10
2b0c9310 00 00 a6 c0 ll a2,0x0(a1)
2b0c9314 06 00 c2 14 bne a2,v0,LAB_2b0c9330
2b0c9318 21 38 00 00 _clear a3
2b0c931c 21 38 60 00 move a3,v1
2b0c9320 00 00 a7 e0 sc a3,0x0(a1)
2b0c9324 fa ff e0 10 beq a3,zero,LAB_2b0c9310
2b0c9328 00 00 00 00 _nop a3,zero,LAB_2b0c9310
2b0c932c 0f 00 00 00 sync 0x0
XREF[1]: 2b0c9324(j)

LAB_2b0c9330
2b0c9330 f6 ff e0 50 beql a3,zero,LAB_2b0c930c
2b0c9334 84 00 82 8c _lw v0,0x84(__seconds)
2b0c9338 f8 88 99 8f _lw t9,-0x7708(gp)=>>FUN_2b0c9fc4
2b0c933c 09 f8 20 03 jalr t9>>FUN_2b0c9fc4
2b0c9340 80 00 84 8c _lw __seconds,0x80(__seconds)
XREF[1]: 2b0c9314(j) = 2b0c9fc4
undefined FUN_2b0c9fc4()

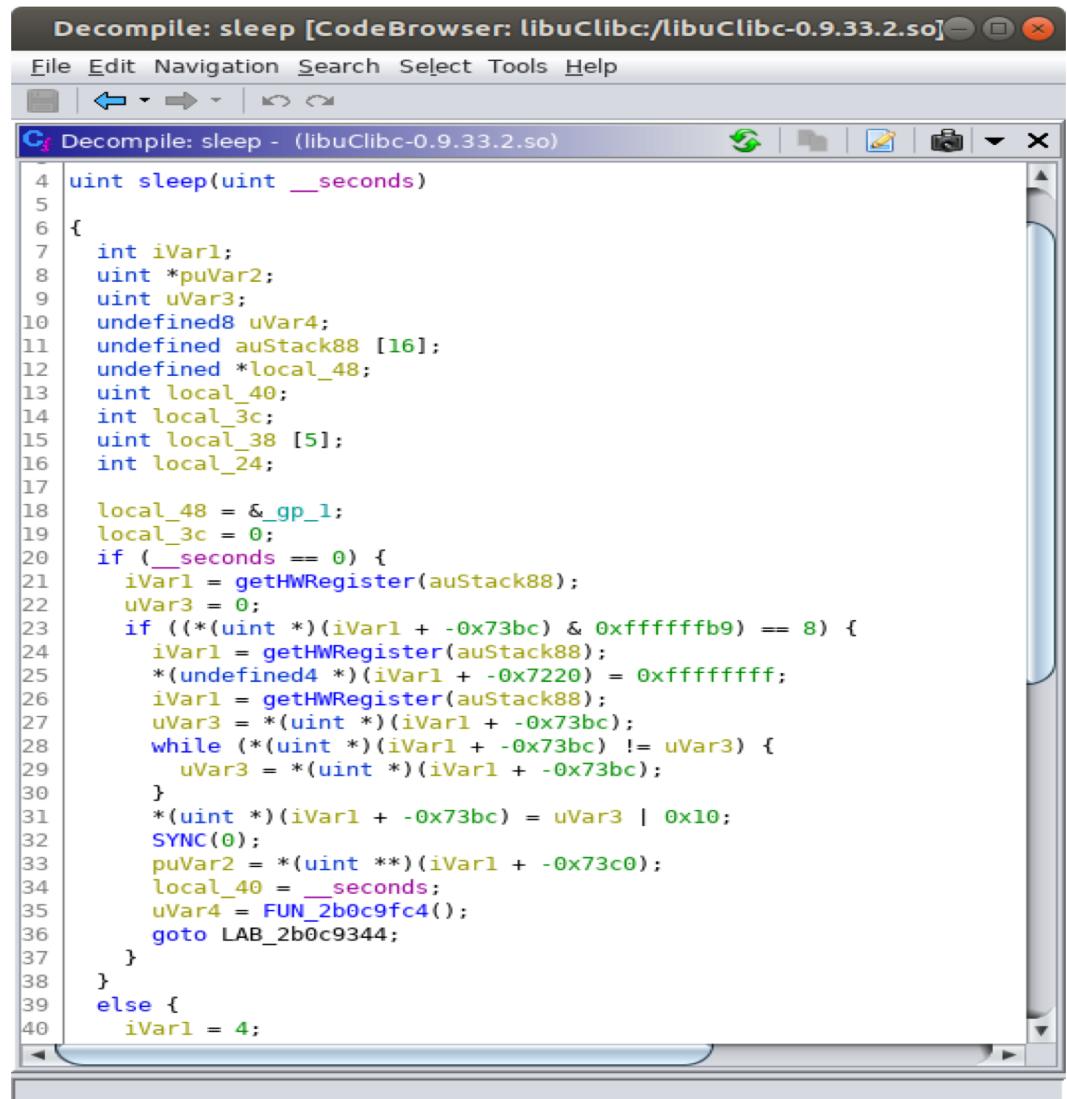
LAB_2b0c9344
2b0c9344 21 10 82 00 addu v0,__seconds,v0
2b0c9348 08 00 40 ac sw zero,local_2c(v0)
XREF[1]: 2b0c9350(j)

LAB_2b0c934c
2b0c934c ff ff 63 24 addiu v1,v1,-0x1
2b0c9350 fc ff 61 04 bgez v1,LAB_2b0c9344
2b0c9354 80 10 03 00 _sll v0,v1,0x2
2b0c9358 20 00 a2 8f _lw v0,local_38(sp)
2b0c935c 02 00 03 3c lui v1,0x2
2b0c9360 25 10 43 00 or v0,v0,v1
2b0c9364 58 83 99 8f _lw t9,-0x7ca8(gp)=>>sigaction
2b0c9368 12 00 04 24 li __seconds,0x12
2b0c936c 21 28 00 00 clear al
2b0c9370 30 00 a6 27 addiu a2,sp,0x30
2b0c9374 09 f8 20 03 ialr t9=>sigaction
XREF[1]: 2b0c92cc(j) = 2b0746f0

int sigaction(int sig, si...
```

# Decompiler View

- A C language representation of the active function selected in the Listing View
- The user can rename arguments and variables and change data types
- Selections and edits are dynamically synced between the Listing View and Decompiler View

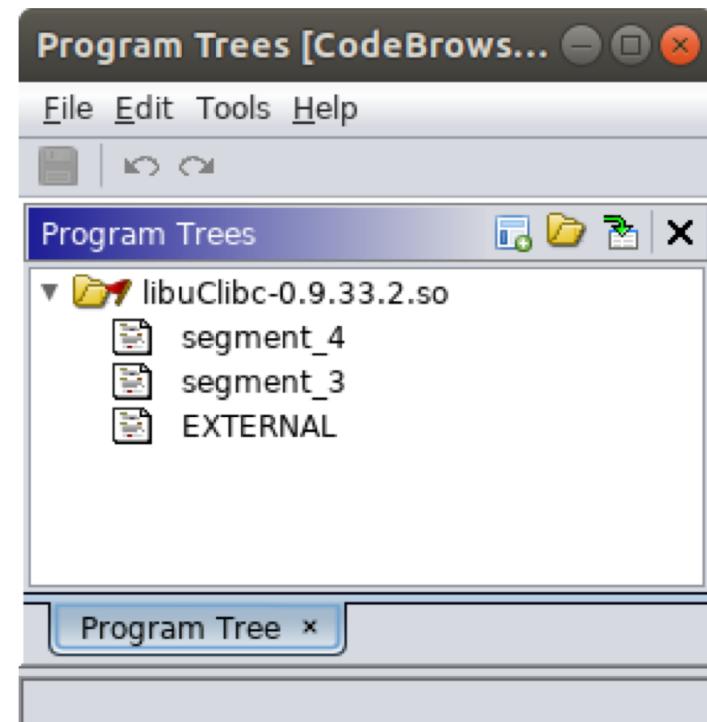


The screenshot shows a software interface titled "Decompile: sleep [CodeBrowser: libuClibc:/libuClibc-0.9.33.2.so]". The window has a menu bar with File, Edit, Navigation, Search, Select, Tools, and Help. Below the menu is a toolbar with icons for back, forward, search, and other functions. The main area displays the decompiled C code for the sleep function. The code uses color-coded syntax highlighting for different data types and identifiers.

```
4 uint sleep(uint __seconds)
5
6 {
7     int iVar1;
8     uint *puVar2;
9     uint uVar3;
10    undefined8 uVar4;
11    undefined auStack88 [16];
12    undefined *local_48;
13    uint local_40;
14    int local_3c;
15    uint local_38 [5];
16    int local_24;
17
18    local_48 = &_gp_1;
19    local_3c = 0;
20    if (__seconds == 0) {
21        iVar1 = getHWRegister(auStack88);
22        uVar3 = 0;
23        if (((uint *)(&iVar1 + -0x73bc) & 0xfffffffffb9) == 8) {
24            iVar1 = getHWRegister(auStack88);
25            *(undefined4 *)(&iVar1 + -0x7220) = 0xffffffff;
26            iVar1 = getHWRegister(auStack88);
27            uVar3 = *(uint *)(&iVar1 + -0x73bc);
28            while (((uint *)(&iVar1 + -0x73bc) != uVar3) {
29                uVar3 = *(uint *)(&iVar1 + -0x73bc);
30            }
31            *(uint *)(&iVar1 + -0x73bc) = uVar3 | 0x10;
32            SYNC(0);
33            puVar2 = *(uint **)(iVar1 + -0x73c0);
34            local_40 = __seconds;
35            uVar4 = FUN_2b0c9fc4();
36            goto LAB_2b0c9344;
37        }
38    }
39    else {
40        iVar1 = 4;
```

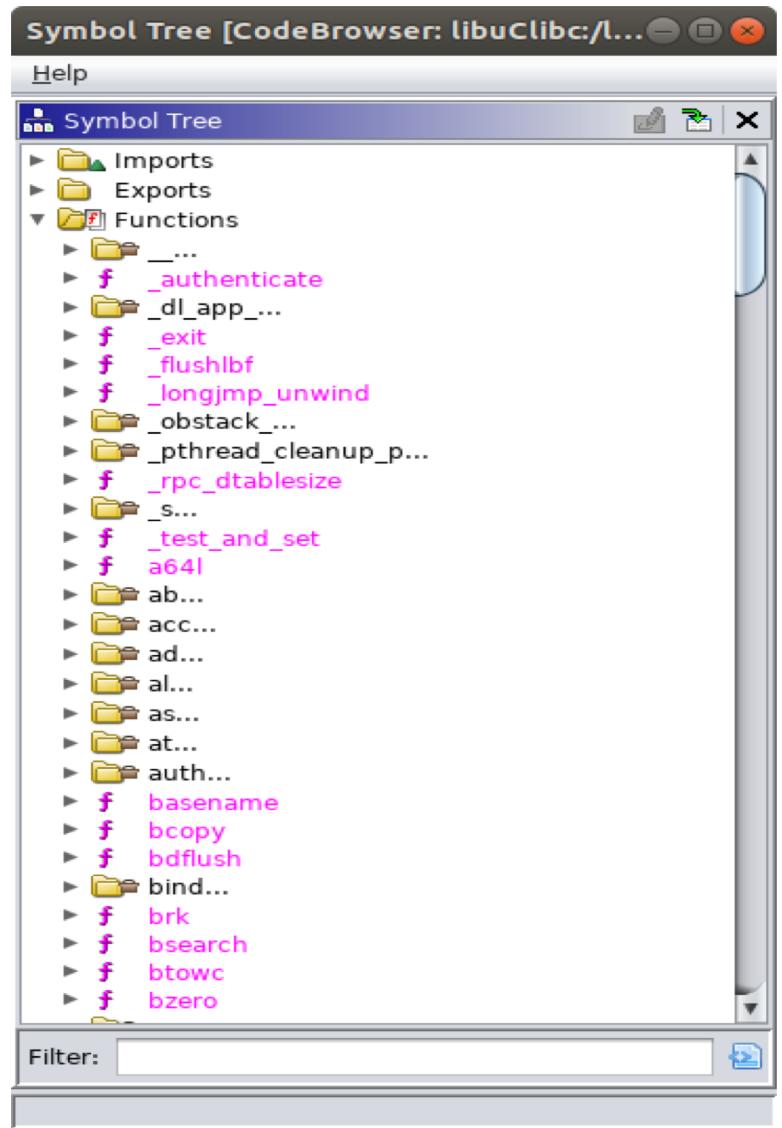
# Program Trees View

- Program Trees may be used to organize sections of a program
- Portions of the program can be selected and associated with a “fragment”
  - Some fragments are automatically created based on segments defined in the program
  - New fragments can be defined and code units can be associated with different fragments
  - Code units can only be associated with one fragment
- Fragments can be organized into folders
  - Folders can also be nested within other folders



# Symbol Tree View

- Shows a hierarchical view of symbols in a program
  - Imports
  - Exports
  - Functions
  - Labels
  - Classes
  - Namespaces
- Clicking on a symbol in the Symbol Tree navigates to where that symbol appears in the Listing View
- The Symbol Tree view includes a filter field that is helpful for quickly finding a symbol and navigating to it



# Defined Data View

- Not shown by default, but can be added by selecting it from the Window menu in the Code Browser
- Shows data (e.g. strings, integers) defined within the program
  - e.g. Global or local variables
- This view also contains a filtering capability
- Can be especially helpful for reverse engineering programs, especially when some of this data appears as program output and you want to examine the code around it

Defined Data [CodeBrowser: libuClibc:/libuClibc-0.9.33.2.so]

Help

Data	Location	Type	Size
FFFDB9A8h	2b0d14e4	uint	4
FFFDB9A8h	2b0d14e8	uint	4
FFFDB9A8h	2b0d14ec	uint	4
FFFDB9A8h	2b0d14f0	uint	4
FFFDB9A8h	2b0d14f4	uint	4
FFFDB9A8h	2b0d14f8	uint	4
"LOGNAME"	2b0d1500	string	8
"POSIXLY_CORRECT"	2b0d1508	string	16
"%s: option `%-s' is ambiguous\n"	2b0d151c	string	30
"%s: option `--%-s' doesn't allow an argument\n"	2b0d153c	string	45
"%s: option `%-c%s' doesn't allow an argument\n"	2b0d156c	string	45
"%s: option `%-s' requires an argument\n"	2b0d159c	string	38
"%s: unrecognized option `--%-s'\n"	2b0d15c4	string	32
"%s: unrecognized option `%-c%-s'\n"	2b0d15e4	string	32
"%s: illegal option -- %c\n"	2b0d1604	string	26
"%s: invalid option -- %c\n"	2b0d1620	string	26
"%s: option requires an argument -- %c\n"	2b0d163c	string	39
FFFDCA6Ch	2b0d1670	uint	4
FFFDCCACCh	2b0d1674	uint	4
FFFDCCACCh	2b0d1678	uint	4
FFFDCA90h	2b0d167c	uint	4
FFFDCCB28h	2b0d1680	uint	4
FFFDCCB28h	2b0d1684	uint	4
FFFDCCB30h	2b0d1688	uint	4
FFFDCCB38h	2b0d168c	uint	4

Filter:

# Memory Map View

- Not shown by default, but can be added by selecting it from the Window menu in the Code Browser
- Helpful for identifying program segments and the memory protections applied to them
  - Readable, Writable, Executable

The screenshot shows the 'Memory Map' window of the CodeBrowser application. The title bar reads 'Memory Map [CodeBrowser: libuClibc:/libuClibc-0.9.33.2.so]'. The menu bar includes 'File', 'Edit', 'Tools', and 'Help'. Below the menu is a toolbar with icons for file operations. The main area is titled 'Memory Map - Image Base: 2b069000'. A table lists memory blocks with the following columns: Name, Start, End, Length, R, W, X, Volat., Type, Initiali..., Source, and Comment. Three entries are listed:

Name	Start	End	Length	R	W	X	Vola...	Type	Initiali...	Source	Comment
segment_3	2b069000	2b0d1c0b	0x68c0c	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	Elf Loader	Loadable segm...
segment_4	2b0e2b2c	2b0e8d1b	0x61f0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input checked="" type="checkbox"/>	Elf Loader	Loadable segm...
EXTERNAL	2b0e9000	2b0e900b	0xc	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Default	<input type="checkbox"/>	Elf Loader	NOTE: This bloc...

# Return Oriented Programming

ROP

# ROP Overview

- **What**
  - ROP leverages existing executable machine code already present in a running process to achieve the exploitation objectives
- **Why**
  - Contemporary systems often use a protection mechanism that prevents memory regions from being both writable and executable (W^X, DEP)
    - This protects against exploits that write shellcode onto the stack then execute it
    - Aleph One described these techniques in his 1996 Phrack paper, “Smashing The Stack For Fun And Profit” [1]
  - Even when W^X protections are not used, ROP is effective in cases where...
    - The vulnerability does not allow sufficient space to write useful shellcode into memory
    - The interface constrains the set of characters (byte) values that can be passed to the vulnerable program

# ROP Evolution

- **Return-into-libc**
  - An evolutionary step toward generalized ROP techniques
  - In 1997, Solar Designer published a proof of concept return-into-libc exploit, and defense recommendations, on a Bugtraq mailing list [6]
  - The exploit involves using a buffer overflow vulnerability to write function arguments and the virtual memory address of a libc function onto the stack such that exploited function returns to the chosen libc function instead of the calling function
- **Return-Oriented Programming (ROP)**
  - ROP as we know it, is generally attributed to Hovav Shacham based on his 2007 paper, “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)” [7]
  - ROP generalizes the return-into-libc approach because doesn’t return to entire functions
  - Instead, ROP returns into short sequences of code called “Gadgets”
- **Return-Oriented Programming (ROP) for the ARM Architecture**
  - Tim Kornau’s 2009 thesis, “Return Oriented Programming for the ARM Architecture,” provides an overview of ROP in the context of ARM and algorithms for searching for ROP gadgets [8]

# ROP Gadgets

- ROP Gadgets are sequences of machine code with these attributes (generally)
- **Short**
  - Anywhere from 2 to roughly 10 instructions
- **No function prologue or epilogue**
  - These are not entire functions, just pieces of functions
- **No defined behavior**
  - Functions typically have defined arguments, processing behaviors, and return values that are described in documentation
    - ROP gadgets are pieces of functions and don't necessarily associate to a single line of source code
  - Programmers do not directly author ROP gadgets, they are part of what gets created when source code is built
    - The exact machine code produced from the source code varies based on the target architecture, the compiler, compiler optimizations, etc.

# Using Ghidra to find ROP Gadgets

# Finding ROP Gadgets With Ghidra

- Scenario
  - We are developing a proof-of-concept exploit for a program on a MIPS system
    - The program accepts input via an UPnP interface (SOAP over HTTP)
  - Our exploit will call the `system` function in the libc library
    - The `system` function accepts one argument, which is a pointer to a string containing a command to execute
- What we need to do
  - Populate the `$a0` register with the address of the command string we want to execute
  - Have execution reach the `system` function in libc
- What we've already determined
  - There is a stack-based buffer overflow vulnerability in the target program
  - By overflowing the buffer we are able to control the contents of the following registers:
    - `$s0-$s8`
    - `$ra`
  - Using another interface on the target program we are able to write a command string into memory at address `0x2b324041`
  - The libc library is loaded into the virtual address space of the target process at offset `0x2b069000` (the end of the executable memory segment of libc is at offset `0x2b0d2000`)

# Finding the Address of the system Function in libc

The libc library we are analyzing includes a symbol for the **system** function so we can use the filter in the Symbol Tree to find it

The screenshot shows the CodeBrowser interface with the following windows:

- Symbol Tree**: Shows a tree structure of symbols. A yellow arrow points from the text box above to the "Exports" section, which contains entries for `svcerr_systemerr` and `system`.
- Listing: libuClibc/libuClibc-0.9.33.2.so**: Displays assembly code for the `system` function. The assembly code is:

```
***** FUNCTION *****
int system(char * __command)
    assume gp = 0xb0b0eb430
    assume t9 = 0xb0cafcd4
    v0:4          <RETURN>
    a0:4          __command
    Stack[-0x4]:4 local_4

    undefined4     Stack[-0x8]:4 local_8
    undefined4     Stack[-0xc]:4 local_c
    undefined4     Stack[-0x18]:4 local_18

    _libc_system
    system
    2b0cafcd4 02 00 1c 3c    lui      gp,0x2
    assume t9 = <UNKNOWN>
    assume gp = <UNKNOWN>
    2b0cafcd8 5c 04 9c 27    addiu   gp,gp,0x45c
    2b0cafcd 21 e0 99 03    addu    gp,sp,t9
    2b0cafe0 d8 ff bd 27    addiu   sp,sp,-0x28
    2b0cafe4 24 00 bf af    sw      ra,local_4(sp)
```
- Decompil...**: Shows the C decompiled code for the `system` function:

```
/* WARNING: Unknown calling convention */ int system(char * __command) { int iVar1; undefined4 uVar2; uint uVar3; undefined auStack40 [16]; undefined *local_18; local_18 = & gp_1; if (__command == (char *)0x0) { iVar1 = FUN_2b0caa50("exit 0"); uVar3 = (uint)(iVar1 == 0); } else { iVar1 = getHWRegister(auStack40); if (*((int *)(&iVar1 + -0x7440))) iVar1 = FUN_2b0caa50(); return iVar1; } uVar2 = FUN_2b0caa0fc(); uVar3 = FUN_2b0caa50(__command); (**(code **)(local_18 + -0x7fd))(); } return uVar3; }
```
- Console - Scripting**: An empty console window.

# Finding the Address of the system Function in libc

Alternatively, we can find the **system** function by filtering or sorting in the Functions View

The screenshot shows the CodeBrowser interface with the following details:

- Title Bar:** CodeBrowser: libuClibc/libuClibc-0.9.33.2.so
- File Menu:** File Edit Analysis Navigation Search Select Tools Window Help
- Toolbar:** Includes icons for file operations, search, and analysis.
- Left Sidebar:**
  - Program Tree
  - Symbol Tree
    - Exports (svcerr\_systemerr, system)
    - Functions (svc..., system)
    - Labels
  - Data Types Manager
- Central Area:** Shows assembly code for the `int system(char * __command)` function. It includes comments like `***** FUNCTION *****`, `assume gp = 0xb0cfa...`, and `assume t9 = <UNKNOWN>`. The assembly code starts with `2b0caf8 02 00 1c 3c lui gp,0x2`.
- Right Sidebar:** Shows the decompiled C code for the `system` function.
- Bottom Panel:**
  - Functions - 2 items (of 1412)

Label	Location	Function Signature	Function Size
svcerr_systemerr	2b0ae7bc	void svcerr_systemerr(SVCPRT * ...)	88
system	2b0caf4	int system(char * __command)	192
  - Filter: system
  - Console and Functions tabs.

# Address of the system Function in libc

The system function is located at virtual address **0x2b0cafd4**

CodeBrowser: libuClibc/libuClibc-0.9.33.2.so

Listing: libuClibc-0.9.33.2.so - (192 addresses selected)

Program Trees

Symbol Tree

Exports

Functions

SVC...

Labels

Data Types

Functions - 2 items (of 1412 )

Label	Location	Function Signature	Function Size
svcerr_systemerr	2b0ae7bc	void svcerr_systemerr(SVCXPRT * __...)	88
system	2b0cafd4	int system(char * __command)	192

Console Functions

Filter: system

File Edit Analysis Navigation Search Select Tools Window Help

Decompil...

```
1 /* WARNING: Unknown calling convention */
2
3 int system(char * __command)
4 {
5     int iVar1;
6     undefined4 uVar2;
7     uint uVar3;
8     undefined auStack40 [16];
9     undefined *local_18;
10
11     local_18 = &gp_1;
12     if (__command == (char *)0x0) {
13         iVar1 = FUN_2b0caa50("exit 0");
14         uVar3 = (uint)(iVar1 == 0);
15     }
16     else {
17         iVar1 = getHWRegister(auStack40);
18         if ((*int *) iVar1 + -0x7440) {
19             iVar1 = FUN_2b0caa50();
20             return iVar1;
21         }
22         uVar2 = FUN_2b0ca0fc();
23         uVar3 = FUN_2b0caa50(__command);
24         (**(code **)(local_18 + -0x7fd))(uVar2, uVar3);
25     }
26     return uVar3;
27 }
28
29 }
```

# Dealing with Bad Characters

- We mentioned that our target program accepts input via an UPnP interface (SOAP over HTTP)
  - This protocol constrains the values of the input characters significantly
  - Supported characters are:
    - 0x09 (Horizontal Tab)
    - 0x0a (Line Feed)
    - 0x0d (Carriage Return)
    - 0x20-0x7e (Printable ASCII Characters)
  - There are some other exceptions for special characters that have special meanings in XML or HTTP (e.g. '<' and '>'), but for this demonstration we will ignore those unless they cause an issue
- All of the bytes in the address containing our command string (0x2b324041) are safe
- The system function is located at address 0x2b0caf4, but
  - Three of those bytes are bad characters so we won't be able to pass them in directly via the UPnP interface
  - Instead, we will need to pass in safe values and find gadgets that will compute the correct address for system

# ROP Gadget Requirements

- We can control the following registers via a stack-based buffer overflow:
  - \$s0-\$s8
  - \$ra
- We need to find a sequence of ROP gadgets that perform the following operations
  - **Move the value 0x2b324041 into \$a0 from one of the saved registers (\$s0-\$s8)**
    - This is the address of the command string
  - **Compute the value 0x2b0caf4 using operands made of safe characters**
    - There are several ways to do this
      - Arithmetic operations (e.g. add, subtract)
      - Logical operations (e.g. XOR, AND)
  - **Move the computed 0x2b0caf4 value into the \$ra register**
    - This is the address of the system function

# Finding Our First Gadget

- At a minimum, let's try to find a gadget that computes the address of the system function using values stored in the registers we control
- We'll start by finding an unsigned subtraction instruction that computes the address of the system function
  - Refer to the MIPS32 Instruction Set Manual for the structure of the SUBU instruction
  - Note that some of the bits in the 4 byte instruction are fixed for the SUBU instruction and some are used to refer to the operands

SUBU						Subtract Unsigned Word		
31	26 25	21 20	16 15	11 10	6 5	0		
SPECIAL 000000	rs	rt	rd	0 00000	SUBU 100011			
Format: SUBU rd, rs, rt							MIPS32	
Purpose: Subtract Unsigned Word To subtract 32-bit integers.								
Description: $GPR[rd] \leftarrow GPR[rs] - GPR[rt]$ The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd.							The MIPS32® Instruction Set Manual [9]	

- $0x4b2d2e52 - 0x20207e7e = 0x2b\textcolor{red}{0caf}d4$

# Finding a SUBU Instruction

- Ghidra provides a memory search feature we can use to find instances of machine code instructions independent of the specific operands
- The ‘Search Memory’ dialog may be launched by choosing ‘Search’ → ‘Memory...’ from the Ghidra CodeBrowser window

The screenshot shows the Ghidra 'Search Memory' dialog on the left and the MIPS32 Instruction Set Manual on the right.

**Ghidra Search Memory Dialog:**

- Search Value: 00100011 .....000 ..... 000000..
- Hex Sequence: 23 00 00 00
- Format Options:
  - Hex
  - String
  - Decimal
  - Binary**
  - Regular Expression
- Memory Block Types:
  - Loaded Blocks**
  - All Blocks
- Selection Scope:
  - Search All
  - Search Selection
- Advanced >>
- Buttons: Next, Previous, Search All, Dismiss

**MIPS32 Instruction Set Manual - SUBU:**

**Format:** SUBU rd, rs, rt

**Purpose:** Subtract Unsigned Word

To subtract 32-bit integers.

**Description:** GPR[rd] ← GPR[rs] – GPR[rt]

The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd.

**Diagram:**

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	5	SUBU 100011
6	5	5	5	5	6	

MIPS32

A callout box highlights the search value in the Ghidra dialog, explaining that dots represent registers and will match any bit value.

The MIPS32® Instruction Set Manual [9] 67

# Finding a SUBU Instruction

Our search found 708 possibilities

The screenshot shows a debugger interface titled "Search Memory - '00100011 .....000 ..... 000000.." [CodeBrowser: libuClibc/libuClibc-0.9.33.2.so]. The search results window has a title bar with "(708 entries)". The main area contains two columns: "Location" and "Code Unit". The "Location" column lists memory addresses, and the "Code Unit" column lists assembly-like instructions. A callout box points to the "Code Unit" column with the text: "Not all of them are the SUBU instructions we were looking for". Another callout box points to the address "2b078138" in the "Location" column with the text: "Some of the addresses contain bad characters". A third callout box points to the bottom right corner of the results window with the text: "Click this button to access the filtering interface".

Location	Code Unit
2b06a344	ddw 23h (dword[1325][41])
2b06d225	ddw 234Eh (Elf32_Sym.st_name)
2b06d834	ddw 1823h (Elf32_Sym.st_name)
2b06dcf5	ddw 23A8h (Elf32_Sym.st_name)
2b06ea05	ddw 2388h (Elf32_Sym.st_name)
2b0764f0	subu _who,v0,_who
2b077f30	subu __cpuset,s0,__cpusetsize
2b078138	subu sp,sp,v0
2b079fbc	subu sp,sp,v0
2b07a0d8	subu v1,s1,s4
2b07c9c4	subu v0,v0,a0
2b07ca14	_subu v0,a1,a3

# Finding a SUBU Instruction

We can filter by the location of the instruction

We can also filter by strings in the assembly instructions

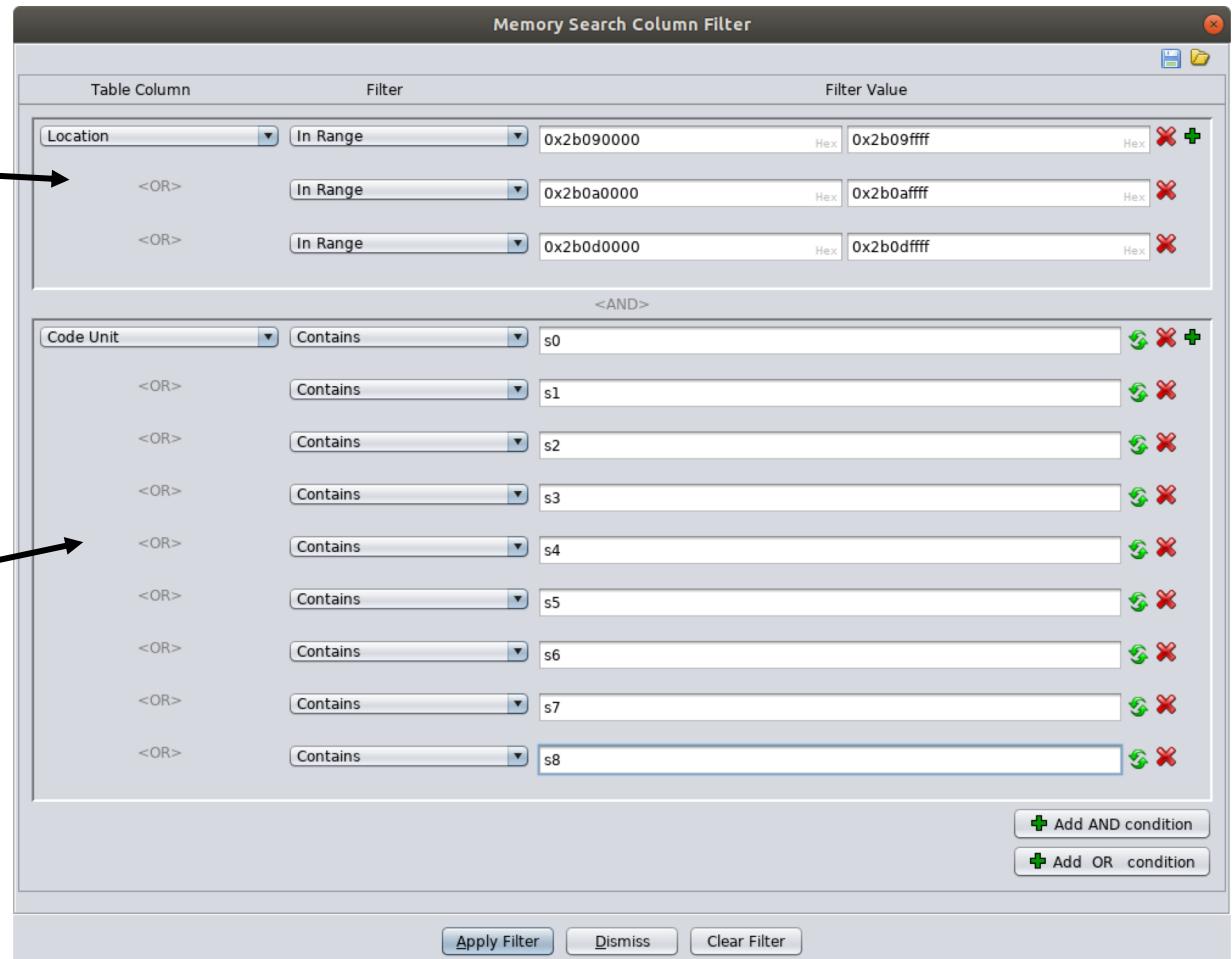
Here we look for instructions that use registers we control as operands

Memory Search Column Filter

Table Column	Filter	Filter Value
Location	In Range	0x2b090000 Hex 0x2b09ffff Hex
<OR>	In Range	0x2b0a0000 Hex 0x2b0affff Hex
<OR>	In Range	0x2b0d0000 Hex 0x2b0dffff Hex
<AND>		
Code Unit	Contains	s0
<OR>	Contains	s1
<OR>	Contains	s2
<OR>	Contains	s3
<OR>	Contains	s4
<OR>	Contains	s5
<OR>	Contains	s6
<OR>	Contains	s7
<OR>	Contains	s8

Add AND condition    Add OR condition

Apply Filter    Dismiss    Clear Filter



# Finding a SUBU Instruction

Some of the addresses still contain bad characters because we didn't filter based on the two least significant bytes

Our filter narrowed the candidates down to 70

Location	Label	Code Unit
2b090b08	LAB_2b090b08	subu s1,v0,s1
2b092044		_subu s2,s1,s0
2b092088		subu s2,s2,s0
2b092868		_subu v0,zero,s0
2b097abc		subu v0,v0,s4
2b0988e0		_subu s0,s0,v0
2b098970		_subu s3,s3,s0
2b098f84		subu s0,s0,a0
2b098fb4	LAB_2b098fb4	subu v0,s0,a0
2b099030		subu v0,s2,s0
2b099070		_subu s2,s2,v1
2b099bf0		_subu s4,s4,param_4

Not all of the instructions use operands we control

...but 70 is a small enough number for us to work through manually

**SUBU → “Subtract Unsigned Word”**

One of the unsigned subtraction instructions we found with our memory search in Ghidra

```

    Listing: libuClibc-0.9.33.2.so
    *libuClibc-0.9.33.2.so

    LAB_2b09202c
    2b09202c 21 c8 60 02 move t9,s3
    2b092030 09 f8 20 03 jalr t9=>fnmatch
    2b092034 00 00 00 00 _nop
    2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
    2b09203c 28 00 bc 8f lw gp,_local_a8(sp)
    2b092040 12 00 00 10 b LAB_2b09208c
    2b092044 23 90 30 02 _subu s2,s1,s0

    LAB_2b092048
    2b092048 03 00 00 10 b LAB_2b092058
    2b09204c ff ff 31 26 _addiu s1,s1,-0x1

    LAB_2b092050
    2b092050 48 81 93 8f lw s3,-0x7eb8(gp)=>->fnmatch
    2b092054 01 00 12 24 li s2,0x1

    LAB_2b092058
    2b092058 2b 10 30 02 sltu v0,s1,s0
    2b09205c 21 28 20 02 move param_2,s1
    2b092060 21 20 c0 02 move param_1,s6
    2b092064 f1 ff 40 10 beq v0,zero,LAB_2b09202c
    2b092068 21 30 00 00 _clear param_3
    2b09206c 9c 00 00 10 b LAB_2b0922e0
    2b092070 21 88 00 02 _move s1,s0
  
```

XREF[1]: 2b092038(j)  
XREF[1]: 2b091f04(j)  
XREF[1]: 2b092048(j)

CodeBrowser: libuClibc:/libuClibc-0.9.33.2.so

The table below shows the registers we can control

As we examine the gadgets we'll capture the register values we need in the table below.

The 'orange' values will need to be set by exploiting the buffer overflow vulnerability in the target program.

The 'green' values are computed and set by the code in the ROP gadgets. We record them below to track their state and indicate their limited availability.

<b>\$s0 = dontcare</b>	<b>\$s1 = dontcare</b>	<b>\$s2 = dontcare</b>	<b>\$s3 = dontcare</b>	<b>\$ra = dontcare</b>
<b>\$s4 = dontcare</b>	<b>\$s5 = dontcare</b>	<b>\$s6 = dontcare</b>	<b>\$s7 = dontcare</b>	<b>\$s8 = dontcare</b>

**Orange → Set this value**      **Green → Computed value**

This is our first gadget.

We need to use the buffer overflow to set the value of \$ra to **0x2b092044** so we reach this code

\$s0 = dontcare	\$s1 = dontcare	\$s2 = dontcare	\$s3 = dontcare	<b>\$ra = 2b092044</b>
\$s4 = dontcare	\$s5 = dontcare	\$s6 = dontcare	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value      Green → Computed value

**\$s2 = \$s1 - \$s0**

We need to make sure **\$s1** and **\$s0** are set correctly so we get the address of 'system' stored in **\$s2**

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = dontcare	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value      Green → Computed value

At this point we are "along for the ride"

Unconditional branch to offset 0x2b092058

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = dontcare	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value

Green → Computed value

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Listing: libuClibc-0.9.33.2.so

```

LAB_2b09202c
2b09202c 21 c8 60 02 move t9,s3
2b092030 09 f8 20 03 jalr t9=>fnmatch
2b092034 00 00 00 00 _nop
2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
2b09203c 28 00 bc 8f lw gp,_local_a8(sp)
2b092040 12 00 00 10 b LAB_2b09208c
2b092044 23 90 30 02 _subu s2,s1,s0

LAB_2b092048
2b092048 03 00 00 10 b LAB_2b092058
2b09204c ff ff 31 26 _addiu s1,s1,-0x1

LAB_2b092050
2b092050 48 81 93 8f lw s3,-0x7eb8(gp)=>->fnmatch
2b092054 01 00 12 24 li s2,0x1

LAB_2b092058
2b092058 2b 10 30 02 sltu v0,s1,s0
2b09205c 21 28 20 02 move param_2,s1
2b092060 21 20 c0 02 move param_1,s6
2b092064 f1 ff 40 10 beq v0,zero,LAB_2b09202c
2b092068 21 30 00 00 _clear param_3
2b09206c 9c 00 00 10 b LAB_2b0922e0
2b092070 21 88 00 02 _move s1,s0

```

XREF[1]: 2b092038(j)

XREF[1]: 2b091f04(j)  
= 2b07c350

XREF[1]: 2b092048(j)

\$v0 = 0

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = dontcare	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value

Green → Computed value

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Listing: libuClibc-0.9.33.2.so

```

LAB_2b09202c
2b09202c 21 c8 60 02 move t9,s3
2b092030 09 f8 20 03 jalr t9=>fnmatch
2b092034 00 00 00 00 _nop
2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
2b09203c 28 00 bc 8f lw gp,_local_a8(sp)
2b092040 12 00 00 10 b LAB_2b09208c
2b092044 23 90 30 02 _subu s2,s1,s0

LAB_2b092048
2b092048 03 00 00 10 b LAB_2b092058
2b09204c ff ff 31 26 _addiu s1,s1,-0x1

LAB_2b092050
2b092050 48 81 93 8f lw s3,-0x7eb8(gp)=>->fnmatch
2b092054 01 00 12 24 li s2,0x1

LAB_2b092058
2b092058 2b 10 30 02 sltu v0,s1,s0
2b09205c 21 28 20 02 move param_2,s1
2b092060 21 20 c0 02 move param_1,s6
2b092064 f1 ff 40 10 beq v0,zero,LAB_2b09202c
2b092068 21 30 00 00 _clear param_3
2b09206c 9c 00 00 10 b LAB_2b0922e0
2b092070 21 88 00 02 _move s1,s0

```

Move the value stored in \$s1 into param\_2

This doesn't accomplish anything we need,  
but it doesn't hurt us

```
int fnmatch(char * __pattern, char *
```

XREF[1]: 2b092038(j)

XREF[1]: 2b091f04(j)  
= 2b07c350

XREF[1]: 2b092048(j)

**\$v0 = 0**

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = dontcare	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value

Green → Computed value

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Listing: libuClibc-0.9.33.2.so

```

LAB_2b09202c
2b09202c 21 c8 60 02 move t9,s3
2b092030 09 f8 20 03 jalr t9=>fnmatch
2b092034 00 00 00 00 _nop
2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
2b09203c 28 00 bc 8f lw gp,_local_a8(sp)
2b092040 12 00 00 10 b LAB_2b09208c
2b092044 23 90 30 02 _subu s2,s1,s0

LAB_2b092048
2b092048 03 00 00 10 b LAB_2b092058
2b09204c ff ff 31 26 _addiu s1,s1,-0x1

LAB_2b092050
2b092050 48 81 93 8f lw s3,-0x7eb8(gp)=>->fnmatch
2b092054 01 00 12 24 li s2,0x1

LAB_2b092058
2b092058 2b 10 30 02 sltu v0,s1,s0
2b09205c 21 28 20 02 move param_2,s1
2b092060 21 20 c0 02 move param_1,s6
2b092064 f1 ff 40 10 beq v0,zero,LAB_2b09202c
2b092068 21 30 00 00 _clear param_3
2b09206c 9c 00 00 10 b LAB_2b0922e0
2b092070 21 88 00 02 _move s1,s0

```

Move the value stored in \$s6 into param\_1

What is param\_1?

```
int fnmatch(char * __pattern, c
```

XREF[1]: 2b092038(j)

XREF[1]: 2b091f04(j)  
= 2b07c350

XREF[1]: 2b092048(j)

\$v0 = 0

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = dontcare	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value

Green → Computed value

File Edit Analysis Navigation

Program Trees X

lib

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

```

2b090f3c 3c 00 03 01  lw      $s0,local_10(sp)
2b090f40 58 00 b2 8f  lw      $s2,local_20(sp)
2b090f44 54 00 b1 8f  lw      $s1,local_24(sp)
2b090f48 50 00 b0 8f  lw      $s0,local_28(sp)
2b090f4c 08 00 e0 03  jr      ra
2b090f50 78 00 bd 27  _addiu   sp,sp,0x78

*****
*          FUNCTION
*****
undefined FUN_2b090f54(undefined param_1, undefined para...
assume gp = 0x2b0eb430
assume t9 = 0x2b090f54
v0:1 <RETURN>
a0:1 |param_1|
a1:1 param_2
a2:1 param_3
a3:1 param_4
Stack[0x10]:4 param_5
XREF[1]: 2b090f90(R)
XREF[8]: 2b091030(R),
2b091088(R),
2b091090(R),
2b0910b4(R),
2b091194(R),
2b091254(R),
2b091ba4(R),
2b091d30(R)
XREF[7]: 2b0910cc(R),
2b0910dc(W),
2b091198(R)

We can mouse over param_1
and see where it is defined

$ v0 = 0

$ s0 = 20207e7e
$ s1 = 4d2d2e52
$ s2 = 2b0caf4
$ s3 = dontcare
$ ra = 2b092044

$ s4 = dontcare
$ s5 = dontcare
$ s6 = dontcare
$ s7 = dontcare
$ s8 = dontcare

```

Approved for public release; unlimited distribution  
Not export controlled per ES-FL-091219-0198

Orange → Set this value

Green → Computed value

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Data Types

- BuiltInTypes
- libuClibc-0.9.33
- generic\_clib

Filter:

Listing: libuClibc-0.9.33.2.so

libuClibc-0.9.33.2.so X

```

LAB_2b09202c
2b09202c 21 c8 60 02 move t9,s3
2b092030 09 f8 20 03 jalr t9=>fnmatch
2b092034 00 00 00 00 _nop
2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
2b09203c 28 00 bc 8f lw gp,_local_a8(sp)
2b092040 12 00 00 10 b LAB_2b09208c
2b092044 23 90 30 02 _subu s2,s1,s0

LAB_2b092048
2b092048 03 00 00 10 b LAB_2b092058
2b09204c ff ff 31 26 _addiu s1,s1,-0x1

LAB_2b092050
2b092050 48 81 93 8f lw s3,-0x7eb8(gp)=>->fnmatch
2b092054 01 00 12 24 li s2,0x1

LAB_2b092058
2b092058 2b 10 30 02 sltu v0,s1,s0
2b09205c 21 28 20 02 move param_2,s1
2b092060 21 20 c0 02 move param_1,s6
2b092064 f1 ff 40 10 beq v0,zero,LAB_2b09202c
2b092068 21 30 00 00 _clear param_3
2b09206c 9c 00 00 10 b LAB_2b0922e0
2b092070 21 88 00 02 _move s1,s0

```

We want to put the address of our command into \$s6 so it will get moved into \$a0 as the argument for the 'system' call

int fnmatch(char \* \_\_pattern, c

param\_1 is an alias for \$a0

XREF[1]: 2b092038(j)

XREF[1]: 2b091f04(j)  
= 2b07c350

XREF[1]: 2b092048(j)

\$v0 = 0

\$a0 = 2b324041

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value

Green → Computed value

Program Trees X

listing: libuClibc-0.9.33.2.so X

libuClibc-0.9.33.2.so X

```

    .text
    .globl _fnmatch
    .type _fnmatch, int
_fnmatch:
    .L2b09202c:                                move    t9,s3
    .L2b092030:       21 c8 60 02      jalr    t9=>fnmatch
    .L2b092034:       09 f8 20 03      _nop
    .L2b092038:       00 00 00 00      beq    v0,s2,LAB_2b092048
    .L2b09203c:       28 00 bc 8f      _lw     gp,local_a8(sp)
    .L2b092040:       2b 00 00 10      b      LAB_2b09208c
    .L2b092044:       23 90 30 02      _subu   s2,s1,s0

    .L2b092048:                                b      LAB_2b092058
    .L2b09204c:       03 00 00 10      _addiu  s1,s1,-0x1

    .L2b092050:       48 81 93 8f      lw      s3,-0x7eb8(gp)=>->fnmatch
    .L2b092054:       01 00 12 24      li      s2,0x1

    .L2b092058:                                sltu   v0,s1,s0
    .L2b09205c:       2b 10 30 02      move   param_2,s1
    .L2b092060:       21 28 20 02      move   param_1,s6
    .L2b092064:       f1 ff 40 10      beq    v0,zero,LAB_2b09202c
    .L2b092068:       21 30 00 00      _clear param_3
    .L2b09206c:       9c 00 00 10      b      LAB_2b0922e0
    .L2b092070:       21 88 00 02      _move   s1,s0

```

Branch to offset 0x2b09202c  
if \$v0 = 0  
(it does)

XREF[1]: 2b092038(j)

XREF[1]: 2b091f04(j)  
= 2b07c350

XREF[1]: 2b092048(j)

\$v0 = 0

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

**Orange → Set this value**

**Green → Computed value**

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Data Types

- BuiltInTypes
- libuClibc-0.9.33
- generic\_clib

Filter:

**Move the value in \$s3 into \$t9**

**\$t9 = \$s3**

**\$a0 = 2b324041**

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

**Orange → Set this value**      **Green → Computed value**

```

2b09202c 21 c8 60 02 move t9,s3
2b092030 09 f8 20 03 jalr t9=>fnmatch
2b092034 00 00 00 00 _nop
2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
2b09203c 28 00 bc 8f lw go_local_a8(sp)
2b092040 12 00 00 10 b LAB_2b09208c
2b092044 23 90 30 02 _subu s2,s1,s0

LAB_2b092048
2b092048 03 00 00 10 b
2b09204c ff ff 31 26 _addiu LAB_2b092058
s1,s1,-0x1

LAB_2b092050
2b092050 48 81 93 8f lw s3,-0x7eb8(sp)=>->fnmatch
2b092054 01 00 12 24 li s2,0x1

LAB_2b092058
2b092058 2b 10 30 02 sltu v0,s1,s0
2b09205c 21 28 20 02 move param_2,s1
2b092060 21 20 c0 02 move param_1,s6
2b092064 f1 ff 40 10 beq v0,zero,LAB_2b09201c
2b092068 21 30 00 00 _clear param_3
2b09206c 9c 00 00 10 b LAB_2b0922e0
2b092070 21 88 00 02 _move s1,s0

```

**JALR → “Jump and Link Register”**

Set the program counter to the value stored in \$t9

...which is the same as the value stored in \$s3  
(we control \$s3)

\$t9 = \$s3

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = dontcare	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value      Green → Computed value

XREF[1]: 2b092038(j)  
XREF[1]: 2b091f04(j)  
XREF[1]: 2b092048(j)

Approved for public release; unlimited distribution  
Not export controlled per ES-FL-091219-0198

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Data Types

- BuiltInTypes
- libuClibc-0.9.33
- generic\_clib

Filter:

Listing: libuClibc-0.9.33.2.so

```

LAB_2b09202c
2b09202c 21 c8 60 02 move t9,s3
2b092030 09 f8 20 03 jalr t9=>fnmatch
2b092034 00 00 00 00 _nop
2b092038 03 00 52 10 beq v0,s2,LAB_2b092048
2b09203c 28 00 bc 8f lw gp,_local_a8(sp)
2b092040 12 00 00 10 b LAB_2b09208c
2b092044 23 90 30 02 _subu s2,s1,s0

LAB_2b092048
2b092048 03 00 00 10 b LAB_2b092058
2b09204c ff ff 31 26 _addiu s1,s1,-0x1

LAB_2b092050
2b092050 48 81 93 8f lw s3,-0x7eb8(gp)=>->fnmatch
2b092054 01 00 12 24 li s2,0x1

LAB_2b092058
2b092058 2b 10 30 02 sltu v0,s1,s0
2b09205c 21 28 20 02 move param_2,s1
2b092060 21 20 c0 02 move param_1,s6
2b092064 f1 ff 40 10
2b092068 21 30 00 00
2b09206c 9c 00 00 10
2b092070 21 88 00 02

```

JALR → “Jump and Link Register”

Set the program counter to the value stored in \$t9

...which is the same as the value stored in \$s3  
(we directly control \$s3)

XREF[1]: 2b092038(j)

XREF[1]: 2b091f04(j)  
= 2b07c350

XREF[1]: 2b092048(j)

So \$s3 needs to contain the address of our next gadget

\$t9 = 2<sup>nd</sup> gadget

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = 2 <sup>nd</sup> gadget	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value

Green → Computed value

# ROP Gadget Requirements Review

- We can control the following registers via a stack-based buffer overflow:
  - \$s0-\$s8
  - \$ra
- We need to find a sequence of ROP gadgets that perform the following operations
  - **Move the value 0x2b324041 into \$a0 from one of the saved registers (\$s0-\$s8)**
    - This is the address of the command string
  - **Compute the value 0x2b0caf4 using operands made of safe characters**
    - There are several ways to do this
      - Arithmetic operations (e.g. add, subtract)
      - Logical operations (e.g. XOR, AND)
  - **Move the computed 0x2b0caf4 value into the \$ra register**
    - This is the address of the system function

# Finding Our Second Gadget

At the end of the first gadget we had an instruction that... **moved a value from a saved register into \$t9**...followed by an instruction that... **jumped to the address in \$t9**

```
2b09202c 21 c8 60 02      move      t9,s3  
2b092030 09 f8 20 03      jalr      t9=>fnmatch
```

Let's look for another sequence like that  
...except we want one that moves the value in \$s2 into \$t9  
because \$s2 has the address of 'system'

# About the MIPS MOVE instruction

- We saw a MIPS MOVE instruction in Ghidra's disassembly
- To search for the specific instruction we want (`move t9,s2`) we should consult the MIPS32 Instruction Set Reference to find the correct machine code bytes
  - ...but there is no MIPS MOVE instruction
- The MOVE instruction is really a pseudoinstruction
- The assembler interprets...

```
move rd, rs  
      as  
addu rd, rs, zero
```

Note: The zero register always holds the value zero

# Finding a ‘move t9,s2’ Instruction

- We use the memory search feature to search specifically for an addu t9,s2,zero instruction

There are no dots in the search value because we are searching for a specific instruction

The screenshot shows a 'Search Memory' dialog box on the left and the MIPS32 Instruction Set Manual on the right.

**Search Memory Dialog:**

- Search Value: 00100001 11001000 01000000 00000010
- Hex Sequence: 21 c8 40 02
- Format: Binary (selected)
- Memory Block Types: Loaded Blocks (selected)
- Selection Scope: Search All (selected)
- Buttons: Next, Previous, Search All, Dismiss, Advanced >>

**MIPS32 Instruction Set Manual:**

**ADDU** Add Unsigned Word

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	ADDU 100001	
6	5	5	5	5	6	

**Format:** ADDU rd, rs, rt

**Purpose:** Add Unsigned Word

To add 32-bit integers.

**Description:** GPR[rd]  $\leftarrow$  GPR[rs] + GPR[rt]

The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd.

The MIPS32® Instruction Set Manual [9]

# Finding a ‘move t9,s2’ instruction

Our search found 128 possibilities

The screenshot shows a memory search results window titled "Search Memory - <hex> [CodeBrowser: libuClibc:/libuClibc-0.9.33.2.so]". The status bar indicates "(128 entries)". The table has columns for Location, Label, and Code Unit. Most entries show the address and the instruction "move t9,s2". One entry at address 2b089f98 is highlighted with a red box and labeled "Some of the addresses contain bad characters". A button in the bottom right corner is labeled "Click this button to access the filtering interface".

Location	Label	Code Unit
2b074d24		move t9,s2
2b074d54		move t9,s2
2b07aeb4		move t9,s2
2b07b4d4		move t9,s2
2b0894f0	LAB_2b0894f0	move t9,s2
2b089ef4		move t9,s2
2b089f14		move t9,s2
2b089f98		move t9,s2
2b089fc0		move t9,s2
2b089fe8		move t9,s2
2b08a010		move t9,s2
2b08a038		move t9,s2

Some of the addresses contain bad characters

Click this button to access the filtering interface

# Finding a 'move t9, s2' instruction

We can filter by the location of the instruction

Note: We aren't filtering based on the two least significant bytes because the filter would be too granular to define manually

Table Column	Filter	Filter Value
Location	In Range	0x2b090000 Hex 0x2b09ffff Hex
<OR>	In Range	0x2b0a0000 Hex 0x2b0affff Hex
<OR>	In Range	0x2b0d0000 Hex 0x2b0dffff Hex

Add AND condition

Add OR condition

Apply Filter Dismiss Clear Filter

# Finding a ‘move t9,s2’ instruction

Some of the addresses still contain bad characters because we didn't filter based on the two least significant bytes

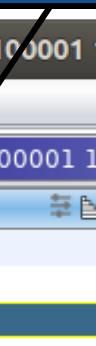
Our filter narrowed the candidates down to 50

The screenshot shows a 'Search Memory' window from a debugger. The title bar reads 'Search Memory - "00100001 11001000 01000000 00000010" [CodeBrowser: libuClibc:/libuClibc-0.9.33.2.so]'. The main pane displays a table of search results:

Location	Label	Code Unit
2b0919a0	LAB_2b0919a0	move t9,s2
2b091b10		move t9,s2
2b09528c		move t9,s2
2b0952fc		move t9,s2
2b09b7cc	LAB_2b09b7cc	move t9,s2
2b09d4f0	LAB_2b09d4f0	move t9,s2
2b09d6a8	LAB_2b09d6a8	move t9,s2
2b09d78c	LAB_2b09d78c	move t9,s2
2b09d870	LAB_2b09d870	move t9,s2
2b09d898		_move t9,s2
2b09e4f8		move t9,s2
2b09ebbe8	LAB_2b09ebbe8	move t9,s2

# Finding a ‘move t9,s2’ instruction

This gadget is located at an address with all safe characters and performs the operations we need



Location	Label	Code Unit
2b0a328c	LAB_2b0a328c	move t9,s2
2b0a36c8		move t9,s2
2b0a5a70		move t9,s2
2b0a5a88		move t9,s2
2b0a68d8		move t9,s2
2b0a6918		move t9,s2
2b0a69b0		move t9,s2
2b0a69ec		move t9,s2
2b0aa2f4		move t9,s2
2b0aa32c		_move t9,s2
2b0aacac		move t9,s2
2b0aacc4		move t9,s2

Program Trees X

- libuClibc-0.9.33.2.
  - segment\_4
  - segment\_3
  - EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

Data Types
 

- BuiltInTypes
- libuClibc-0.9.33
- generic\_clib

Filter:

Listing: libuClibc-0.9.33.2.so

libuClibc-0.9.33.2.so X

```

2b0a5a54 21 88 80 00 move s1,a0
2b0a5a58 1c 00 65 26 addiu a1,s3,0x1c
2b0a5a5c 21 20 00 02 move a0,s0
2b0a5a60 90 01 06 24 li a2,0x190
2b0a5a64 09 f8 20 03 jalr t9=>xdrmem_create
2b0a5a68 21 38 00 00 _clear a3
2b0a5a6c 21 20 00 02 move a0,s0
2b0a5a70 21 c8 40 02 move t9,s2
2b0a5a74 09 f8 20 03 jalr t9=>xdr_opaque_auth
2b0a5a78 21 28 20 02 _move a1,s1
2b0a5a7c 07 00 40 10 beq v0,zero,LAB_2b0a5a9c
2b0a5a80 10 00 bc 8f _lw gp,local_38(sp)
2b0a5a84 0c 00 25 26 addiu a1,s1,0xc
2b0a5a88 21 c8 40 02 move t9,s2
2b0a5a8c 09 f8 20 03 jalr t9=>xdr_opaque_auth
2b0a5a90 21 20 00 02 _move a0,s0
2b0a5a94 07 00 40 14 bne v0,zero,LAB_2b0a5ab4
2b0a5a98 10 00 bc 8f _lw gp,local_38(sp)

LAB_2b0a5a9c
2b0a5a9c 4c 80 84 8f lw a0,-0x7fb4(gp)=>PTR_LAB_2b0e347c
2b0a5aa0 f0 84 99 8f lw t9,-0x7b10(gp)=>->perror
2b0a5aa4 09 f8 20 03 jalr t9=>perror
2b0a5aa8 e0 6c 84 24 _addiu a0=>s_auth_none.c_-_Fatal_marshall_p_2b0cfca...
2b0a5aac 07 00 00 10 b LAB_2b0a5acc
2b0a5ab0 10 00 bc 8f lw qp,local_38(sp)

```

One of the `move t9,s2` instructions  
we found with our memory search in  
Ghidra

2b0a5a70 FUN\_2b0a5alc move t9,s2

We want to reach the instruction at this address (because this is the address of the **system** function) after the second gadget finishes executing.

We continue populating the table we started earlier.

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = 2 <sup>nd</sup> gadget	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

**Orange → Set this value**

**Green → Computed value**

Program Trees

- libuClibc-0.9.33.2
  - segment\_4
  - segment\_3
  - EXTERNAL

Program Tree

Symbol Tree

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type Manager

Data Types
 

- BuiltInTypes
- libuClibc-0.9.33
- generic\_clib

Filter:

Listing: libuClibc-0.9.33.2.so

```

2b0a5a54 21 88 80 00 move    $1,a0
2b0a5a58 1c 00 65 26 addiu   a1,$3,0x1c
2b0a5a5c 21 20 00 02 move    a0,s0
2b0a5a60 90 01 06 24 li     a2,0x190
2b0a5a64 09 f8 20 03 jalr   t9=>xdrmem_create
2b0a5a68 21 38 00 00 _clear
2b0a5a6c 21 20 00 02 move    a0,s0
2b0a5a70 21 c8 40 02 move    t9,s2
2b0a5a74 09 f8 20 03 jalr   t9=>xdr_opaque_auth
2b0a5a78 21 28 20 02 _move
2b0a5a7c 07 00 40 10 beq    v0,zero,LAB_2b0a5a9c
2b0a5a80 10 00 bc 8f _lw    gp,local_38(sp)
2b0a5a84 0c 00 25 20 addiu  a1,s1,0xc
2b0a5a88 21 c8 40 02 move    t9,s2
2b0a5a8c 09 f8 20 03 jalr   t9=>xdr_opaque_auth
2b0a5a90 21 20 00 02 _move
2b0a5a94 07 00 40 14 bne    a0,s0
2b0a5a98 10 00 bc 8f _lw    v0,zero,LAB_2b0a5ab4
                                         gp,local_38(sp)

LAB_2b0a5a9c
2b0a5a9c 4c 80 84 8f lw     a0,-0x7fb4(gp)=>PTR_LAB_2b0e347c
2b0a5aa0 f0 84 99 8f lw     t9,-0x7b10(gp)=>->perror
2b0a5aa4 09 f8 20 03 jalr   t9=>perror
2b0a5aa8 e0 6c 84 24 addiu  a0=>s_auth_none.c_-_Fatal_marshall_p_2b0cfca...
2b0a5aac 07 00 00 10 b     LAB_2b0a5ac
2b0a5ab0 10 00 bc 8f _lw    qp,local_38(sp)

```

Based on the first gadget we determined we need to populate the **\$s3** register with the address of the second gadget.

We need to use the buffer overflow to set the value of **\$s3** to **0x2b0a5a70** so we reach this code

```

bool_t xdr_opaque_auth(XDR * pa
XREF[1]: 2b0a5a7c(j)
          = 2b0c9000
          = 2b096ae0
void perror(char * __s)
          void perror(char * __s)
          "auth_none.c - Fatal marshall...

```

<b>\$s0 = 20207e7e</b>	<b>\$s1 = 4d2d2e52</b>	<b>\$s2 = 2b0caf4</b>	<b>\$s3 = 2b0a5a70</b>	<b>\$ra = 2b092044</b>
<b>\$s4 = dontcare</b>	<b>\$s5 = dontcare</b>	<b>\$s6 = 2b324041</b>	<b>\$s7 = dontcare</b>	<b>\$s8 = dontcare</b>

Orange → Set this value

Green → Computed value

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Console - Scripting

**Move the value in \$s2 into \$t9**

```

Listing: libuClibc-0.9.33.2.so
libuClibc-0.9.33.2.so X

2b0a5a54 21 88 80 00 move    s1,a0
2b0a5a58 1c 00 65 26 addiu   a1,s3,0x1c
2b0a5a5c 21 20 00 02 move    a0,s0
2b0a5a60 90 01 06 24 li      a2,0x190
2b0a5a64 09 f8 20 03 jalr   t9=>xdrmem_create
2b0a5a68 21 38 00 00 _clear
2b0a5a6c 21 20 00 02 move    a3
2b0a5a70 21 c8 40 02 move    t9,s2
jalr   t9=>xdr_opaque_auth
_move
beq
_lw
addiu
move
jalr
_move
a0,s0
_bne
_lw
_LAB_2b0a5a9c
lw
a0,-0x7fb4(gp)=>PTR_LAB_2b0e347c
2b0a5aa0 f0 84 99 8f lw
t9,-0x7b10(gp)=>->perror
2b0a5aa4 09 f8 20 03 jalr   t9=>perror
2b0a5aa8 e0 6c 84 24 addiu
2b0a5aac 07 00 00 10 b
2b0a5ab0 10 00 bc 8f lw
LAB_2b0a5acc
gp,local_38(sp)
_XREF[1]: 2b0a5a7c(j)
= 2b0c9000
= 2b006ae0
void perror(char * __s)
= "auth none.c - Fatal_marshall
$ra = 2b092044
$t9 = 2b0caf4
$a0 = 2b324041

```

**XREF[1]: 2b0a5a7c(j)**

**\$t9 = 2b0caf4**

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = 2b0a5a70	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

**Orange → Set this value****Green → Computed value**

**JALR → “Jump and Link Register”**

Set the program counter to the value stored in \$t9

...which is the same as the value stored in \$s2  
(we indirectly control \$s2)

\$t9 = 2b0caf4

\$s0 = 20207e7e	\$s1 = 4d2d2e52	\$s2 = 2b0caf4	\$s3 = 2b0a5a70	\$ra = 2b092044
\$s4 = dontcare	\$s5 = dontcare	\$s6 = 2b324041	\$s7 = dontcare	\$s8 = dontcare

Orange → Set this value      Green → Computed value

Approved for public release; unlimited distribution  
Not export controlled per ES-FL-091219-0198

Program Trees X

libuClibc-0.9.33.2.

- segment\_4
- segment\_3
- EXTERNAL

Program Tree X

Symbol T... X

- Imports
- Exports
- Functions
- Labels
- Classes
- Namespaces

Filter:

Data Type M... X

Data Types
 

- BuiltInTypes
- libuClibc-0.9.33
- generic\_clib

Filter:

Listing: libuClibc-0.9.33.2.so

```

2b0a5a54 21 88 80 00 move    $1,a0
2b0a5a58 1c 00 65 26 addiu   $1,s3,0x1c
2b0a5a5c 21 20 00 02 move    $0,s0
2b0a5a60 90 01 06 24 li     $2,0x190
2b0a5a64 09 f8 20 03 jalr   t9=>xdrmem_create
2b0a5a68 21 38 00 00 _clear
2b0a5a6c 21 20 00 02 move    $0,s0
2b0a5a70 21 c8 40 02 move    $9,s2
2b0a5a74 09 f8 20 03 jalr   t9=>xdr_opaque_auth
2b0a5a78 21 28 20 02 _move
2b0a5a7c 07 00 40 10 beq    v0,zero,LAB_2b0a5a9c
2b0a5a80 10 00 bc 8f _lw
2b0a5a84 0c 00 25 26 addiu
2b0a5a88 21 c8 40 02 move
2b0a5a8c 09 f8 20 03 jalr
2b0a5a90 21 20 00 02 _move
2b0a5a94 07 00 40 14 bne
2b0a5a98 10 00 bc 8f _lw
LAB_2b0a5a9c
2b0a5a9c 4c 80 84 8f lw
2b0a5a9d 50 01 00 0f lw

```

JALR → “Jump and Link Register”

Set the program counter to the value stored in **\$t9**

...which is the same as the value stored in **\$s2**  
(the first gadget populated **\$s2**)

**\$t9** contains the address of the **system** function

...and after the **jalr** instruction is executed we will reach the **system** function

**\$a0** contains the address of the command string we want the **system** function to execute  
(the first gadget populated **\$s2**)

The **system** function will find its argument in the **\$a0** register.

**\$t9 = 2b0caf4**

**\$a0 = 2b324041**

<b>\$s0 = 20207e7e</b>	<b>\$s1 = 4d2d2e52</b>	<b>\$s2 = 2b0caf4</b>	<b>\$s3 = 2b0a5a70</b>	<b>\$ra = 2b092044</b>
<b>\$s4 = dontcare</b>	<b>\$s5 = dontcare</b>	<b>\$s6 = 2b324041</b>	<b>\$s7 = dontcare</b>	<b>\$s8 = dontcare</b>

Orange → Set this value

Green → Computed value

# ROP Gadget Requirements Review

- We can control the following registers via a stack-based buffer overflow:
  - \$s0-\$s8
  - \$ra
- We need to find a sequence of ROP gadgets that perform the following operations
  - Move the value **0x2b324041** into **\$a0** from one of the saved registers (**\$s0-\$s8**)
    - This is the address of the command string
  - Compute the value **0x2b0caf4** using operands made of safe characters
    - There are several ways to do this
      - Arithmetic operations (e.g. add, subtract)
      - Logical operations (e.g. XOR, AND)
  - Move the computed **0x2b0caf4** value into the **\$ra** register
    - This is the address of the system function

# ROP Gadgets With Ghidra: Conclusions

- We used off-the-shelf features in Ghidra to find a sequence of two ROP gadgets that perform register manipulations needed to construct a proof-of-concept exploit
  - This work would be preceded by vulnerability identification and characterization using fuzzing and debugging, which we did not cover here
- Ghidra's off-the-shelf GUI features helped us do this reasonably efficiently
  - The automation and scripting features provided by Ghidra can improve our efficiency
- The difficult part of constructing ROP chains is examining candidate gadgets to find ones that...
  - Do, at least, perform the necessary operations
  - Do not perform operations that undo or invalidate the results of previous operations
- Olivia Lucca Fraser (a.k.a. Oblivia Simplex) presents interesting and recent research in her master's thesis related to algorithms and tooling for ROP chain generation [10] [11]

# References

# References

- [1] E. J. Chikofsky and J. H. Cross, “Reverse engineering and design recovery: A taxonomy,” *IEEE Software*, vol. 7, pp. 13-17, 1990.
- [2] Trend Micro Inc., “Back-to-Back Campaigns: Neko, Mirai, and Bashlite Malware Variants Use Various Exploits to Target Several Routers, Devices,” Trend Micro Security Intelligence Blog, Aug. 13, 2019. [Online]. Available: <https://blog.trendmicro.com/trendlabs-security-intelligence/back-to-back-campaigns-neko-mirai-and-bashlite-malware-variants-use-various-exploits-to-target-several-routers-devices/>. [accessed Aug. 30, 2019].
- [3] S. Hollister, “Hackers obtain PS3 private cryptography key due to epic programming fail? (update),” Engadget, Dec. 29, 2010. [Online]. Available: <https://www.engadget.com/2010/12/29/hackers-obtain-ps3-private-cryptography-key-due-to-epic-programm/>. [Accessed Aug. 30, 2019].
- [4] C. Johnston, “PS3 hacked through poor cryptography implementation,” Ars Technica, Dec. 20, 2010. [Online]. Available: <https://arstechnica.com/gaming/2010/12/ps3-hacked-through-poor-implementation-of-cryptography/>. [Accessed Aug. 30, 2019].

# References

- [5] Aleph One, “Smashing The Stack For Fun And Profit,” *Phrack*, vol. 7, issue 49, Nov. 8, 1996. [Online]. Available: <http://phrack.org/issues/49/14.html>. [Accessed Aug. 30, 2019].
- [6] Solar Designer, “Getting around non-executable stack (and fix),” Bugtraq, Aug. 10, 1997. [Online]. Available: <https://seclists.org/bugtraq/1997/Aug/63>. [Accessed Aug. 30, 2019].
- [7] H. Shacham, “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86),” In Proc. 14th ACM conference on Computer and Communications Security, 2007, pp. 552-561.
- [8] T. Kornau, “Return Oriented Programming for the ARM Architecture.” M.S. thesis, Ruhr-Universität Bochum, Dec. 22, 2009. [Online]. Available: <http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf>. [Accessed Aug. 30, 2019].
- [9] MIPS Technologies Inc., “MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual,” Doc. No. MD00086, Rev. 6.06, Dec. 15, 2016. [Online]. Available: <https://www.mips.com/?do-download=the-mips32-instruction-set-v6-06>. [Accessed Aug. 30, 2019].

# References

- [10] O. L. Fraser, “Urschleim in Silicon: Return Oriented Program Evolution with ROPER,” M.S. thesis, Dalhousie University, Halifax, Nova Scotia, Dec. 2018. [Online]. Available: <https://pdfs.semanticscholar.org/959e/d1164c7919e9bc22779c75b13c32f4e37129.pdf>. [Accessed Aug. 30, 2019].
- [11] O. L. Fraser (Oblivia-Simplex), “Return Oriented Programme Evolution with ROPER,” [Online]. Available: <https://github.com/oblivia-simplex/roper>. [Accessed Aug. 30, 2019].