Internet of Things Reverse Engineering with Ghidra

Troy Shurtleff (Author)

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 failoverflow 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 failoverflow 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

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

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

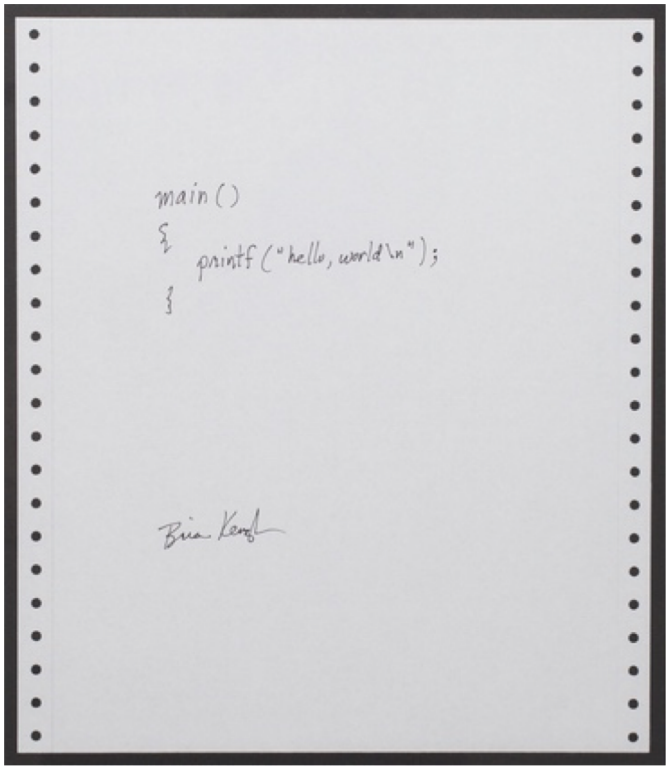
Reverse Engineering Demonstration

- **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

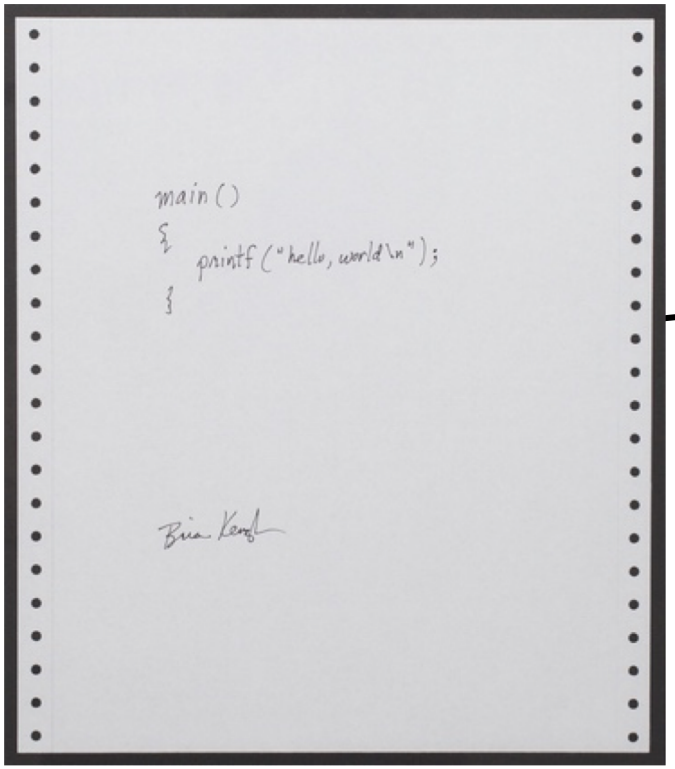


Original Expression of Requirements and Design

- 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

Writing the Original Source Code



Original Expression of Requirements and Design

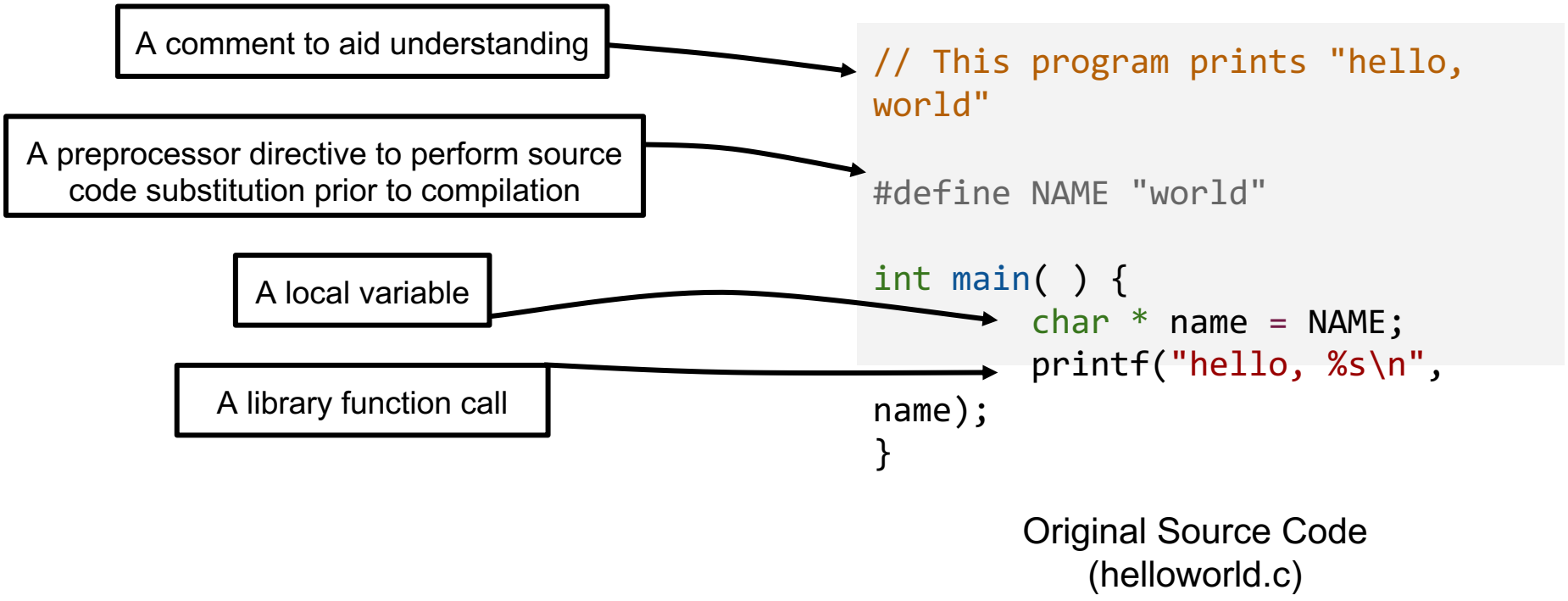
Initially, programmers translate some expression of the intended design into source code

```
// 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

Original Source Code Observations



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

The C preprocessor removes comments and applies the macros

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

#define NAME "world"

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

Original Source Code
(helloworld.c)

```
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 00  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  |...5.0..........|
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 00 01 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 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 01 00 00 00  06 00 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  1b 00 00 00 09 00 00 00  |................|
000002b0  40 00 00 00 00 00 00 00  dc 01 00 00 18 00 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  |l...............|
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  |........l.......|
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  |l...............|
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                           |........|
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:                             0x5000000, 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

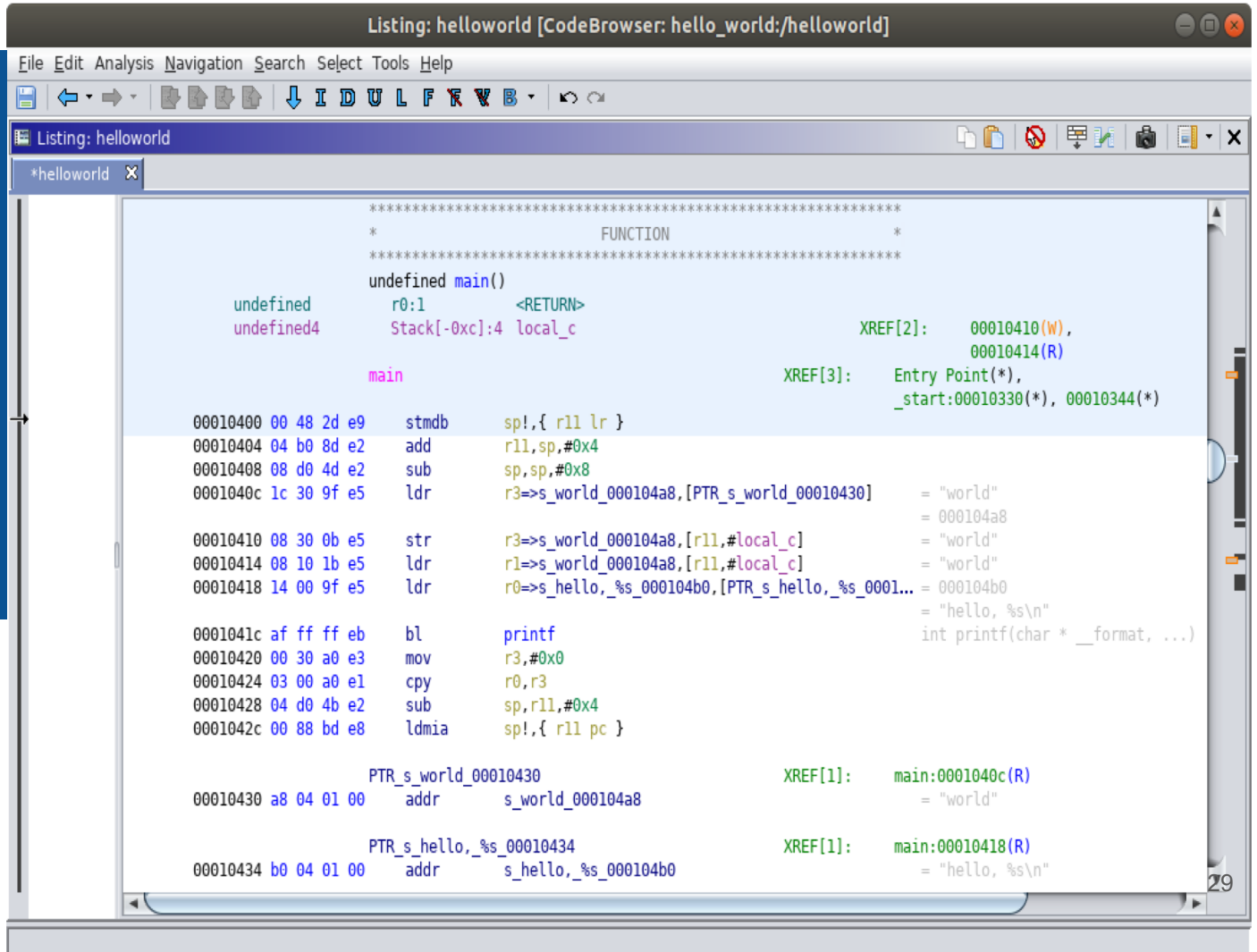
- We may only have a piece of executable software to begin our reverse engineering efforts
  - This is likely to be the case for a malware sample
- If we are reverse engineering an IoT device, we may have a fully running system with associated configuration files
- For this demonstration, we focus reversing 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



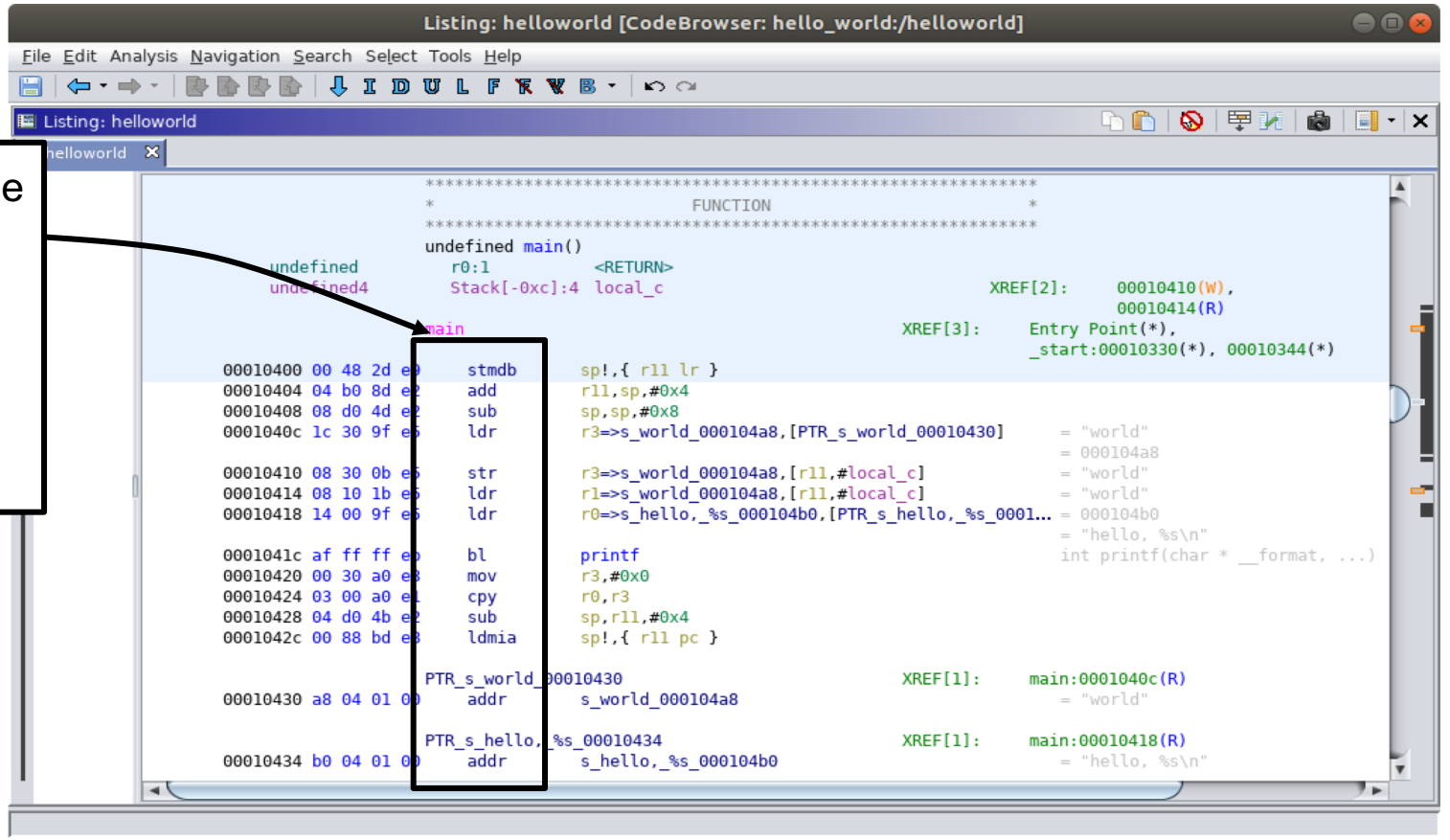
Ghidra's Disassembly (helloworld)

Disassembly

Decompilation | Disassembly

ARM Machine Code Instructions and Data

Directly from the helloworld ELF executable

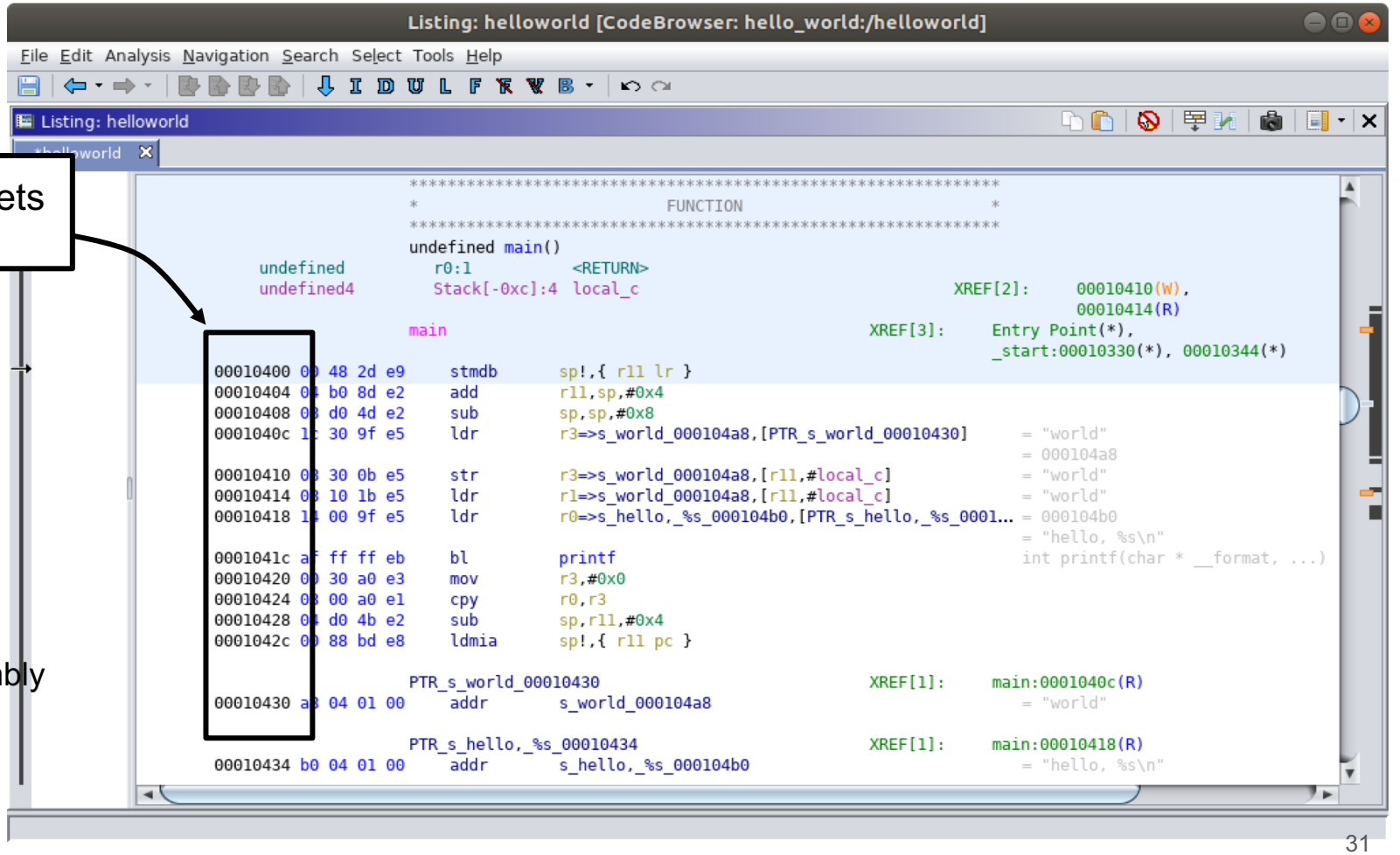


Ghidra's Disassembly (helloworld)

Disassembly

Decompilation | Disassembly

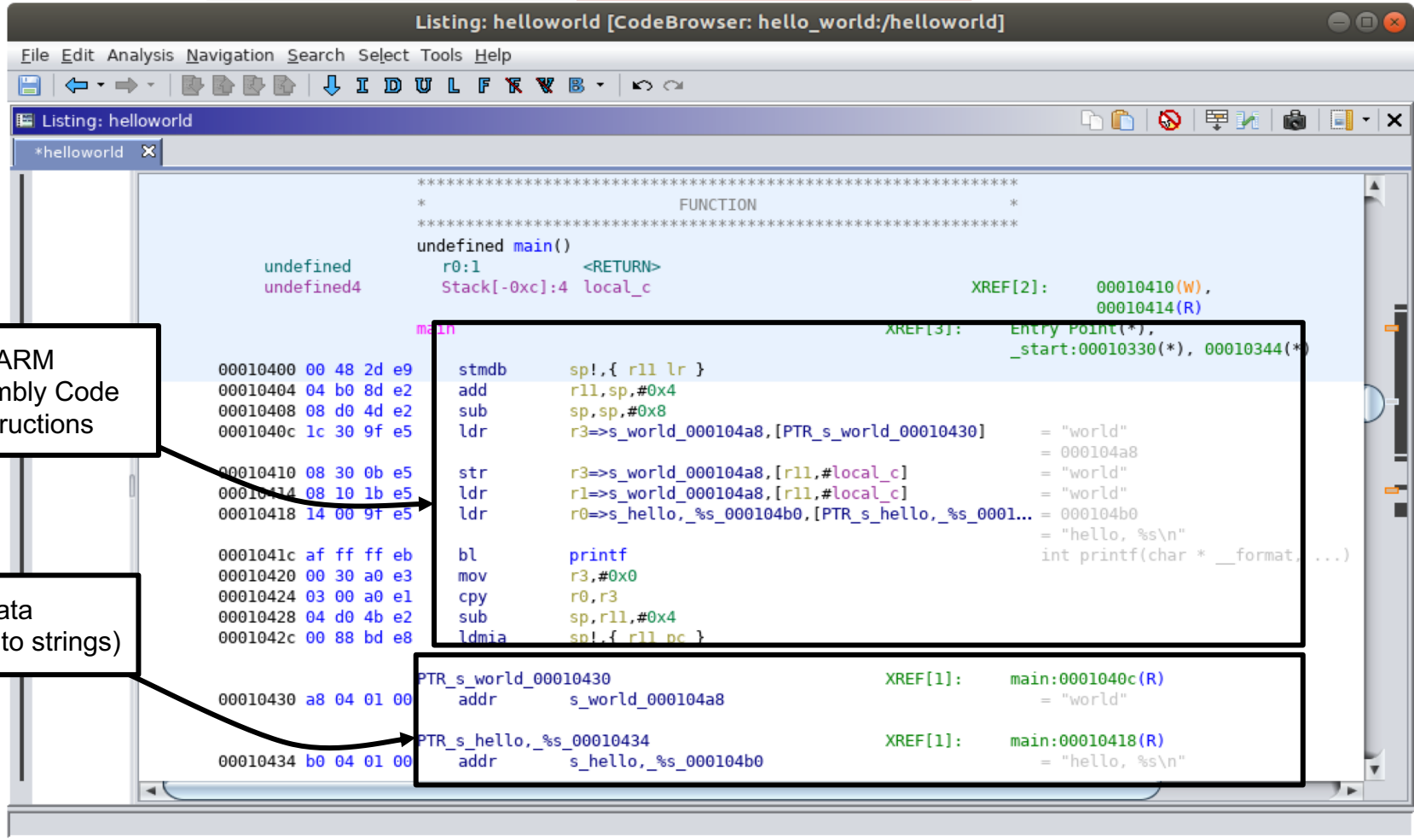
Memory Offsets



Ghidra's Disassembly (helloworld)

Disassembly

Decompilation | Disassembly



Ghidra's Disassembly (helloworld)

Disassembly Observations

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

```
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*

```
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`’

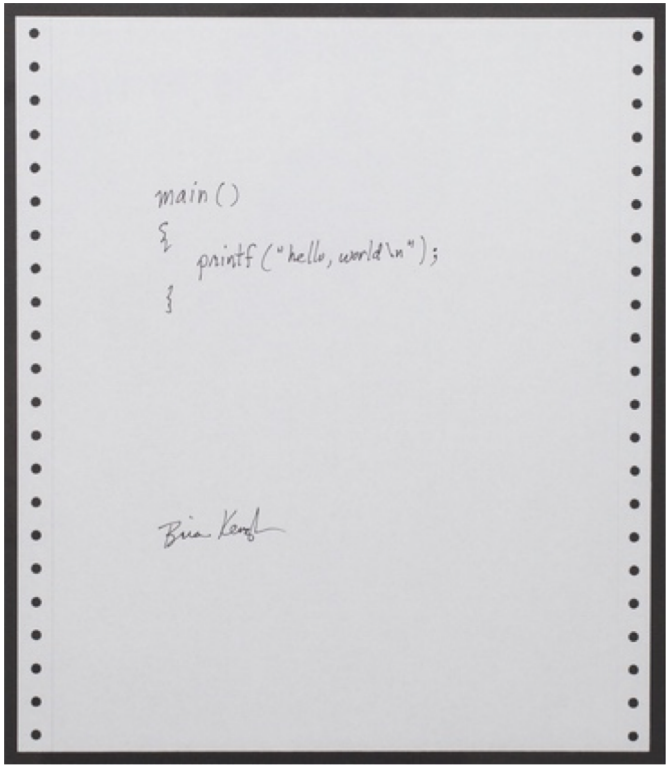
Decompilation

Decompilation | Disassembly

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

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.

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

#define NAME "world"

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

Original Source Code (helloworld.c)

Ghidra's Decompilation of the main function

Brian Kernighan, 1978, Public Domain
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

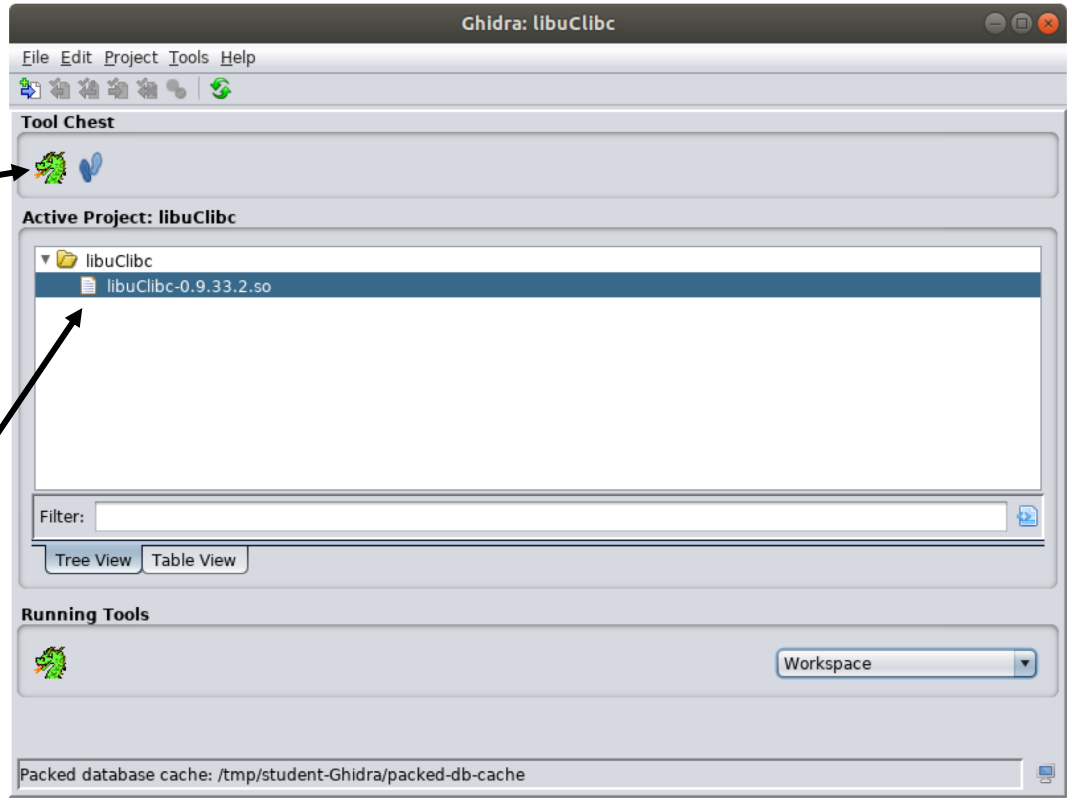
Refer to the glossary in the Ghidra Help for additional definitions

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**

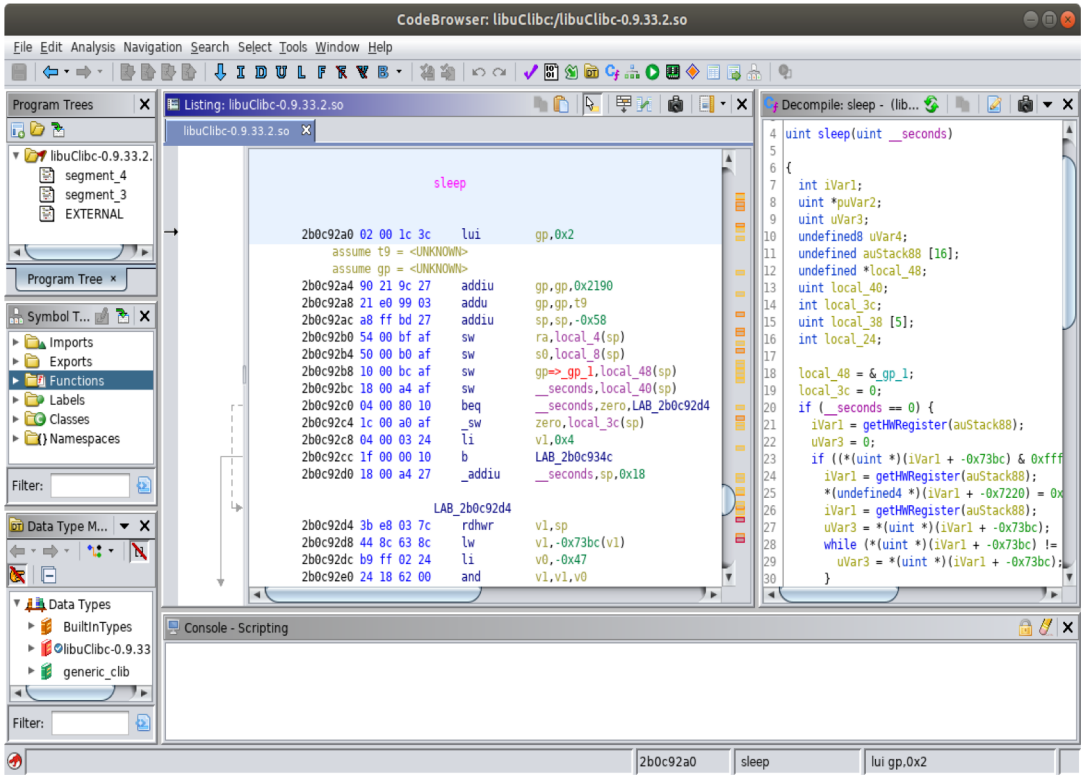
The Tool Chest shows available **tools** The dragon icon represents the Code Browser tool

The Active Project panel shows the programs loaded into the current project and available for analysis

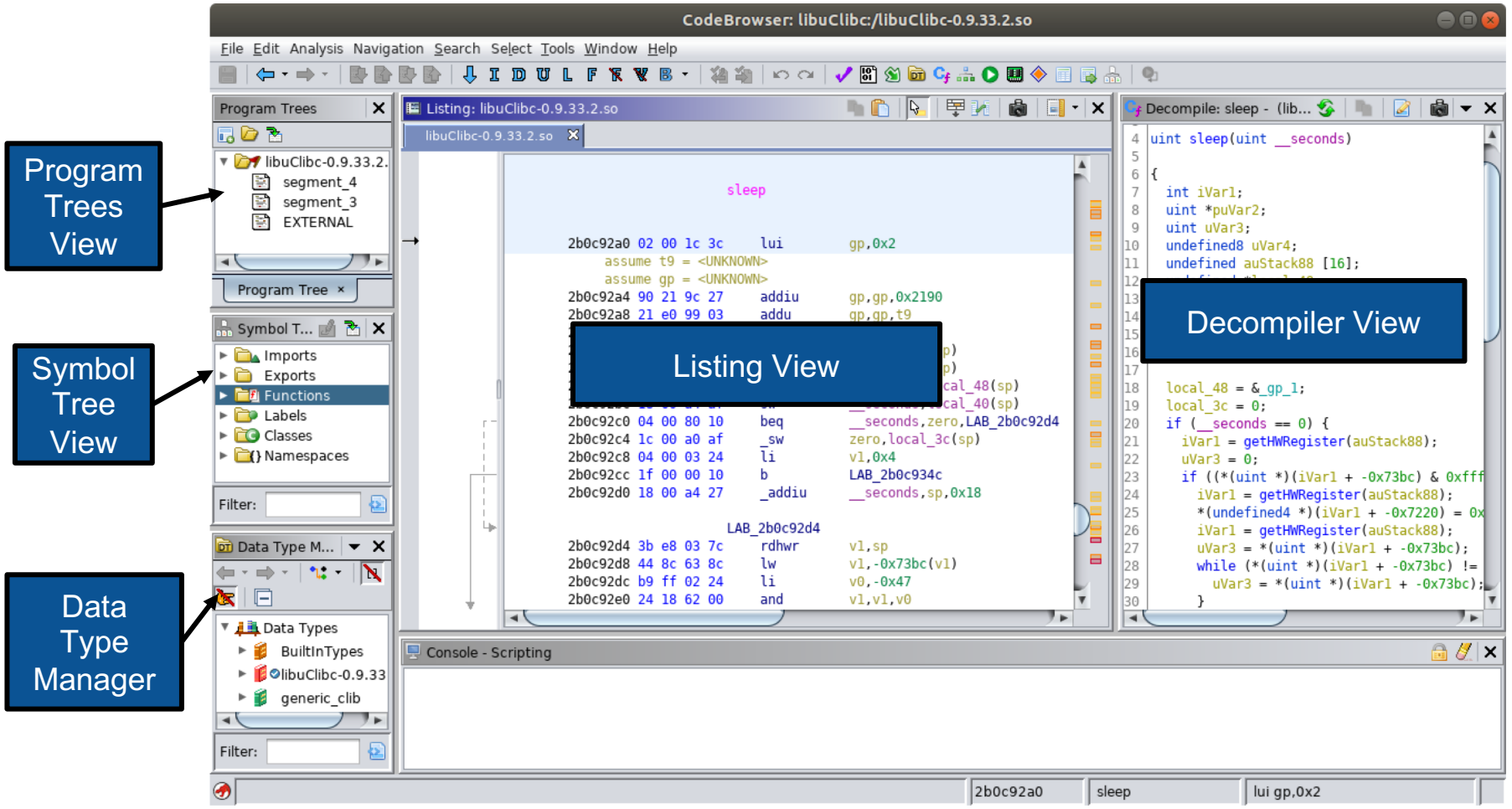


Code Browser Tool

- 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

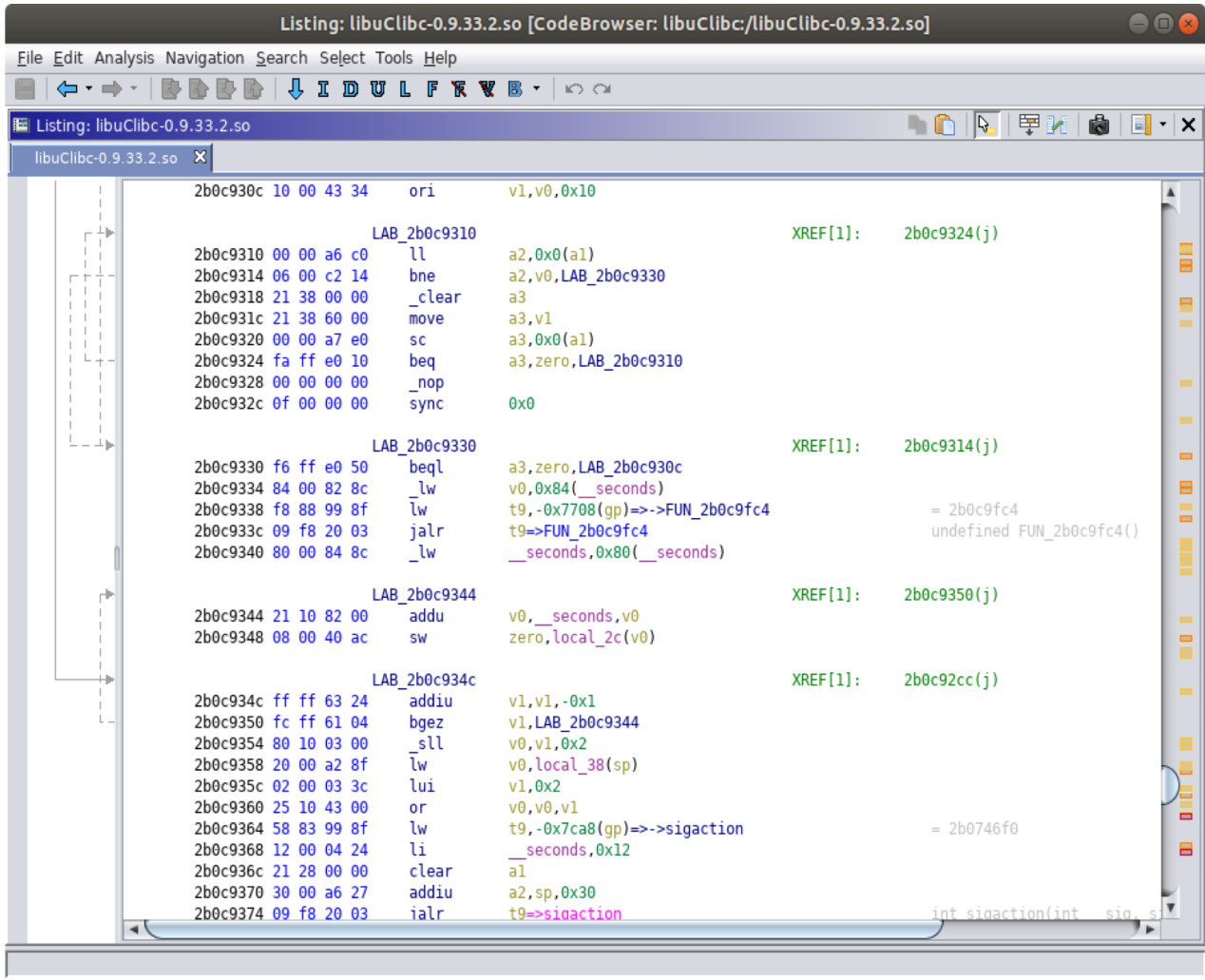


Code Browser Tool: Default Views



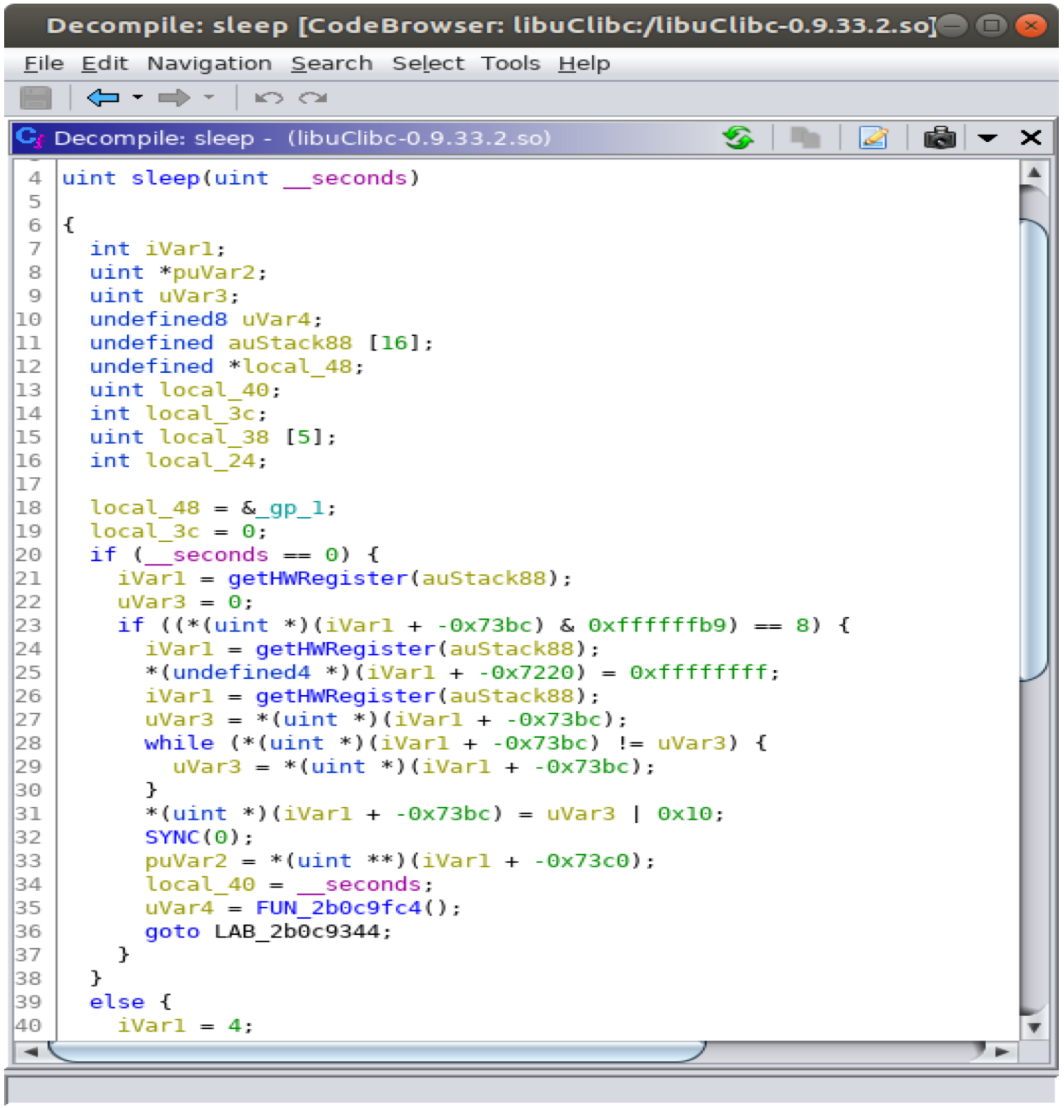
Listing View

- 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



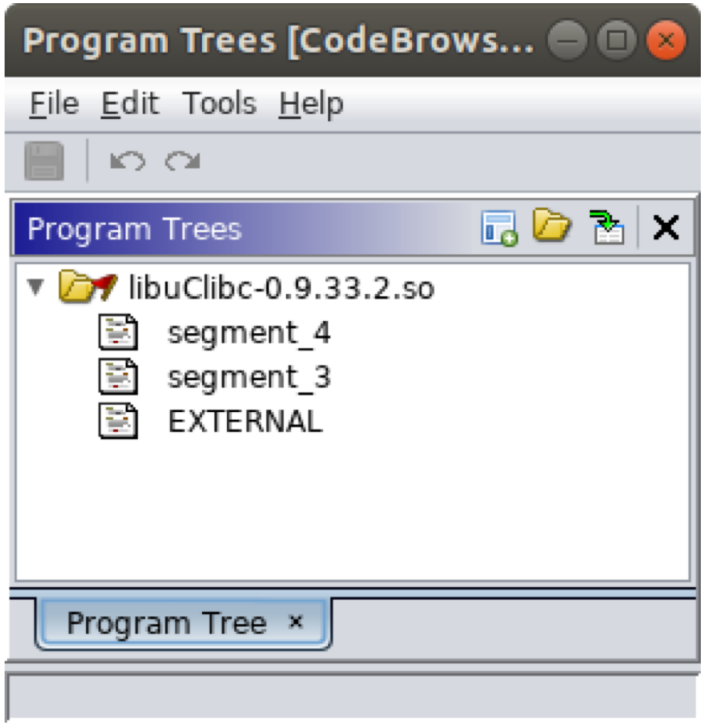
Decompiler View

- 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



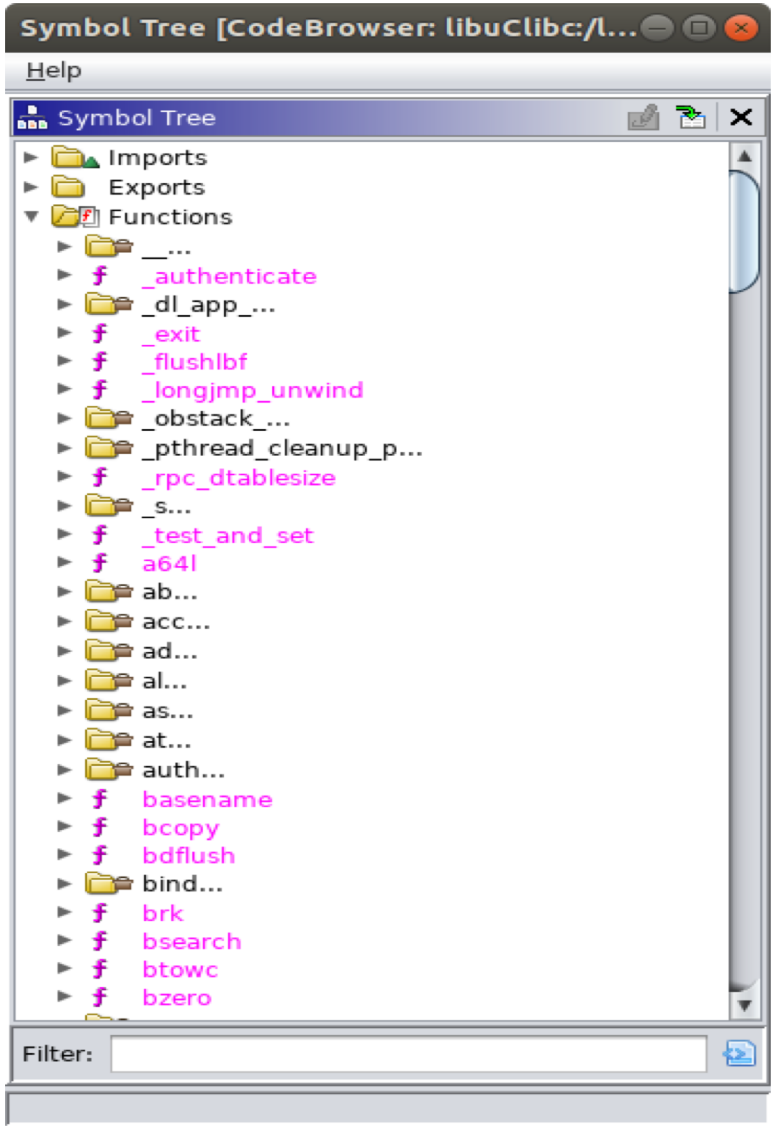
Program Trees View

- 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 cans can be organized into folders
  - Folders can also be nested within other folders



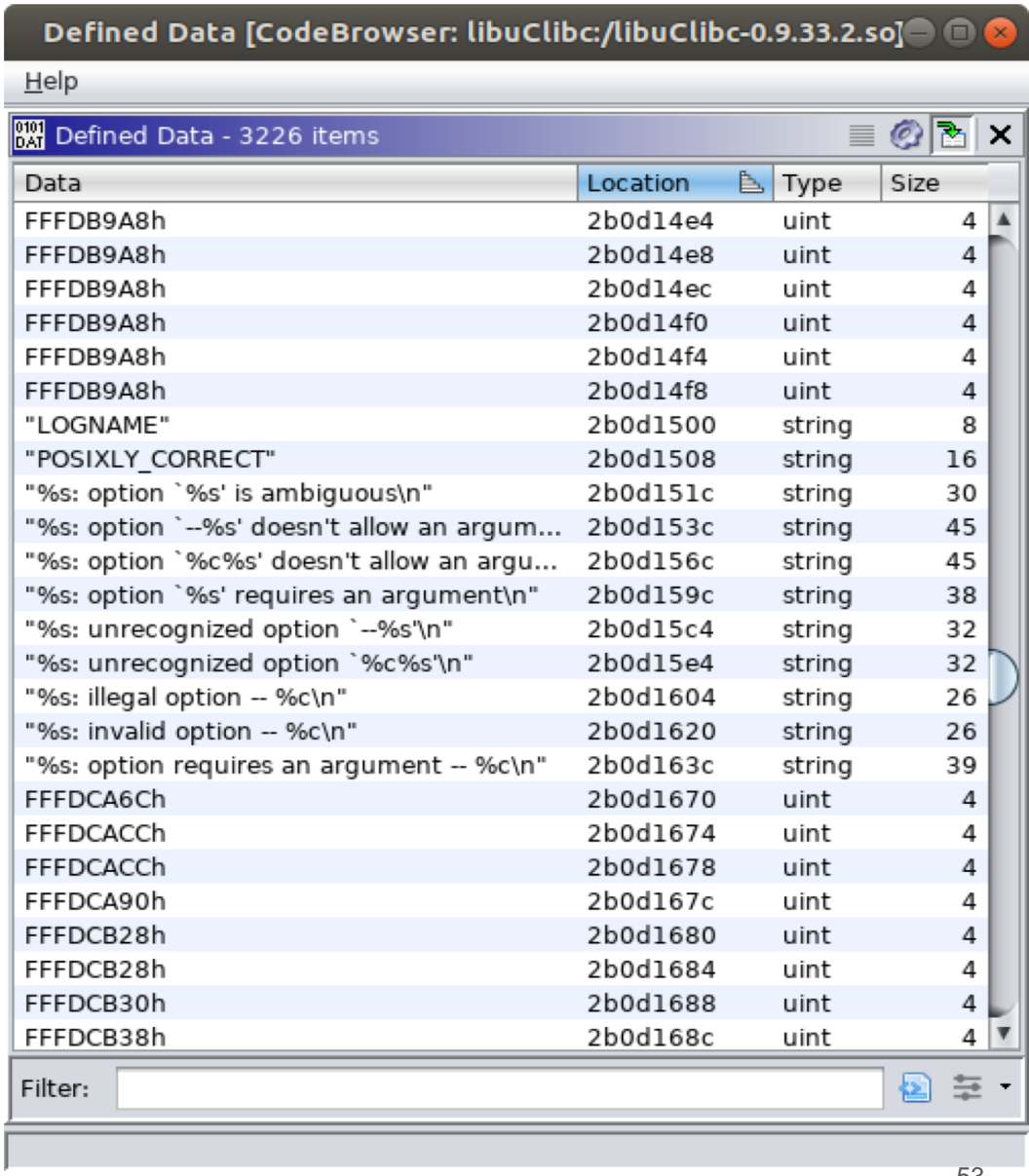
Symbol Tree View

- 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

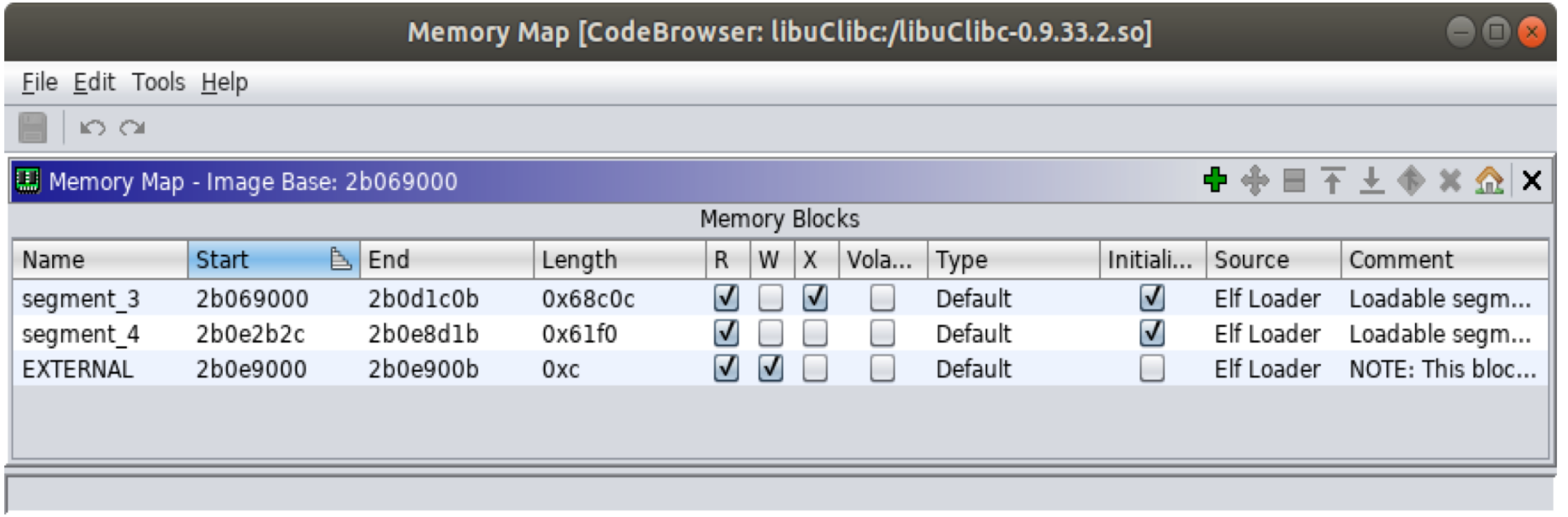
- Not shown by default, but can be added by selecting it from the Window menu in the Code Browser
- Shows data (e.g. strings, integers) defined within the program
  - e.g. Global or local variables
- This view also contains a filtering capability
- Can be especially helpful for reverse engineering programs,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



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 argum...	2b0d153c	string	45
"%s: option `%c%s' doesn't allow an argu...	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
FFFDCACCh	2b0d1674	uint	4
FFFDCACCh	2b0d1678	uint	4
FFFDCA90h	2b0d167c	uint	4
FFFDCB28h	2b0d1680	uint	4
FFFDCB28h	2b0d1684	uint	4
FFFDCB30h	2b0d1688	uint	4
FFFDCB38h	2b0d168c	uint	4

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



Name	Start	End	Length	R	W	X	Vola...	Type	Initiali...	Source	Comment
segment_3	2b069000	2b0d1c0b	0x68c0c	☑	☐	☑	☐	Default	☑	Elf Loader	Loadable segm...
segment_4	2b0e2b2c	2b0e8d1b	0x61f0	☑	☐	☐	☐	Default	☑	Elf Loader	Loadable segm...
EXTERNAL	2b0e9000	2b0e900b	0xc	☑	☑	☐	☐	Default	☐	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.

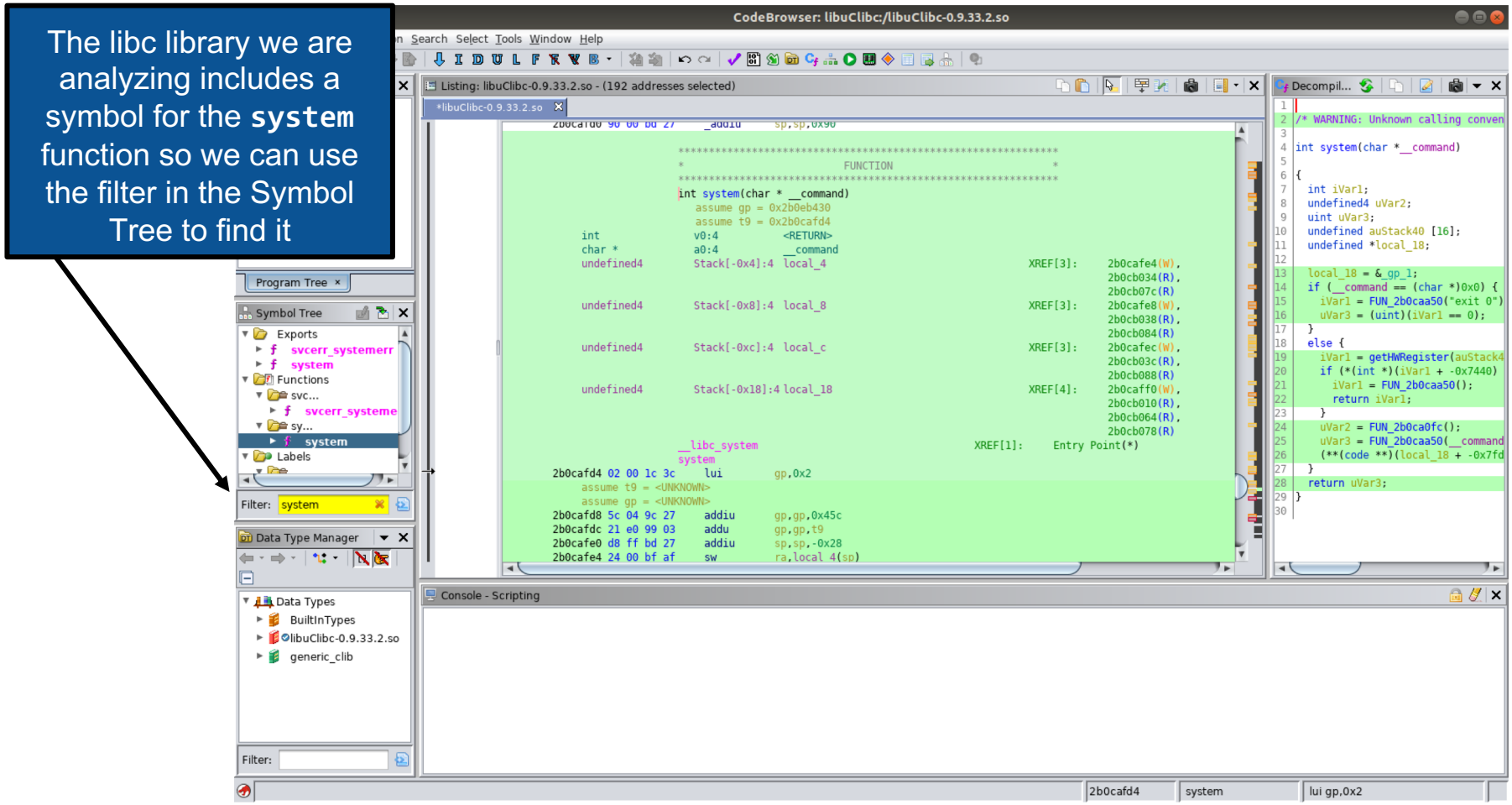
Using Ghidra to find ROP Gadgets

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`)

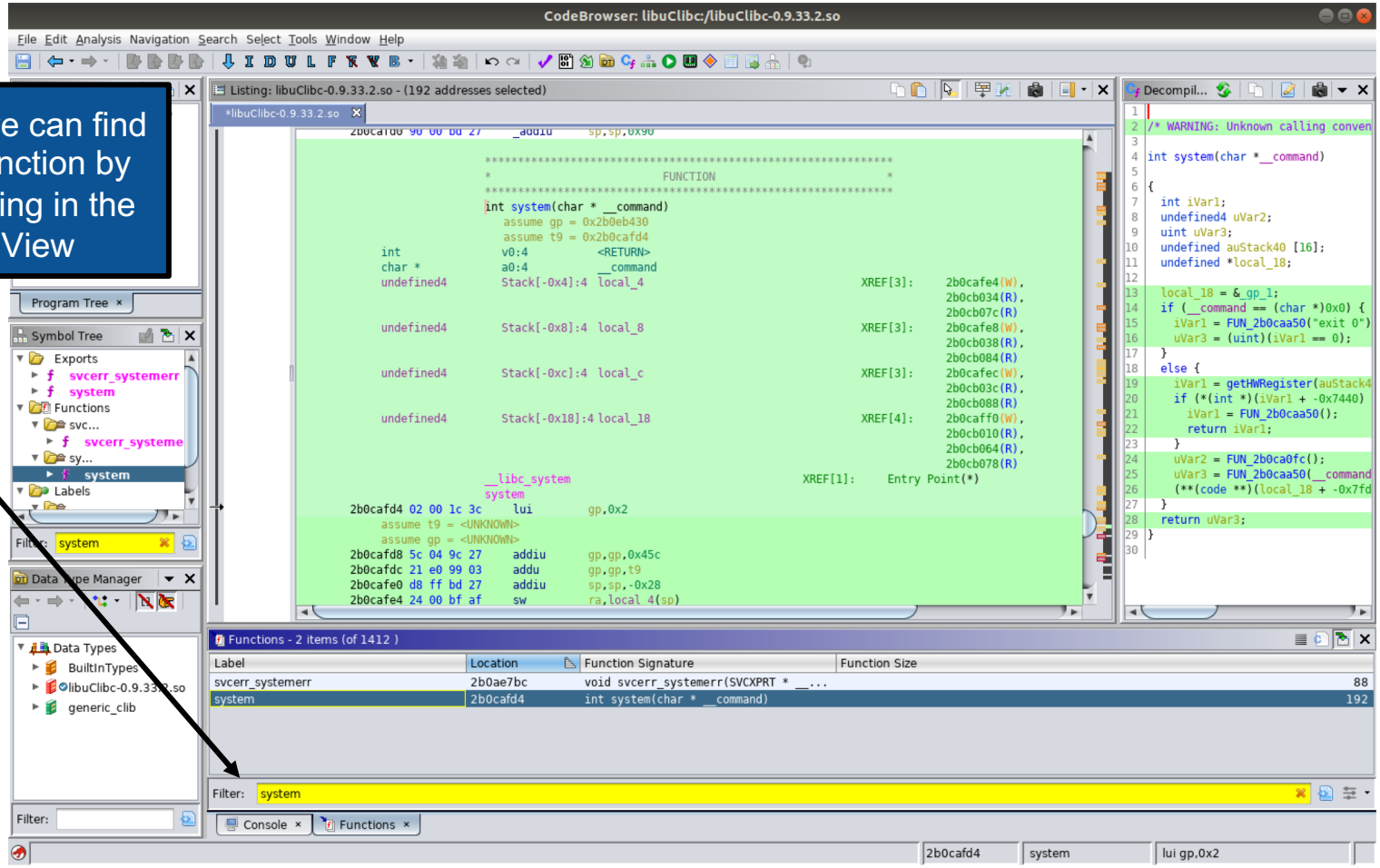
Finding the Address of the `system` Function in libc

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

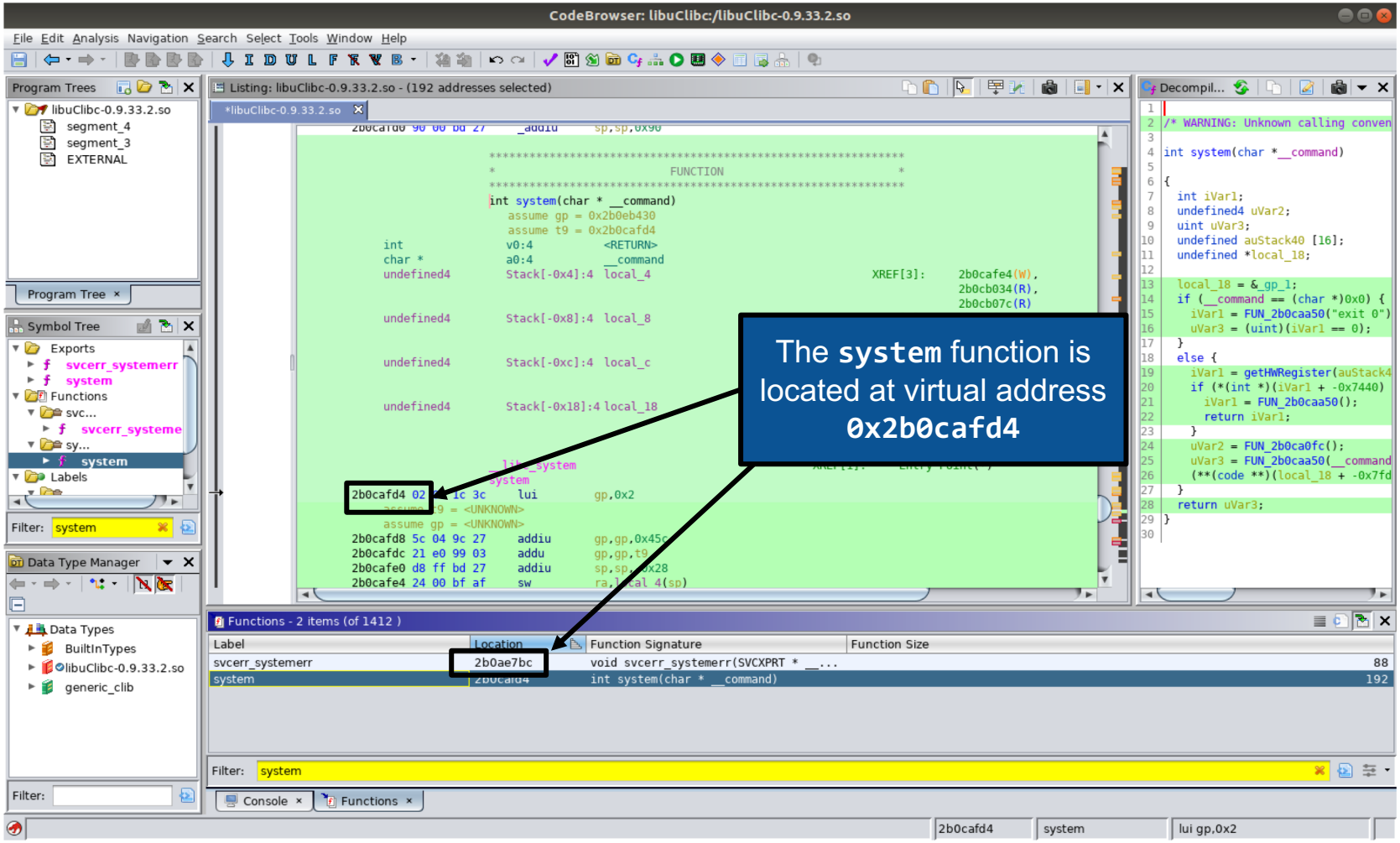


Finding the Address of the `system` Function in libc

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



Address of the `system` Function in libc



Dealing with Bad Characters

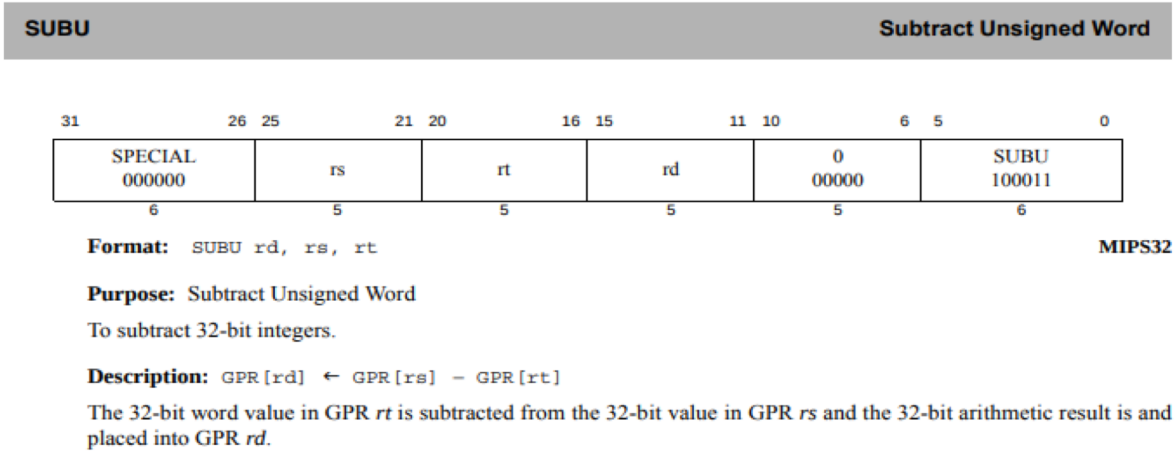
- We mentioned that our target program accepts input via an UPnP interface (SOAP over HTTP)
  - This protocol constrains the values of the input characters significantly
  - Supported characters are:
    - `0x09` (Horizontal Tab)
    - `0x0a` (Line Feed)
    - `0x0d` (Carriage Return)
    - `0x20-0x7e` (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 `0x2b0cafd4`, 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 `0x2b0cafd4` 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 `0x2b0cafd4` 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
6	5	5	5	5	6

Format: `SUBU rd, rs, rt` **MIPS32**

Purpose: Subtract Unsigned Word

To subtract 32-bit integers.

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

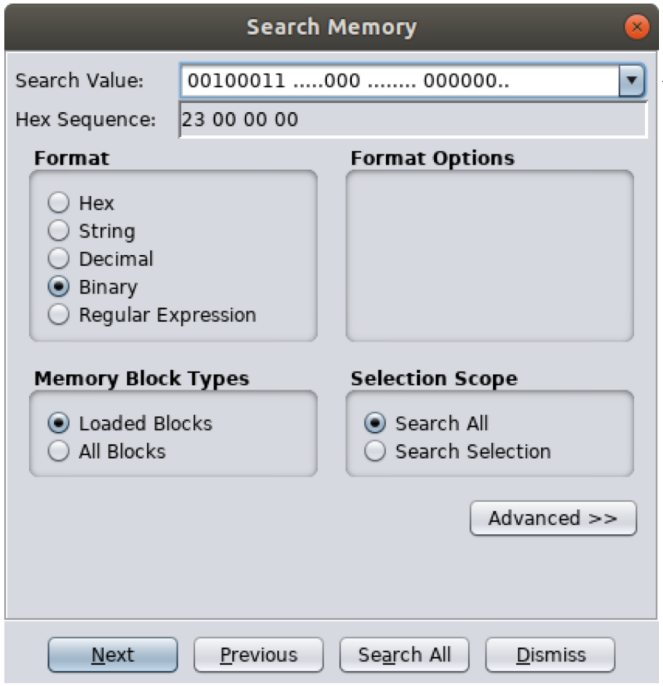
The 32-bit word value in GPR *rt* is subtracted from the 32-bit value in GPR *rs* and the 32-bit arithmetic result is and placed into GPR *rd*.

The MIPS32® Instruction Set Manual [9]

- `0x4b2d2e52 - 0x20207e7e = 0x2b0cafd4`

Finding a SUBU Instruction

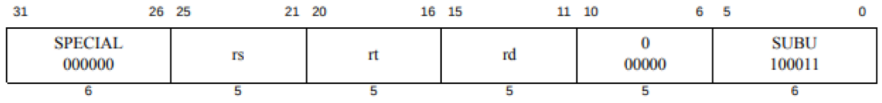
- Ghidra provides a memory search feature we can use to find instances of machine code instructions independent of the specific operands
- The ‘Search Memory’ dialog may be launched by choosing ‘Search’ → ‘Memory…’ from the Ghidra CodeBrowser window



The dots in the search value will match any bit value

The dots are placed where registers are encoded in the SUBU instruction

SUBU — **Subtract Unsigned Word**



Format: `SUBU rd, rs, rt` **MIPS32**

Purpose: Subtract Unsigned Word

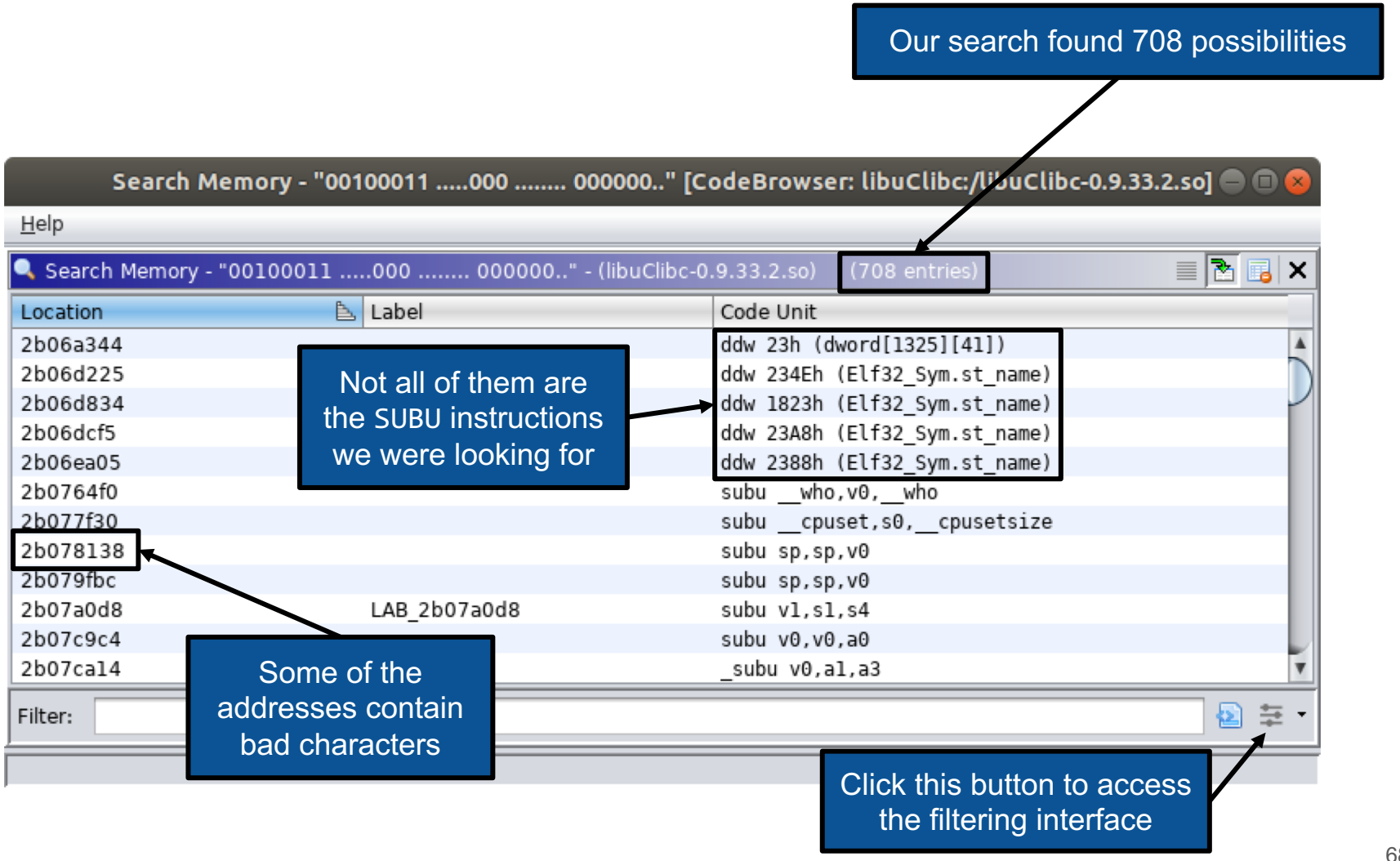
To subtract 32-bit integers.

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 and placed into GPR *rd*.

The MIPS32® Instruction Set Manual [9]

Finding a SUBU Instruction

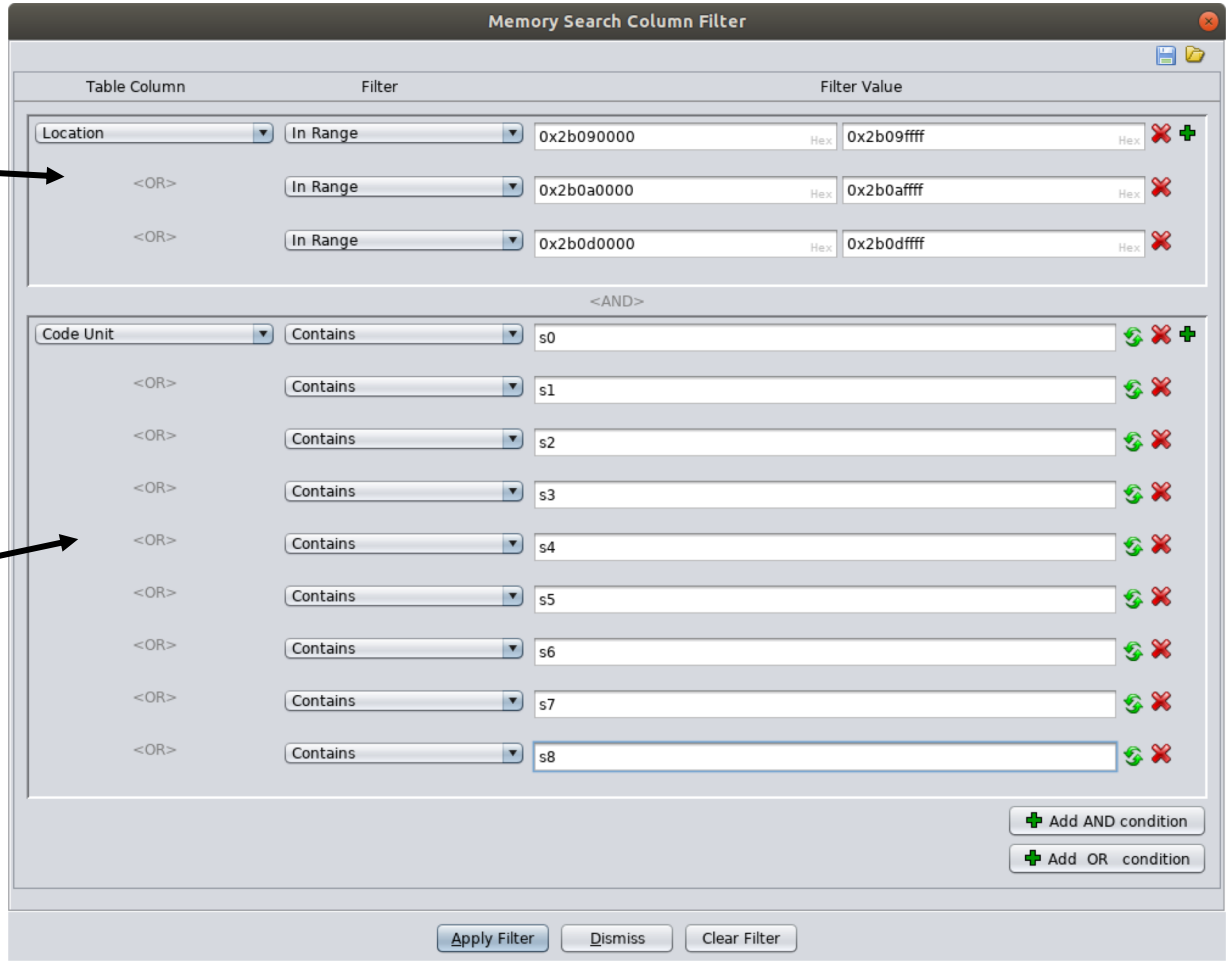


Finding a SUBU Instruction

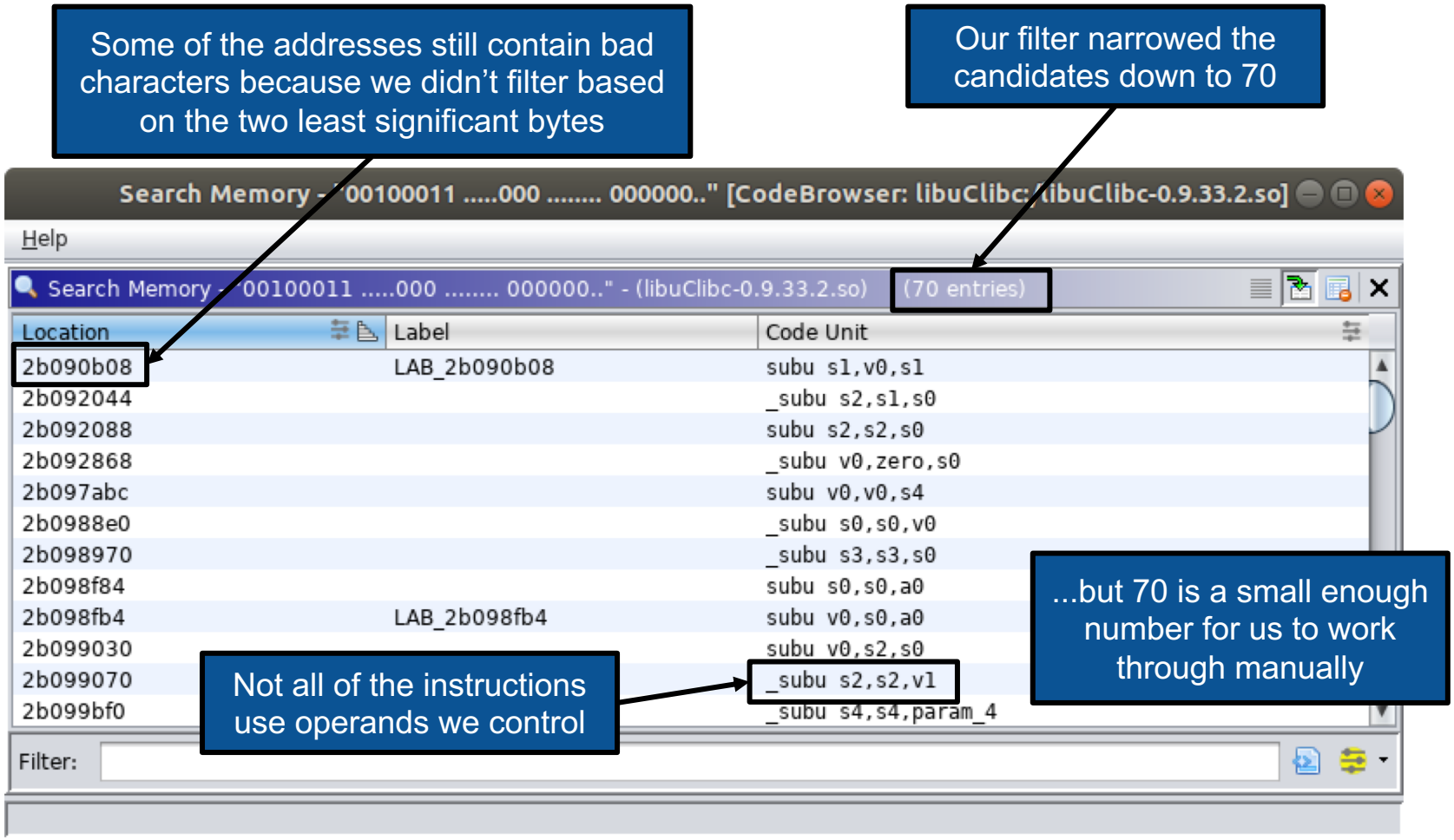
We can filter by the location of the instruction

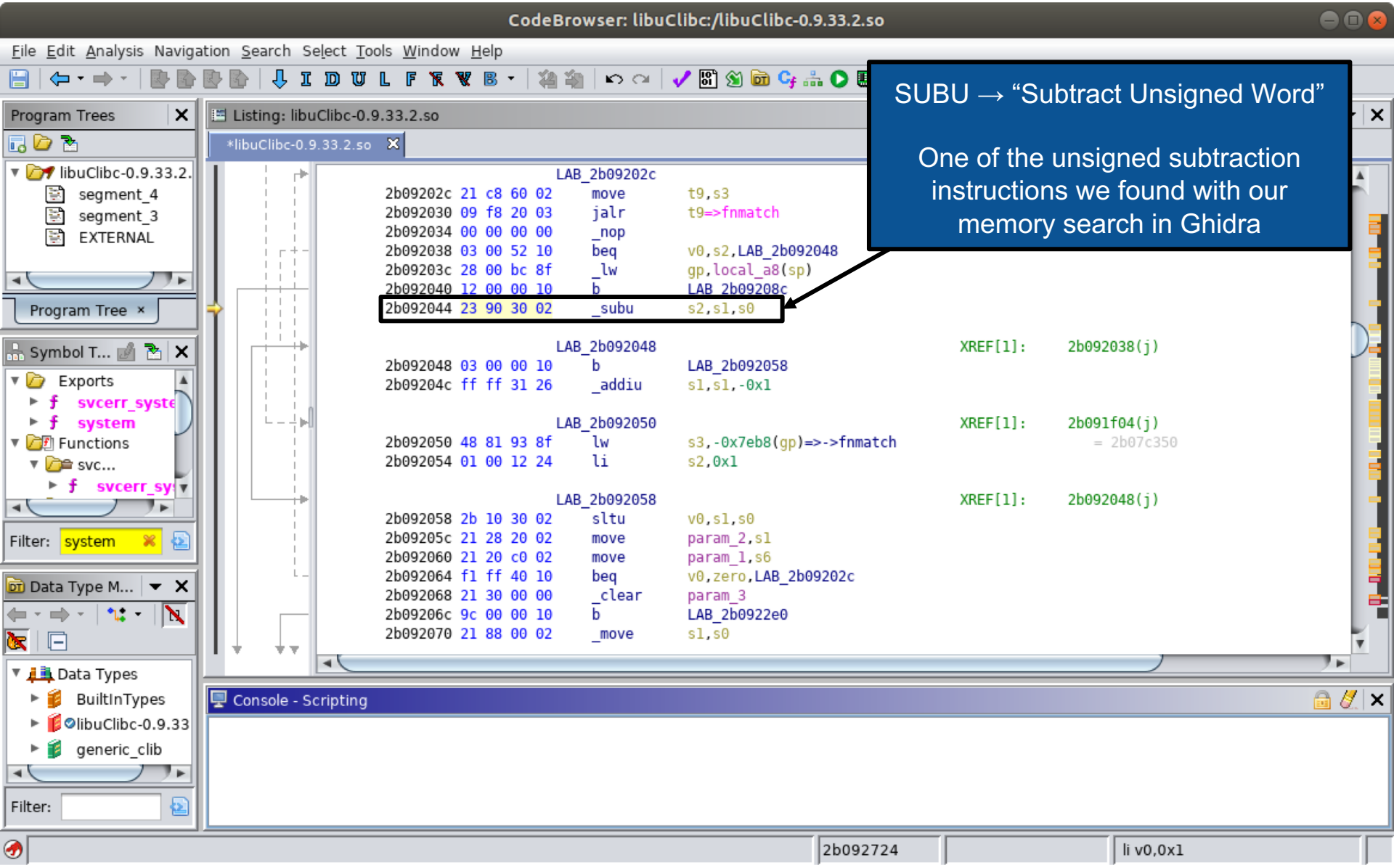
We can also filter by strings in the assembly instructions

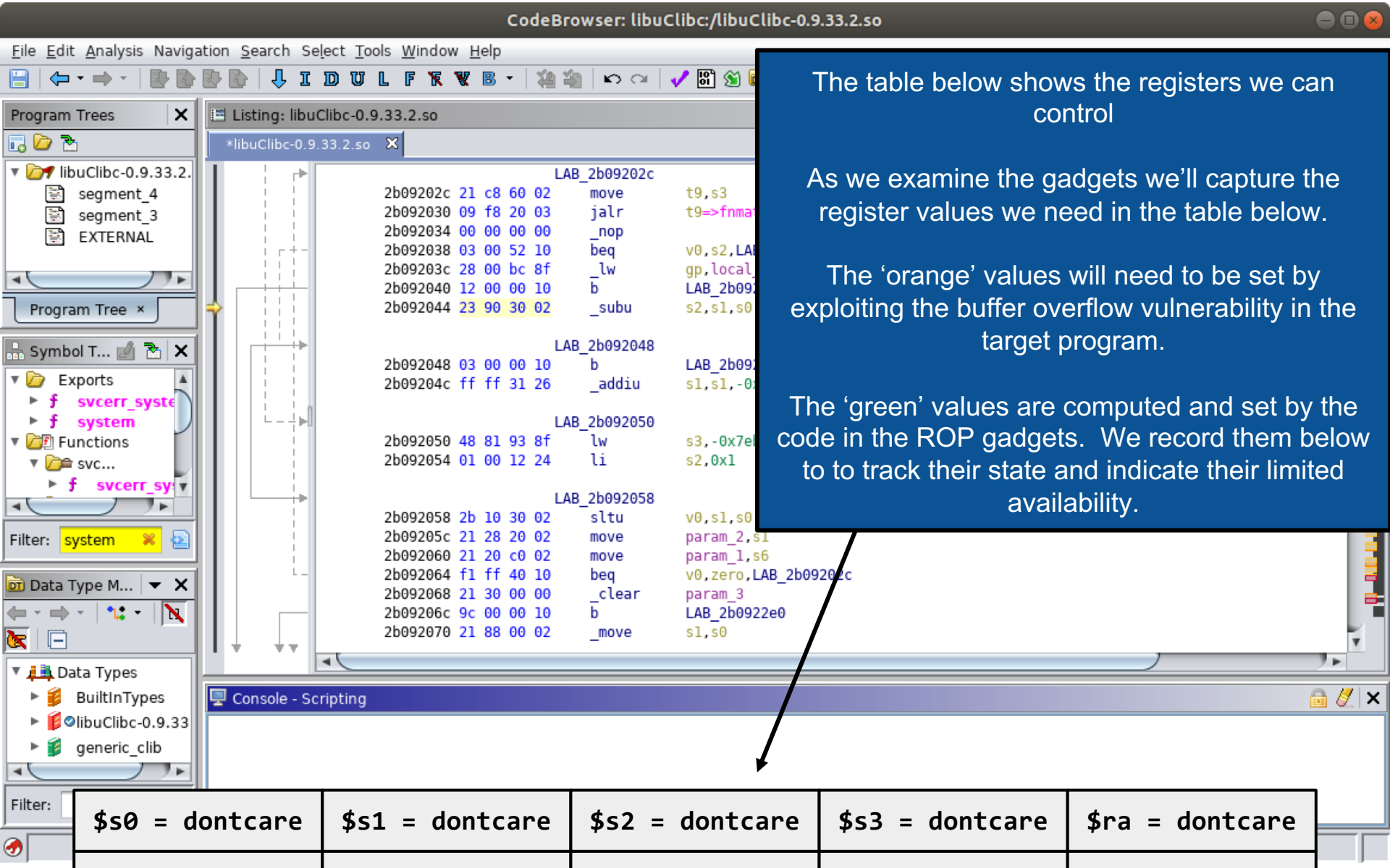
Here we look for instructions that use registers we control as operands



Finding a SUBU Instruction







The table below shows the registers we can control

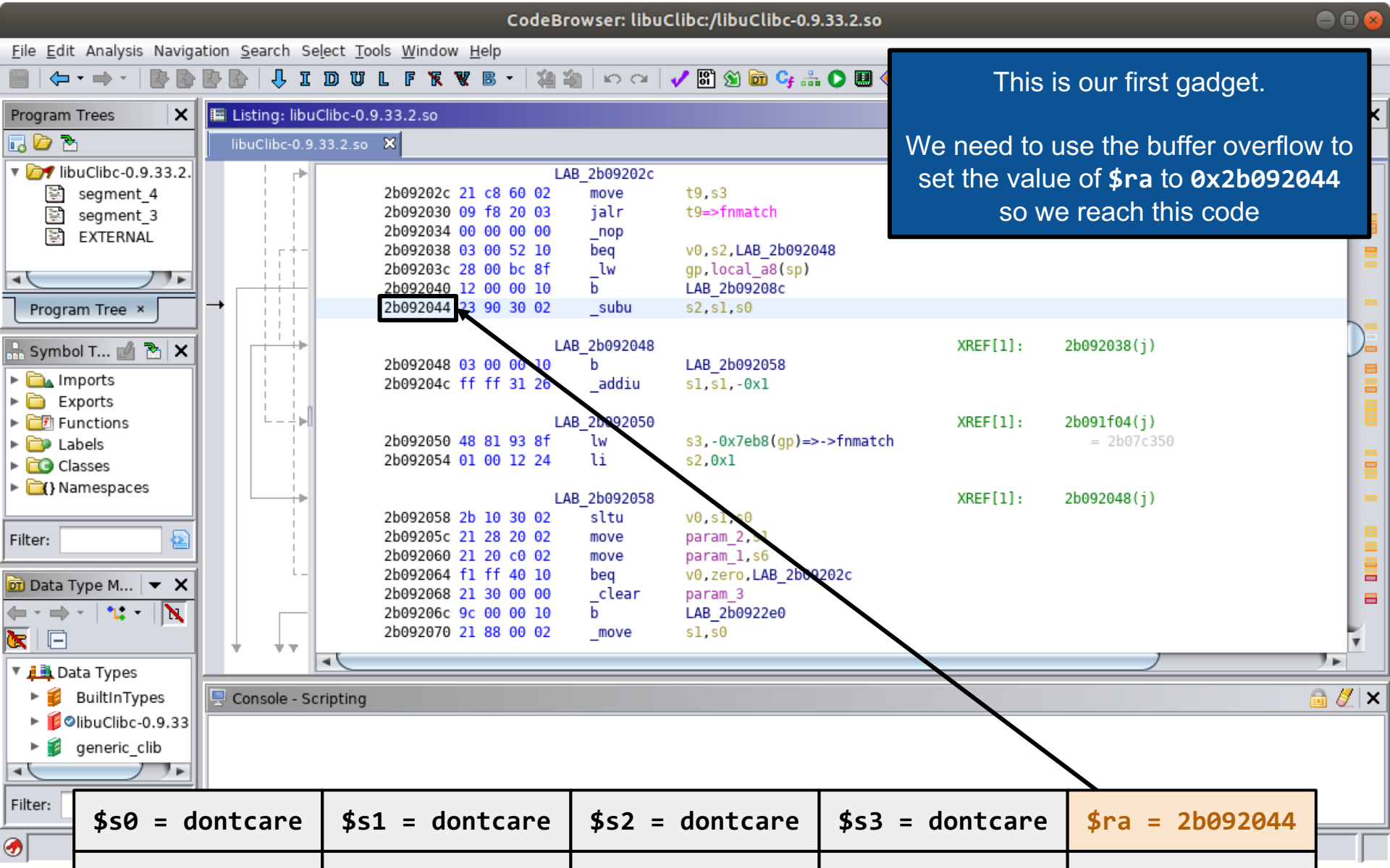
As we examine the gadgets we'll capture the register values we need in the table below.

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 to track their state and indicate their limited availability.

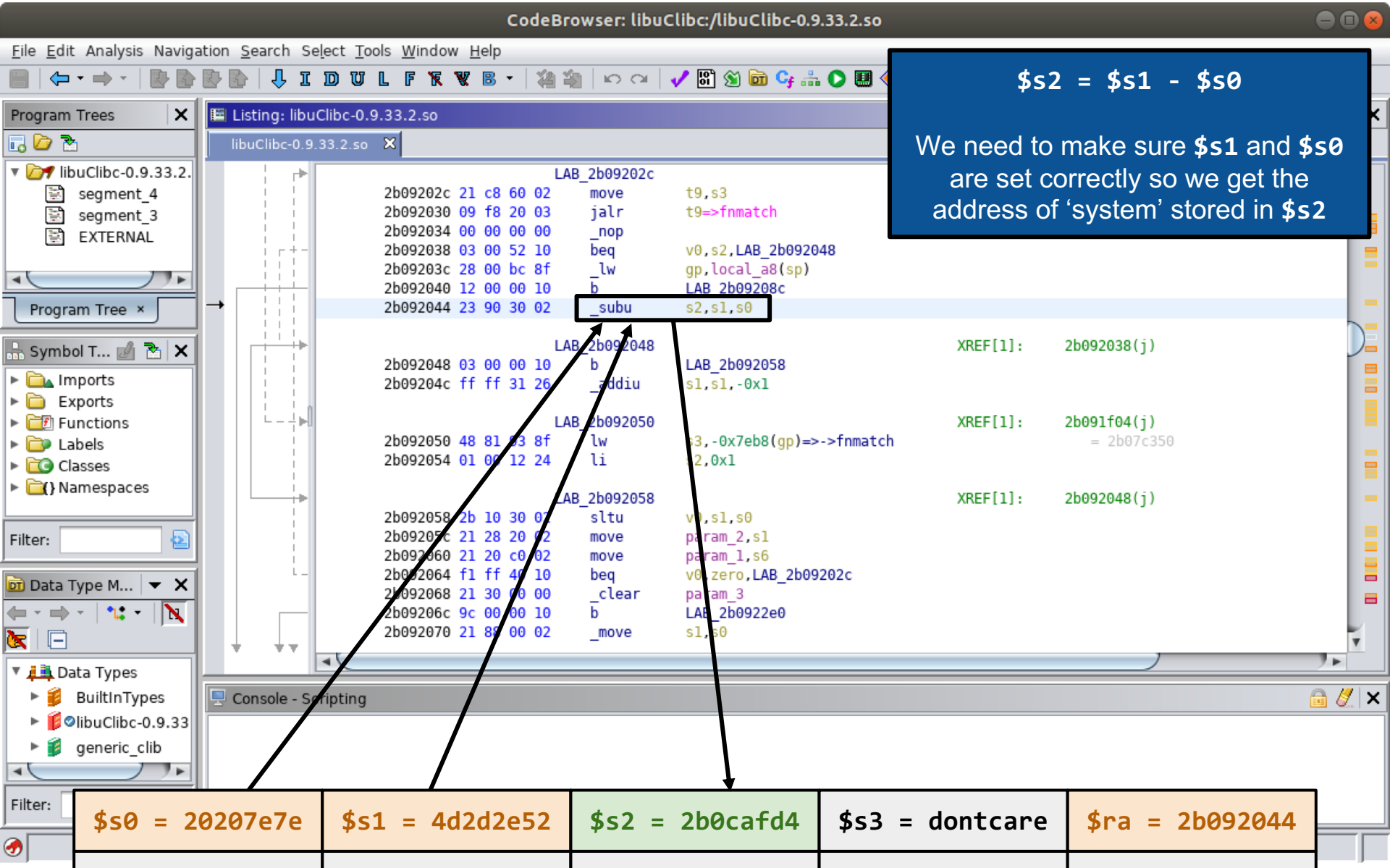
$s0 = dontcare	$s1 = dontcare	$s2 = dontcare	$s3 = dontcare	$ra = dontcare
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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



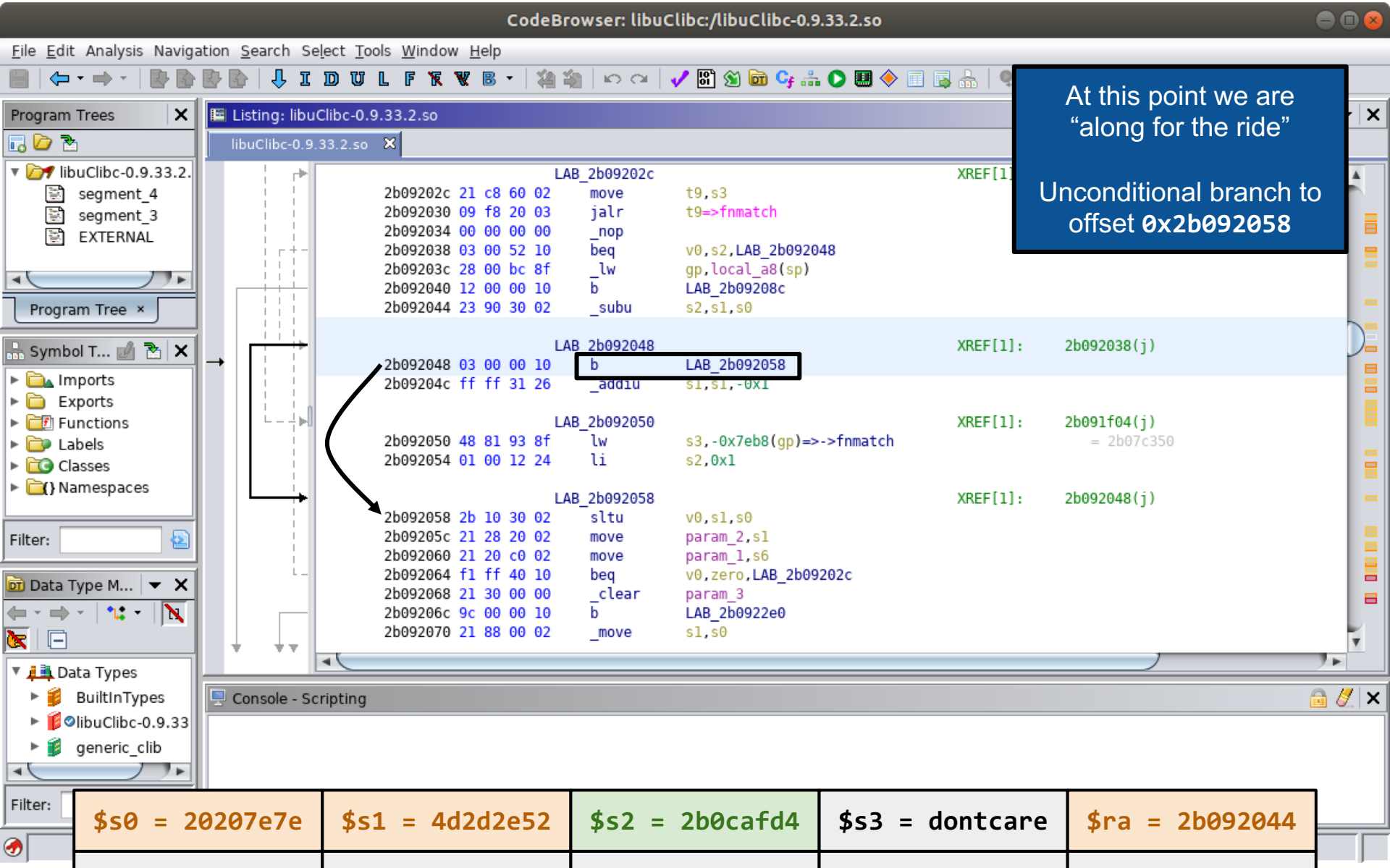
$s0 = dontcare	$s1 = dontcare	$s2 = dontcare	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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



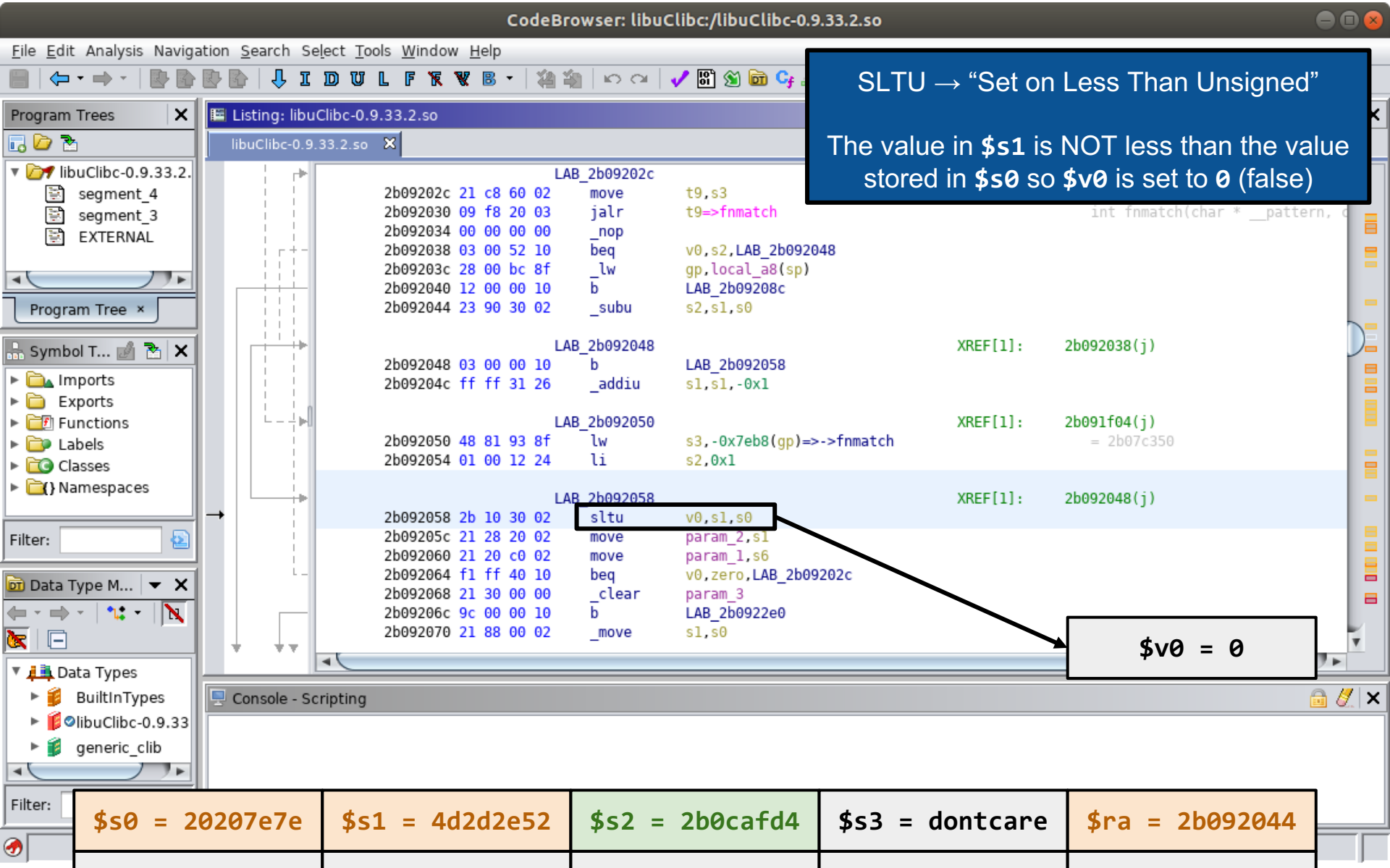
$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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



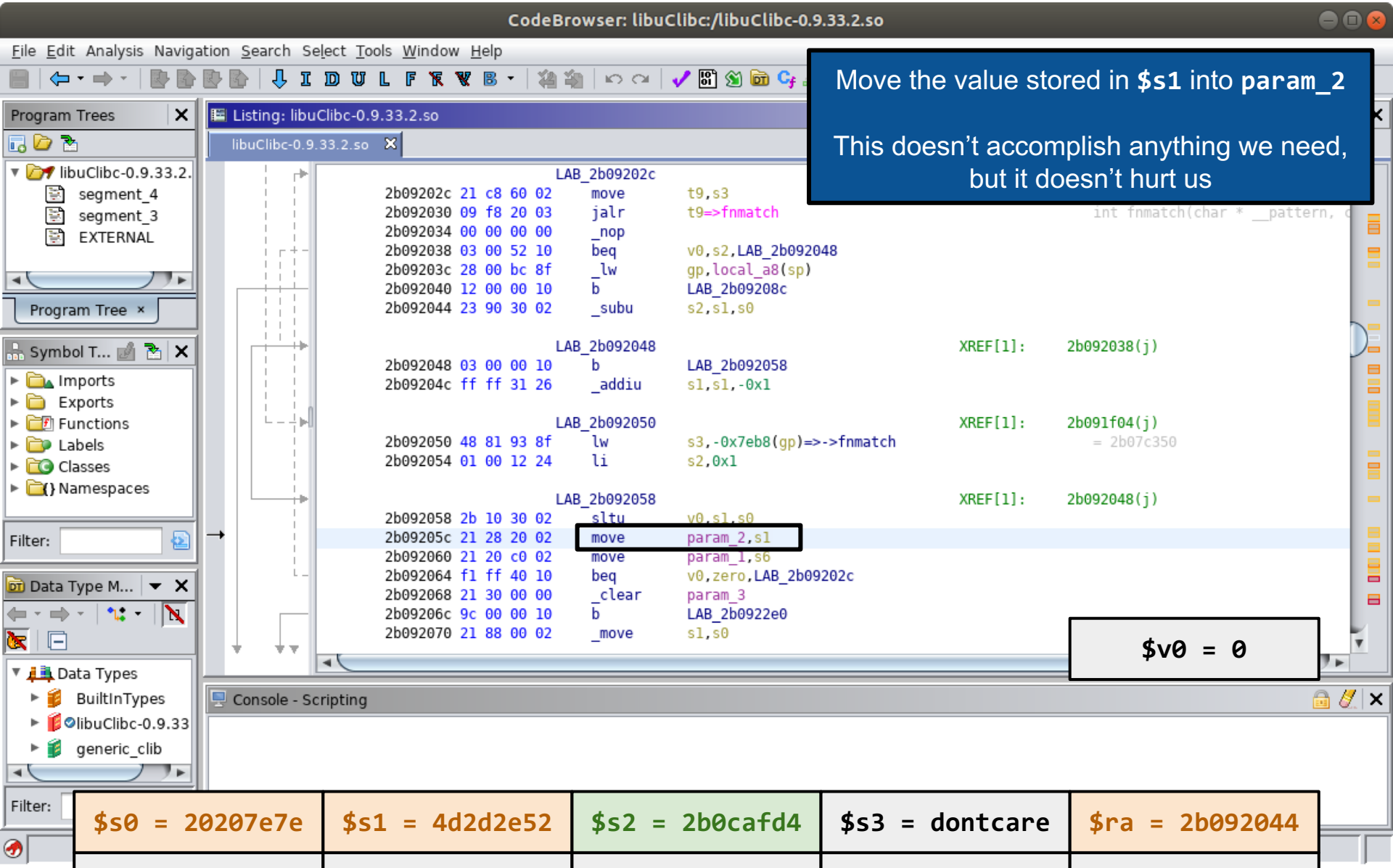
$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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



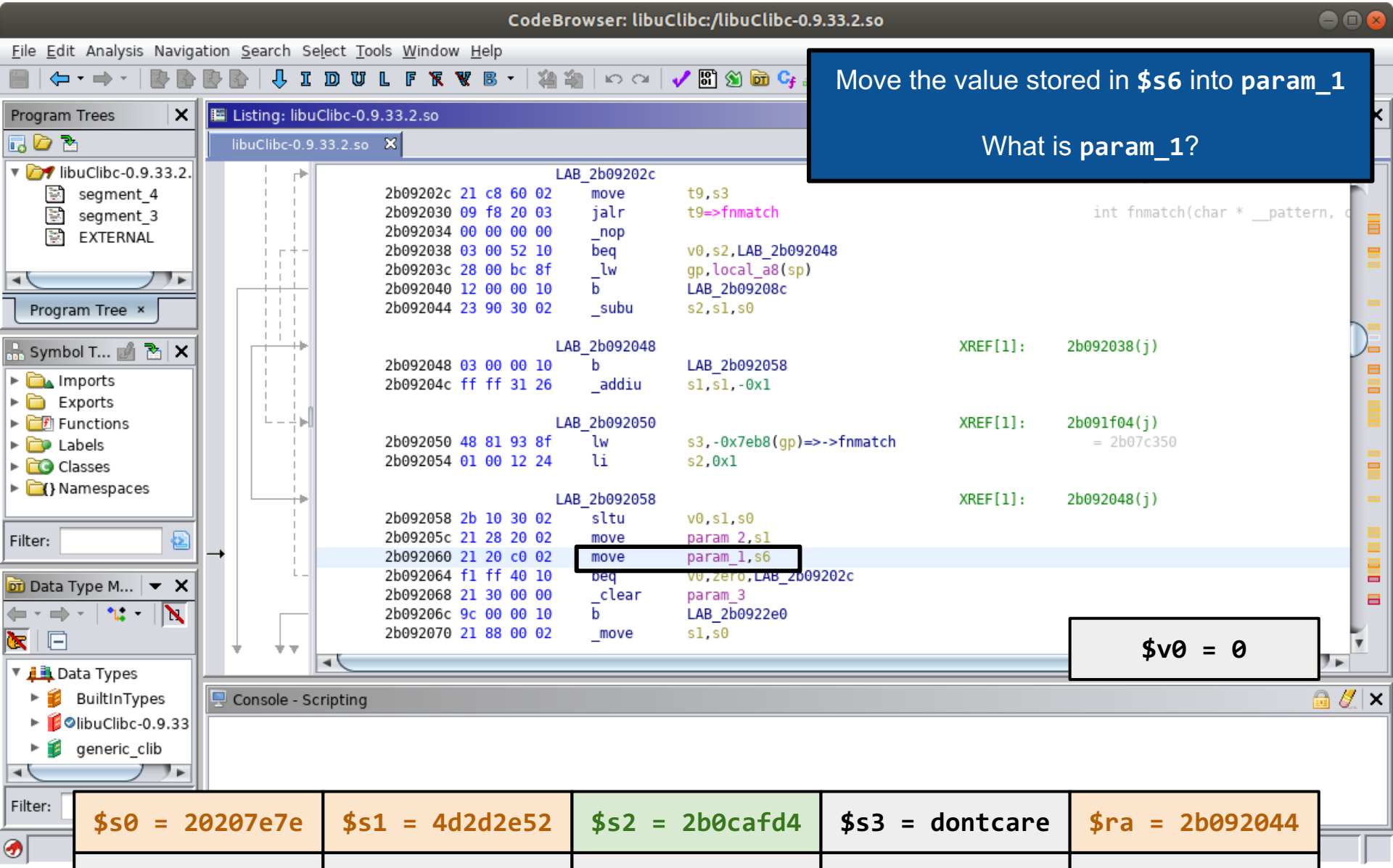
$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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



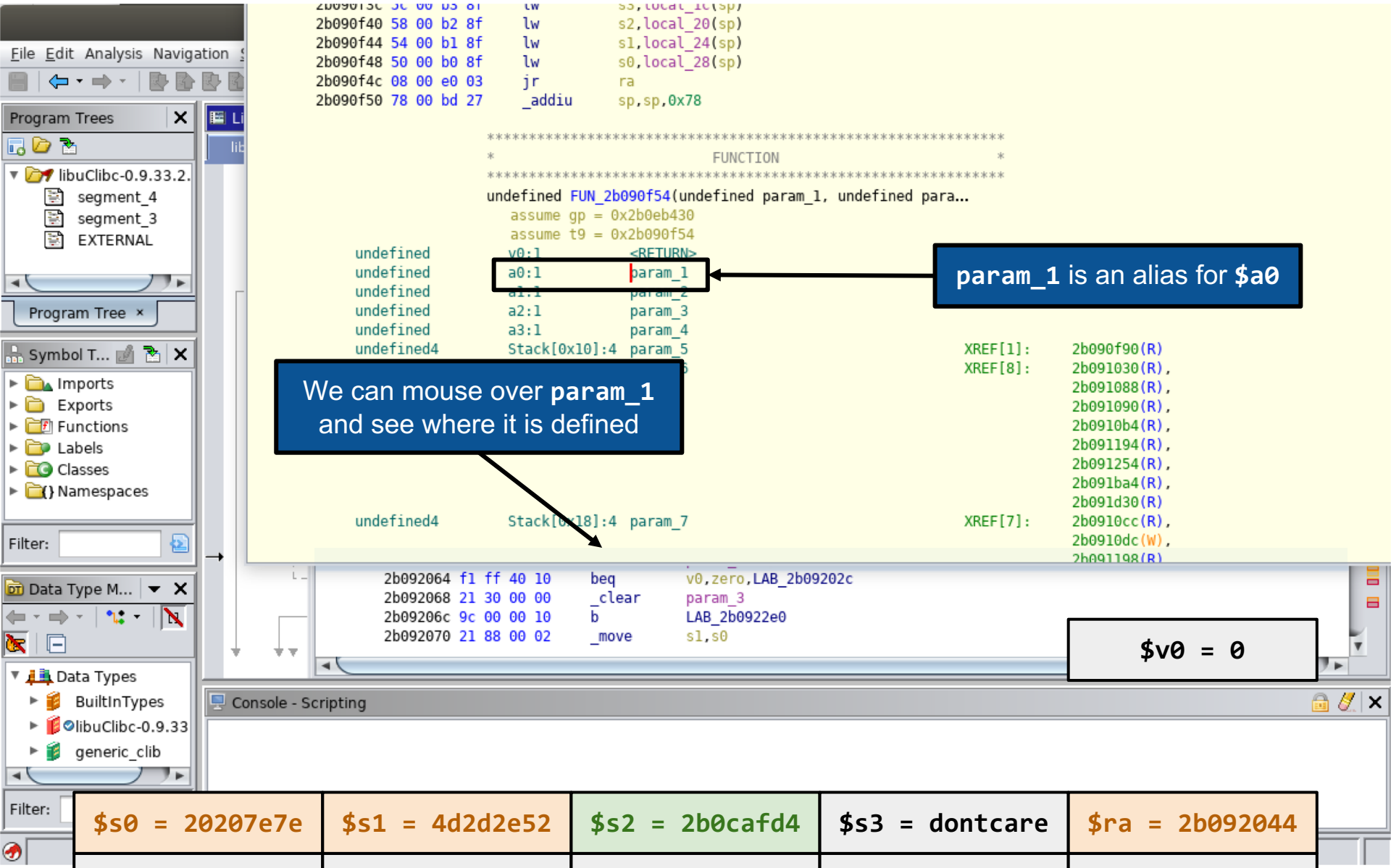
$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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



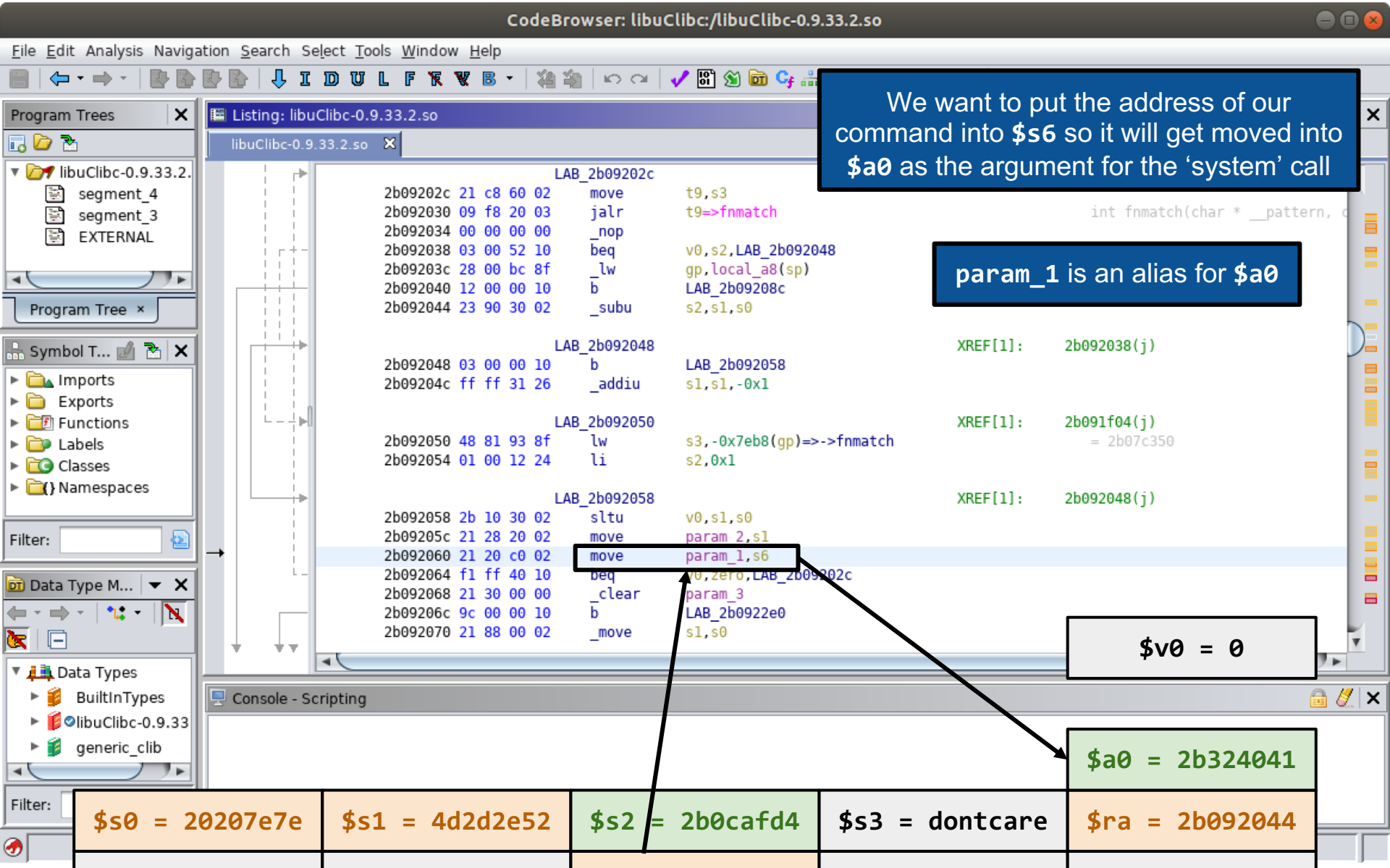
$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

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

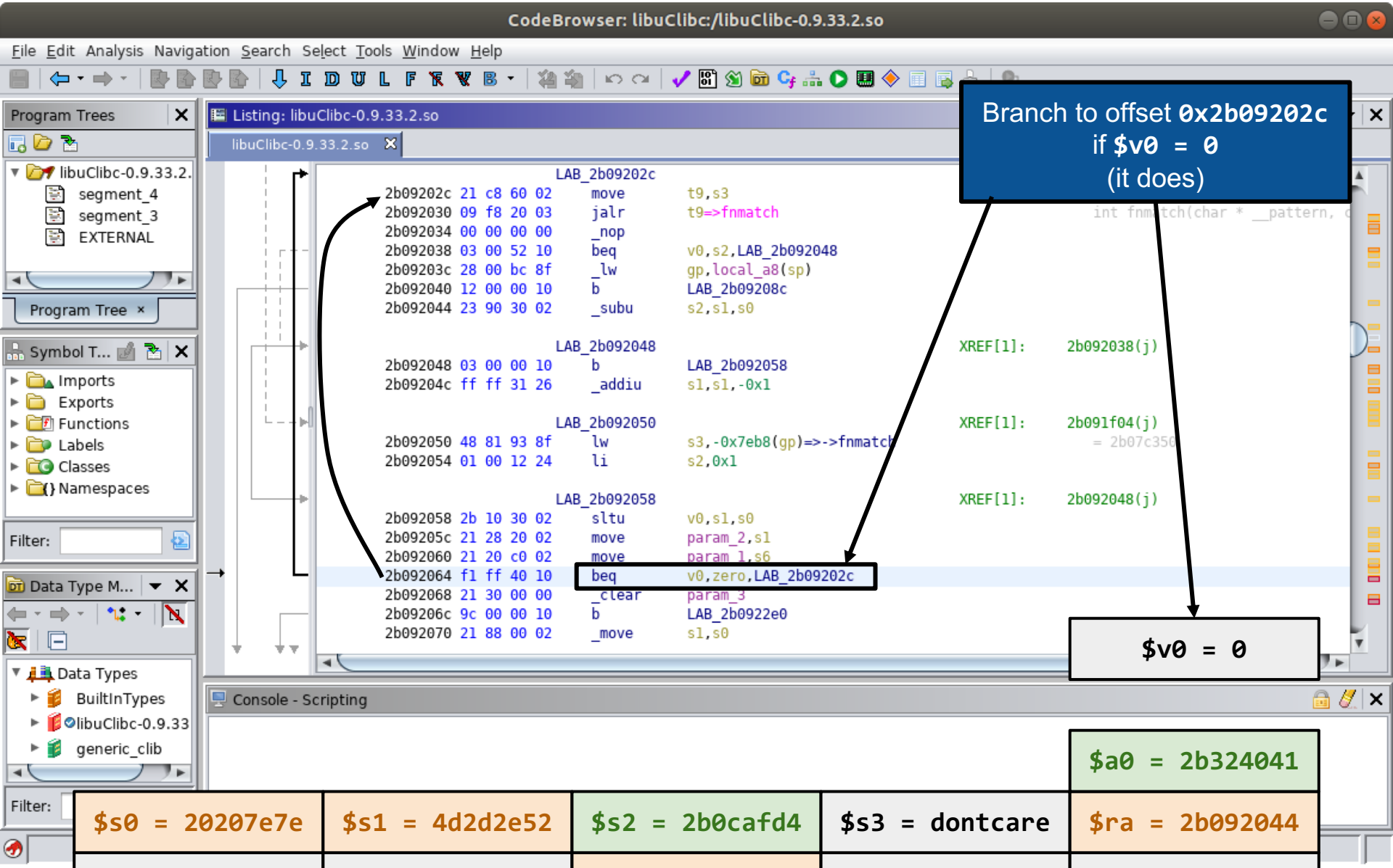


$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = dontcare	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = dontcare	$s7 = dontcare	$s8 = dontcare

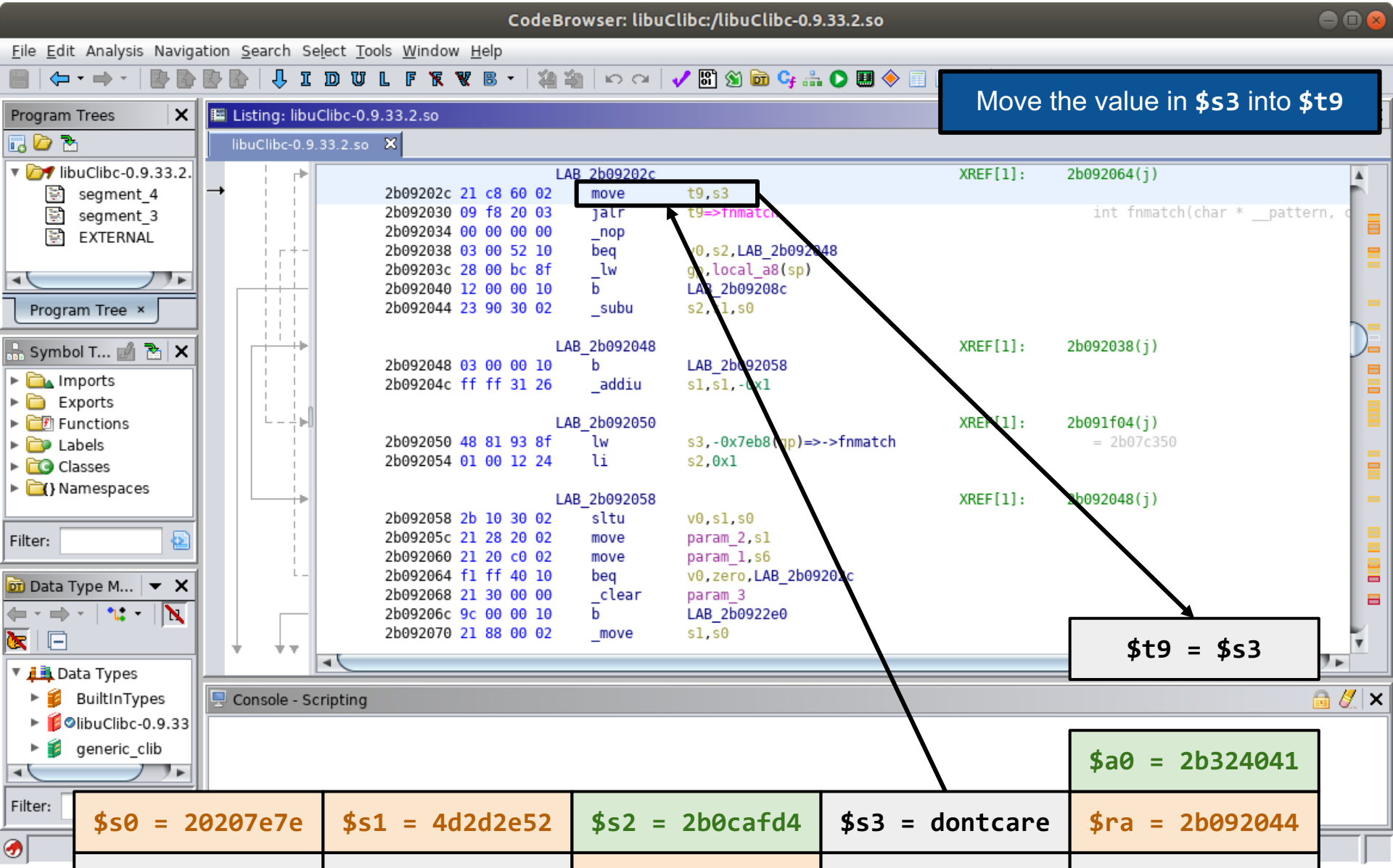
Orange → Set this value **Green** → Computed value



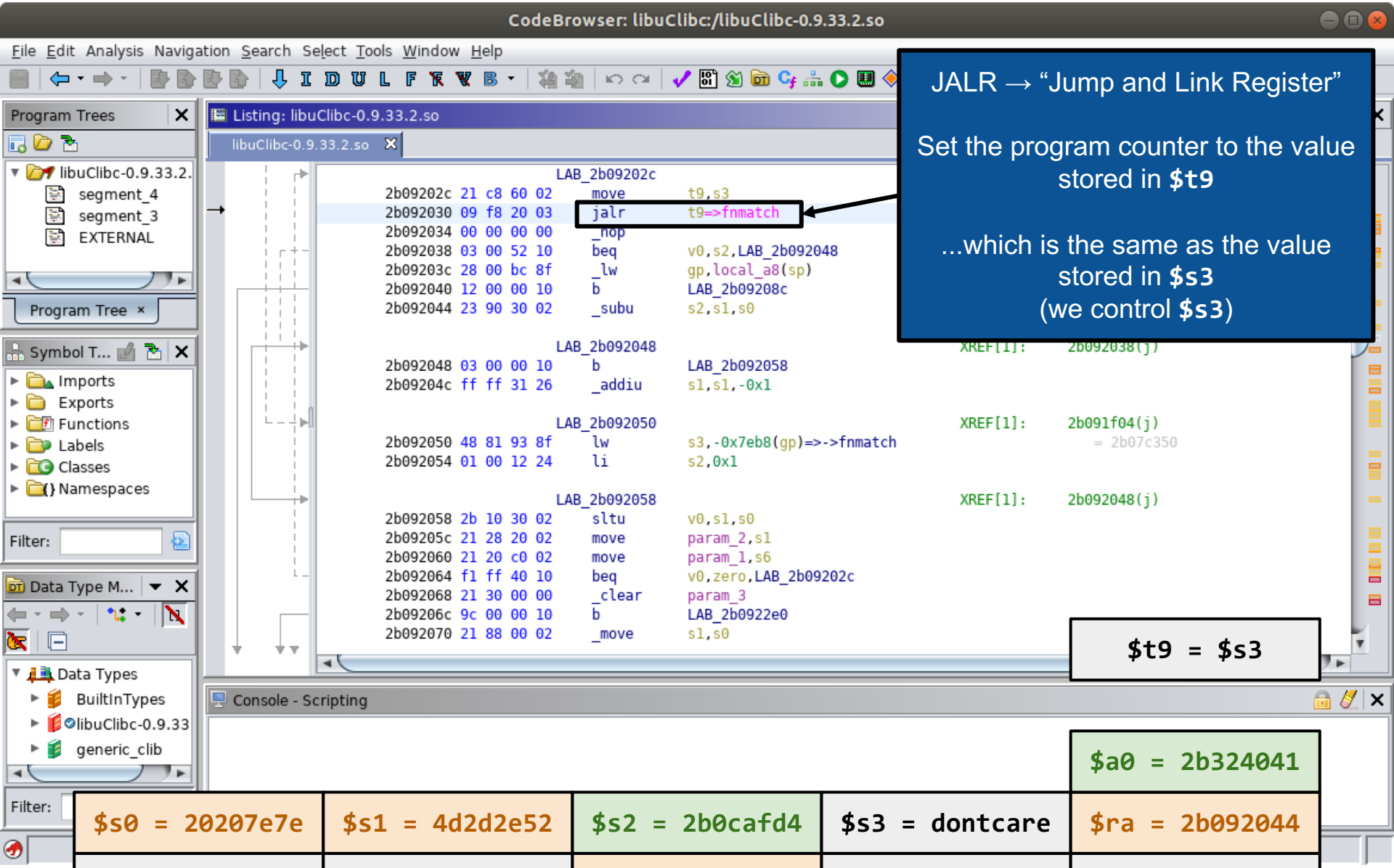
Orange → Set this value **Green** → Computed value



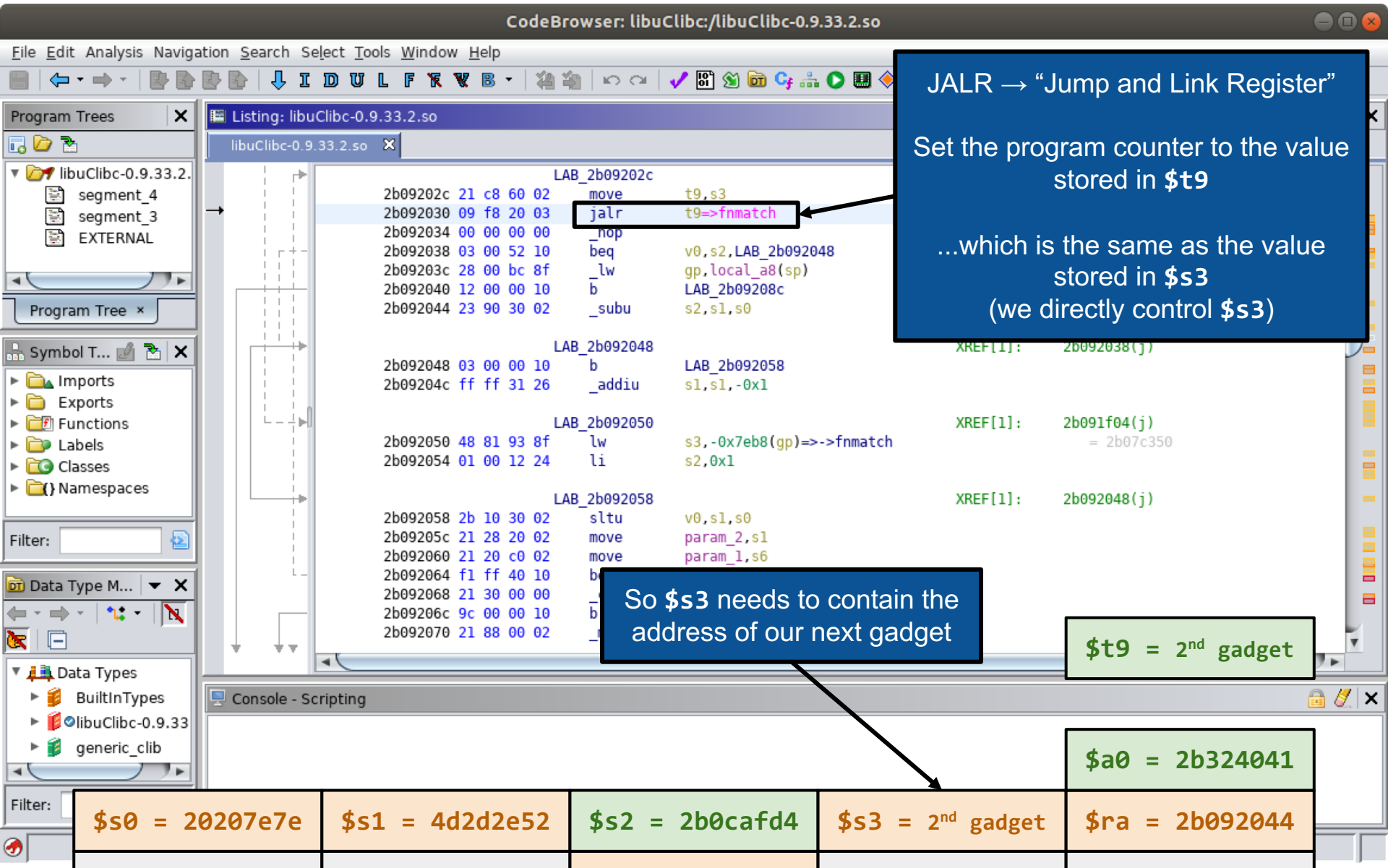
Orange → Set this value **Green** → Computed value



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



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



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 `0x2b0cafd4` 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 `0x2b0cafd4` 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

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` **MIPS32**

Purpose: Add Unsigned Word

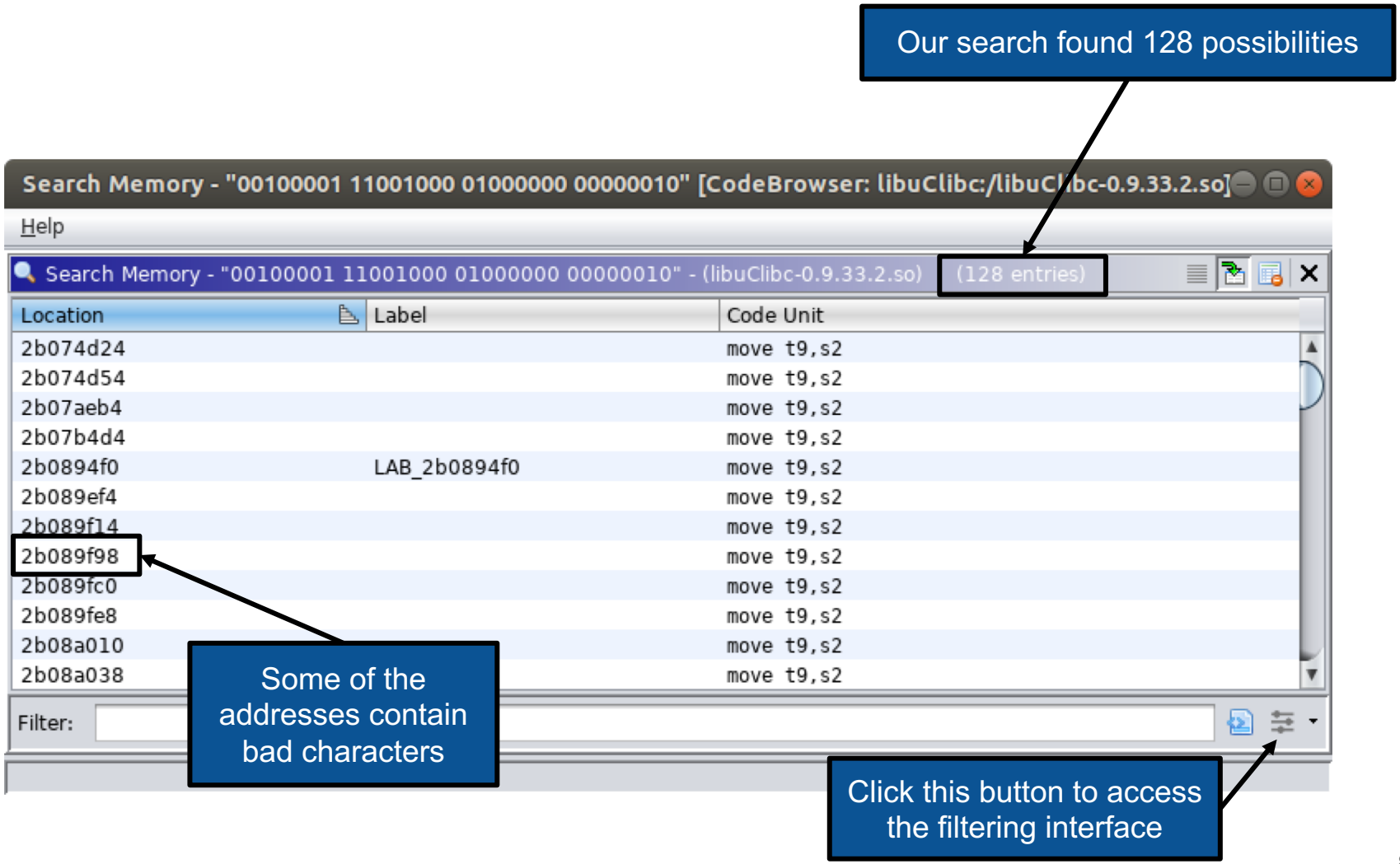
To add 32-bit integers.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

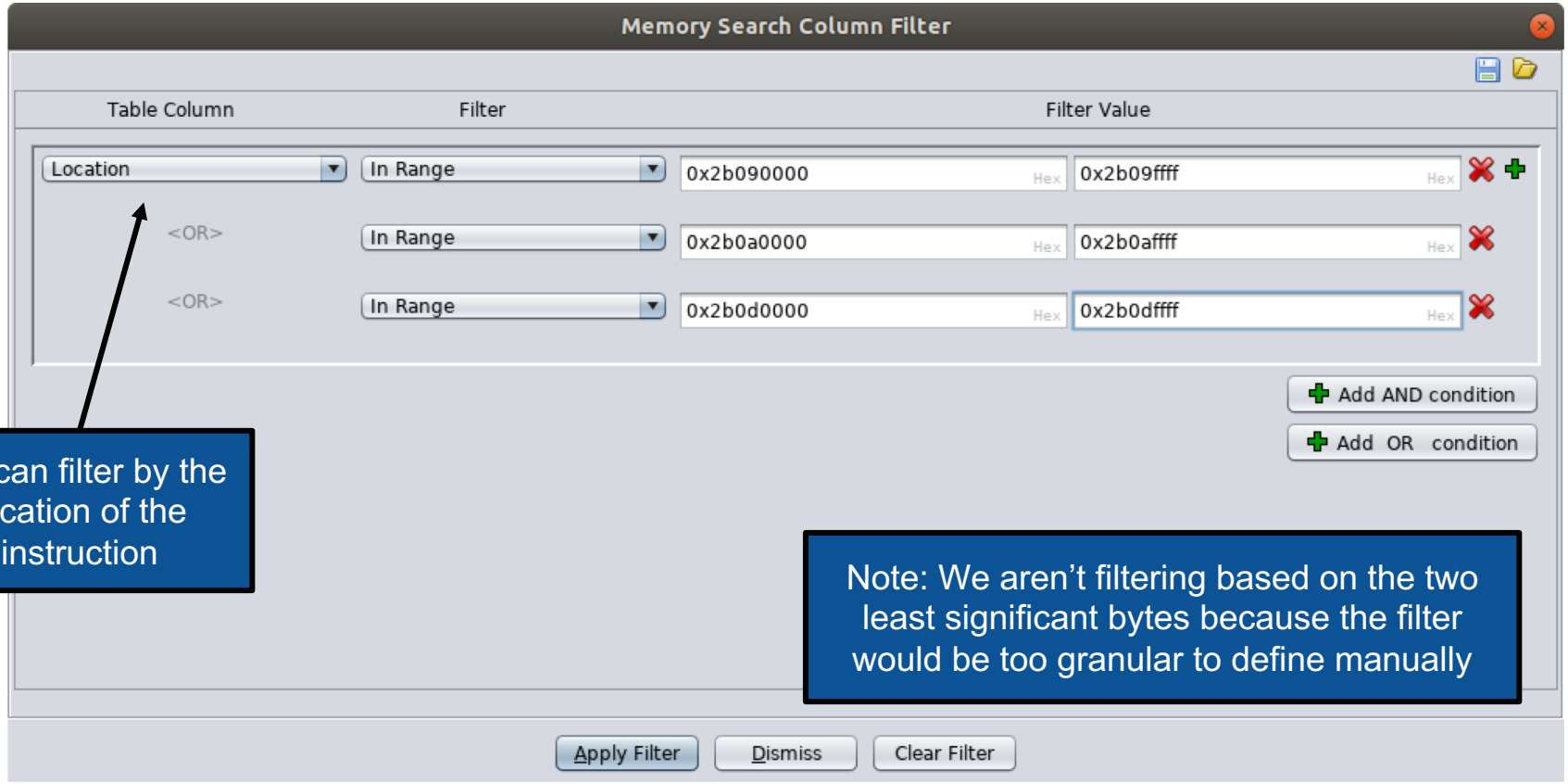
The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* and the 32-bit arithmetic result is placed into GPR *rd*.

The MIPS32® Instruction Set Manual [9]

Finding a '`move t9,s2`' instruction



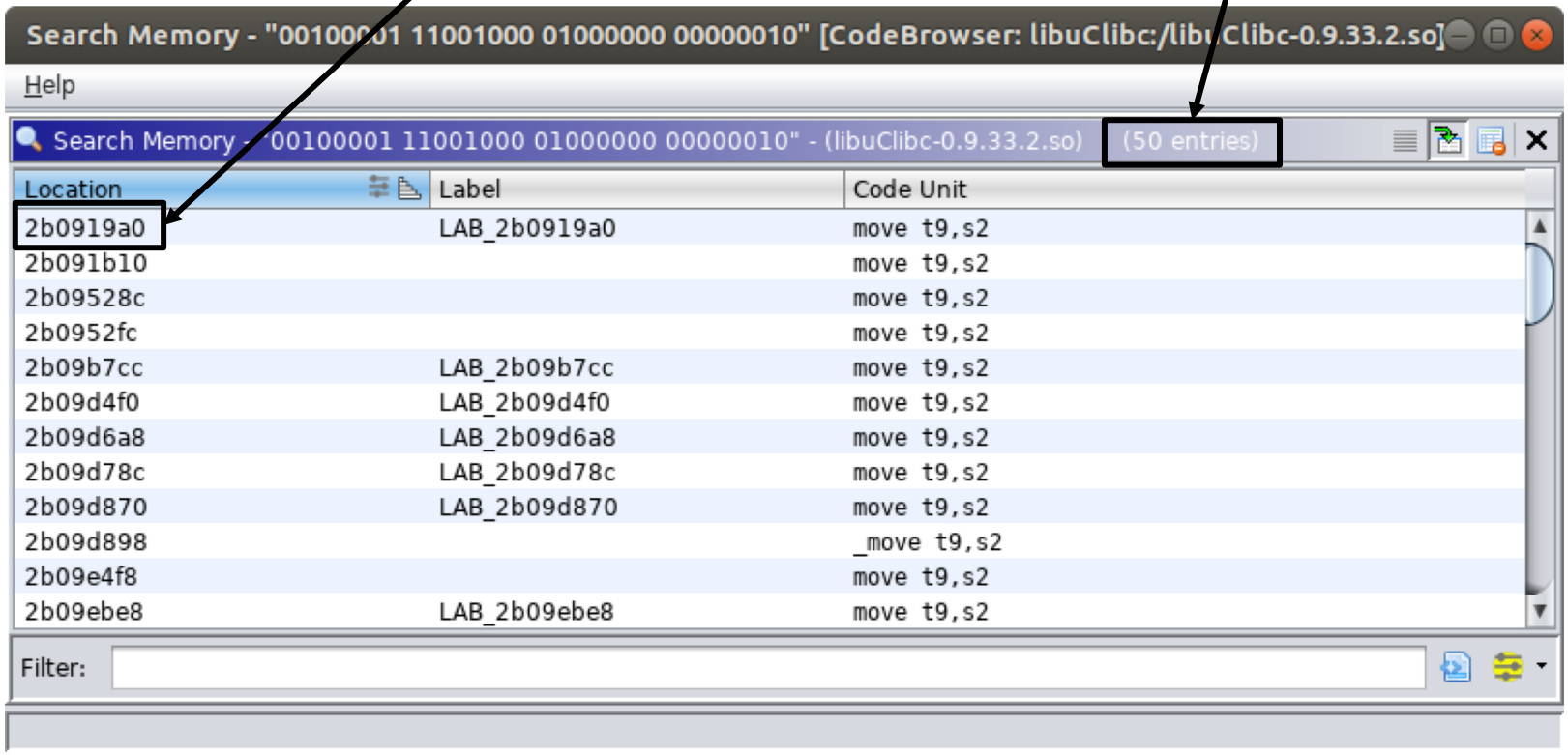
Finding a ‘move t9,s2’ instruction



Finding a ‘`move t9,s2`’ instruction

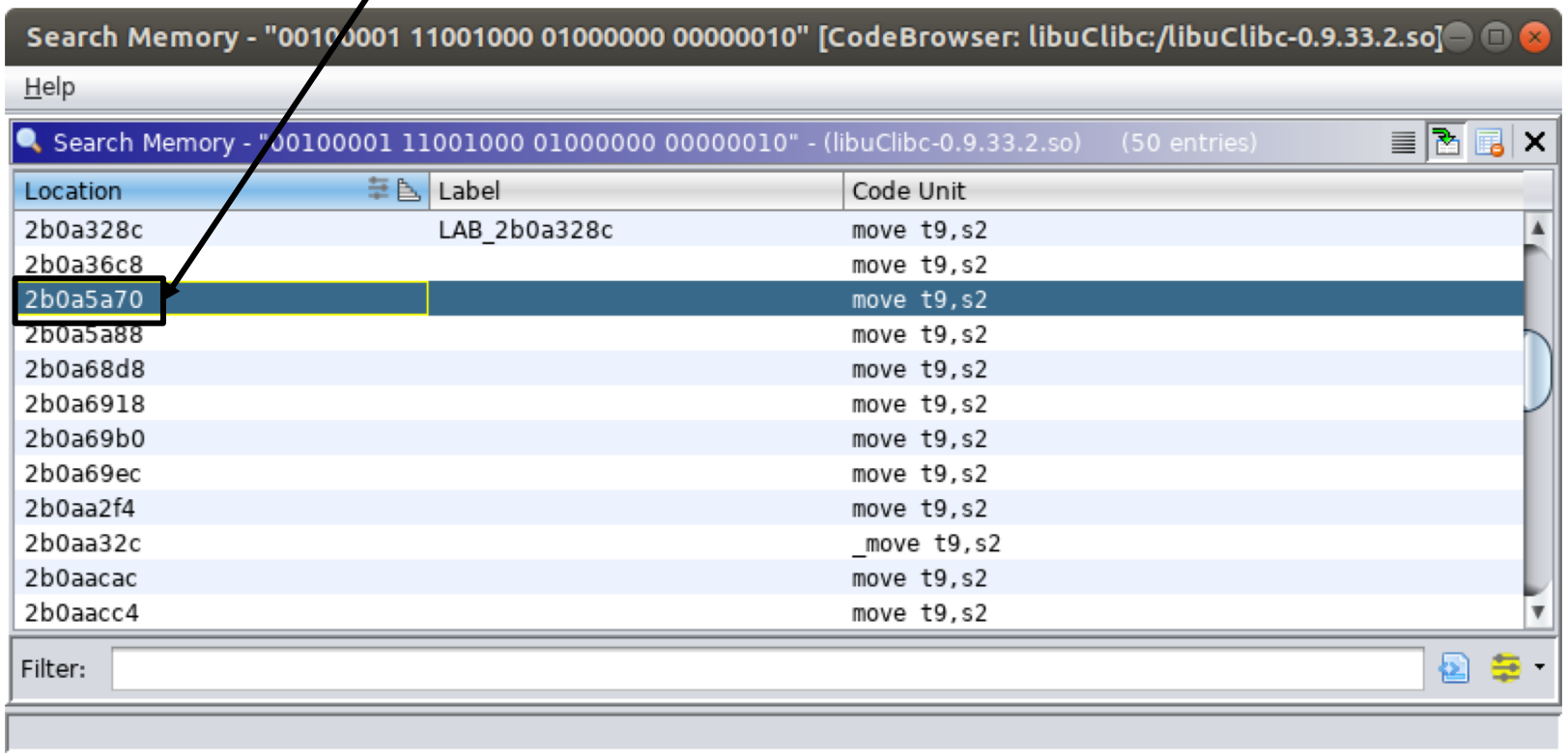
Some of the addresses still contain bad characters because we didn’t filter based on the two least significant bytes

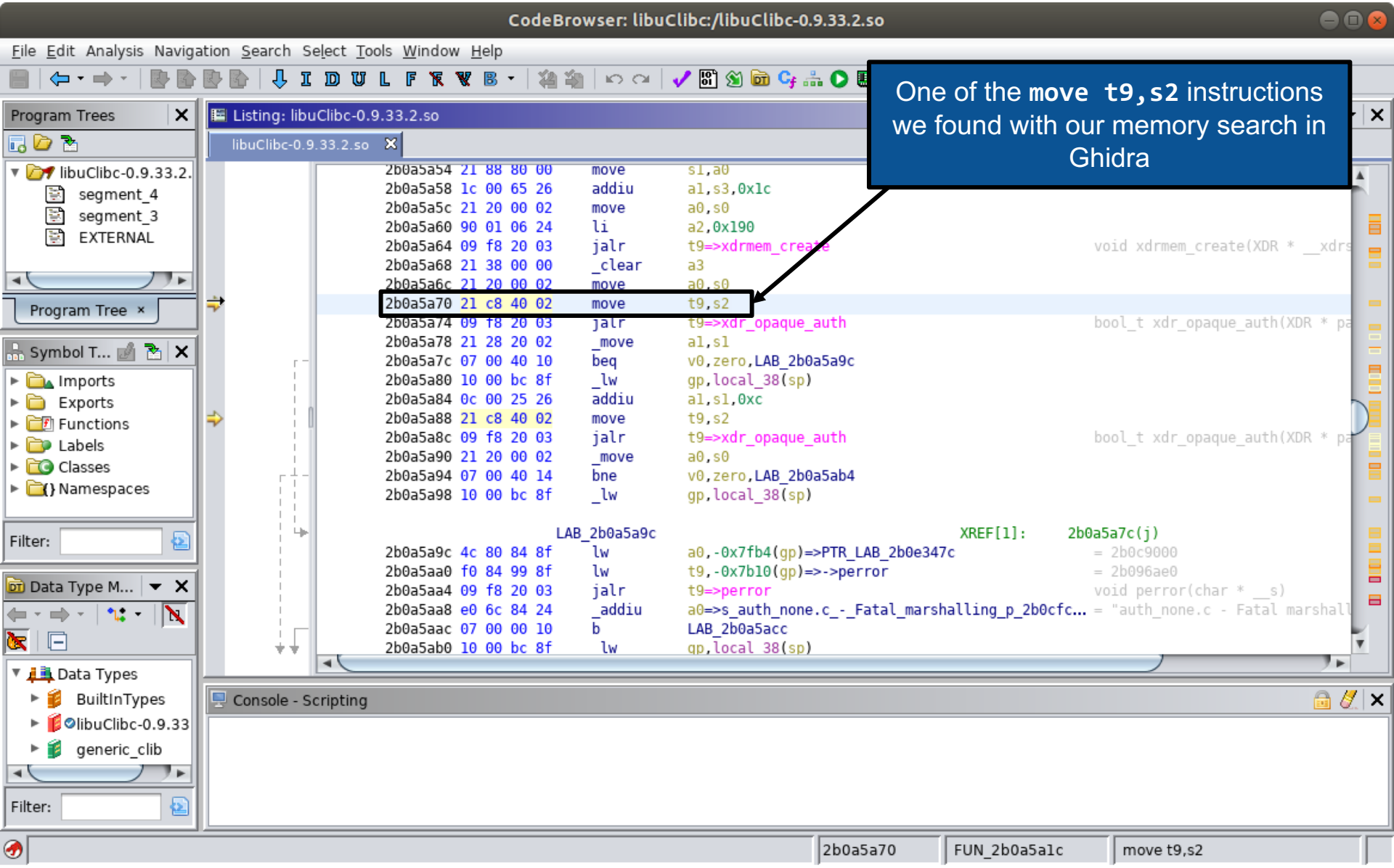
Our filter narrowed the candidates down to 50

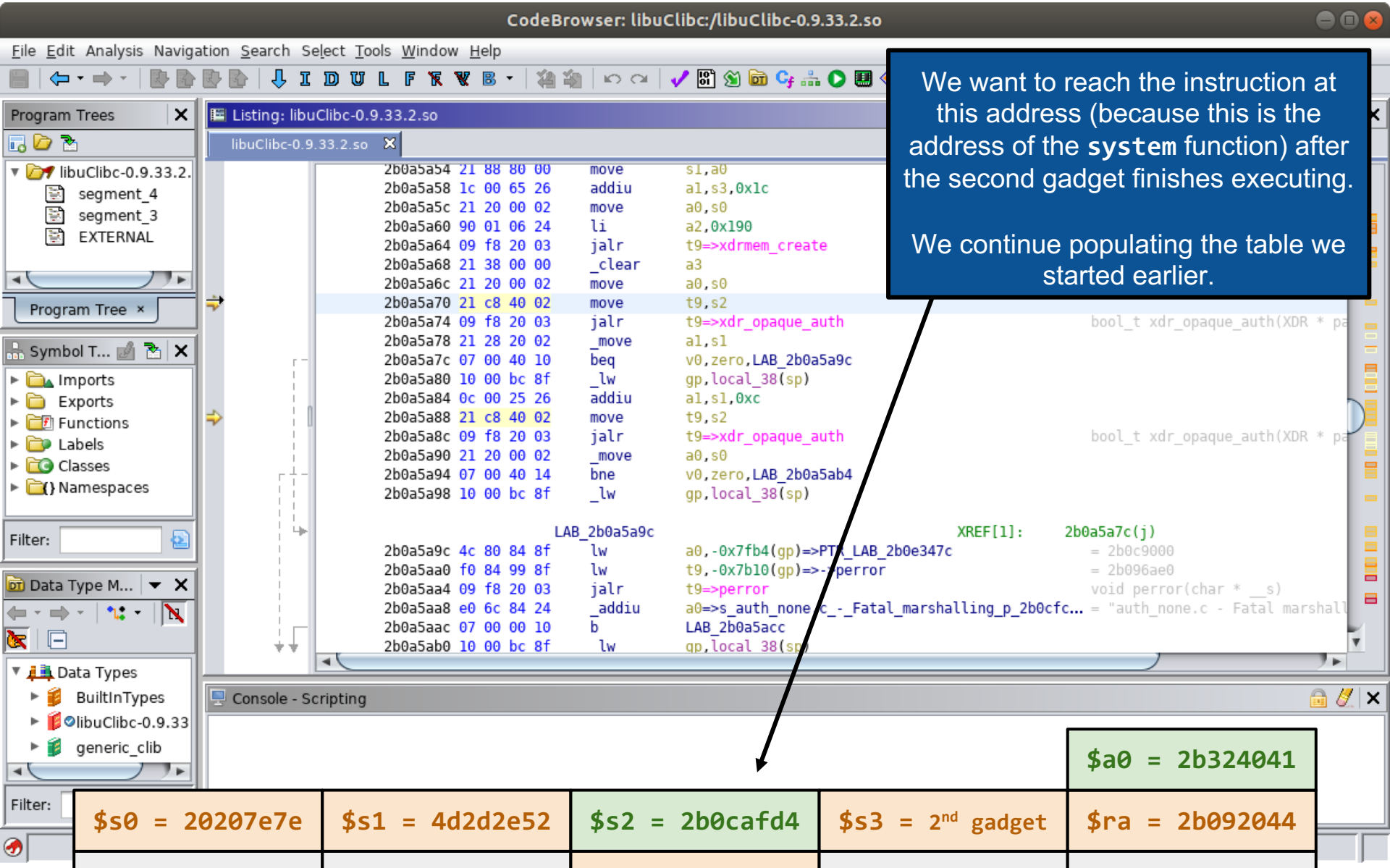


Finding a ‘move t9,s2’ instruction

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

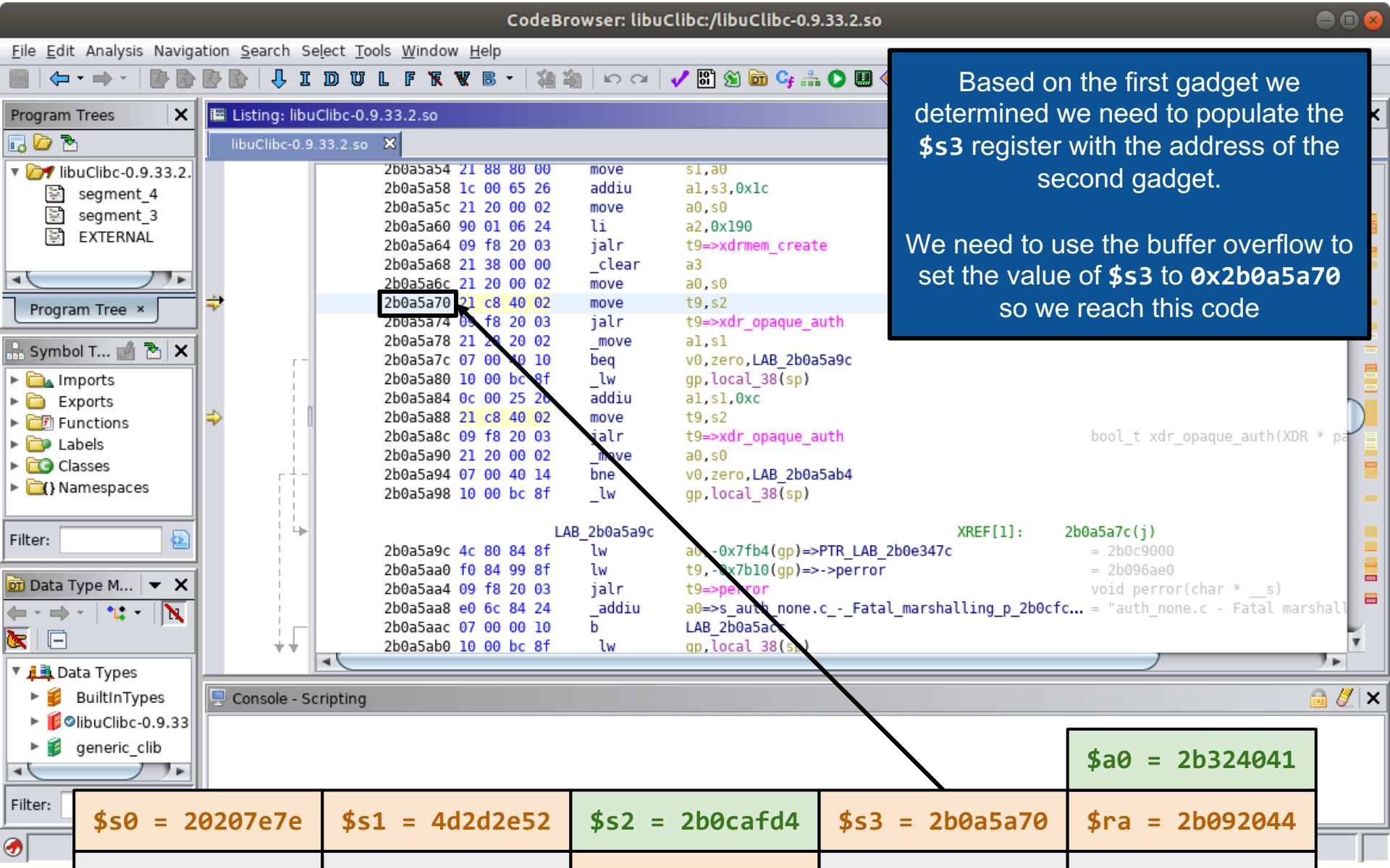




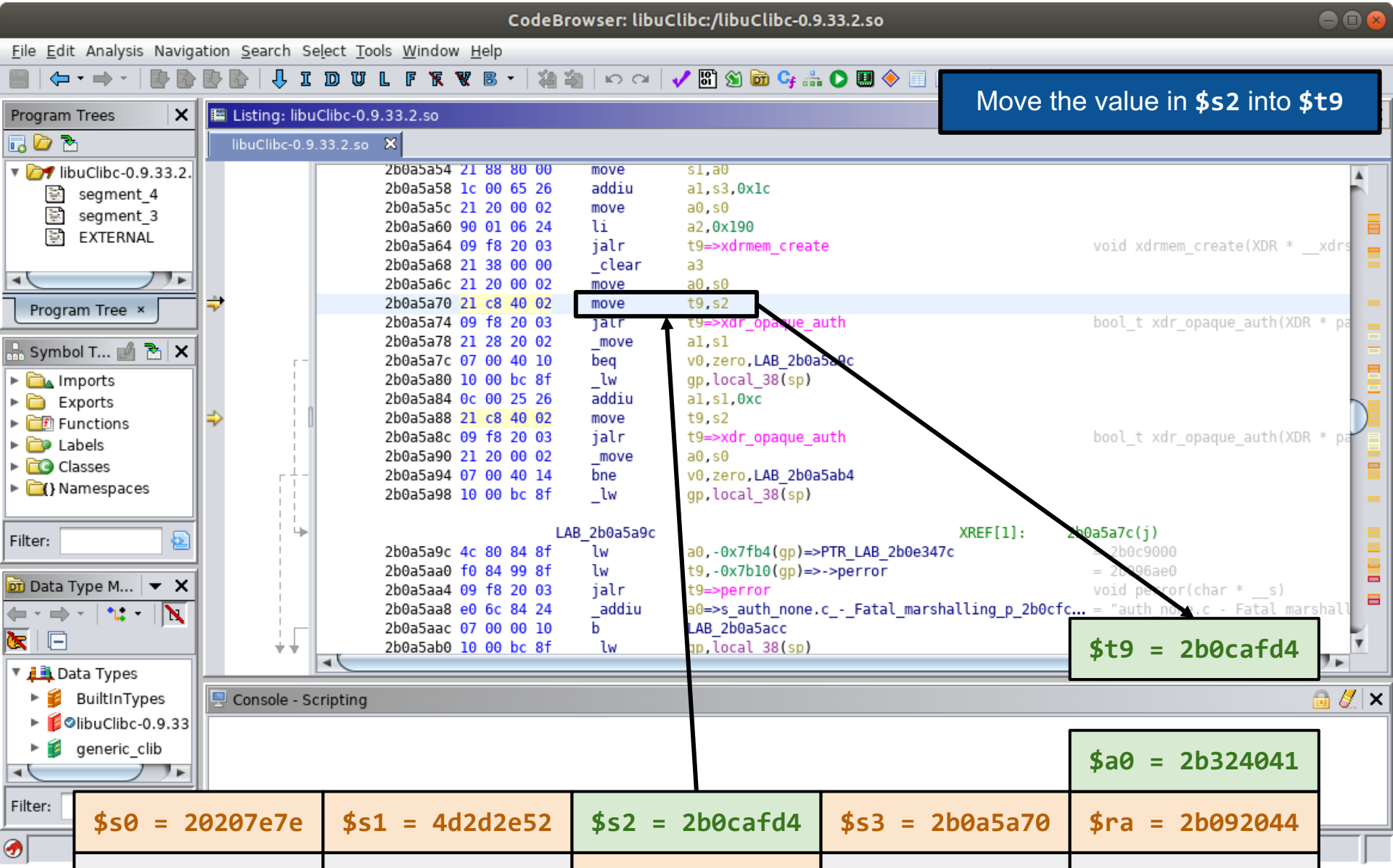


				$a0 = 2b324041
$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = 2nd gadget	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = 2b324041	$s7 = dontcare	$s8 = dontcare

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

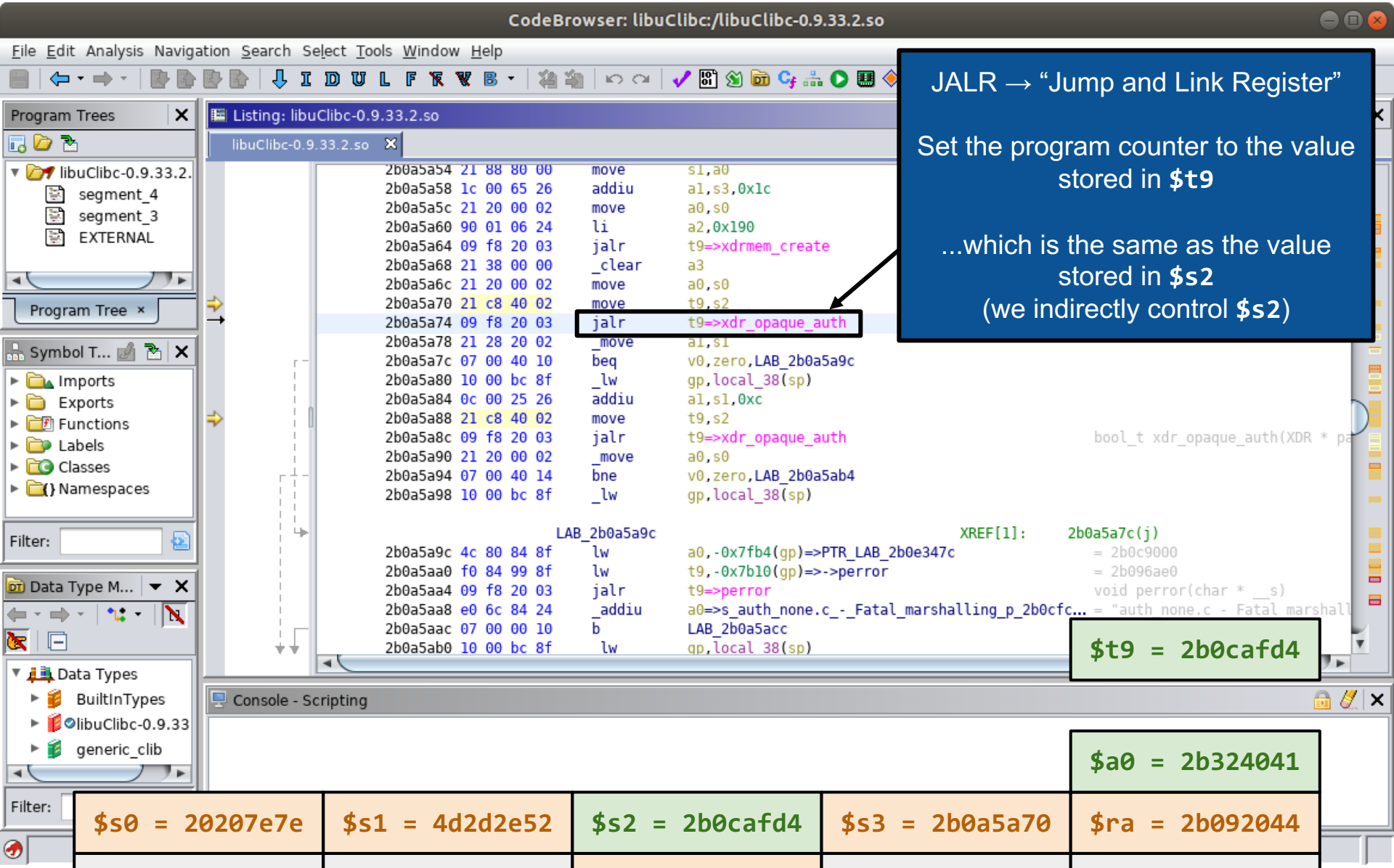


Orange → Set this value **Green** → Computed value

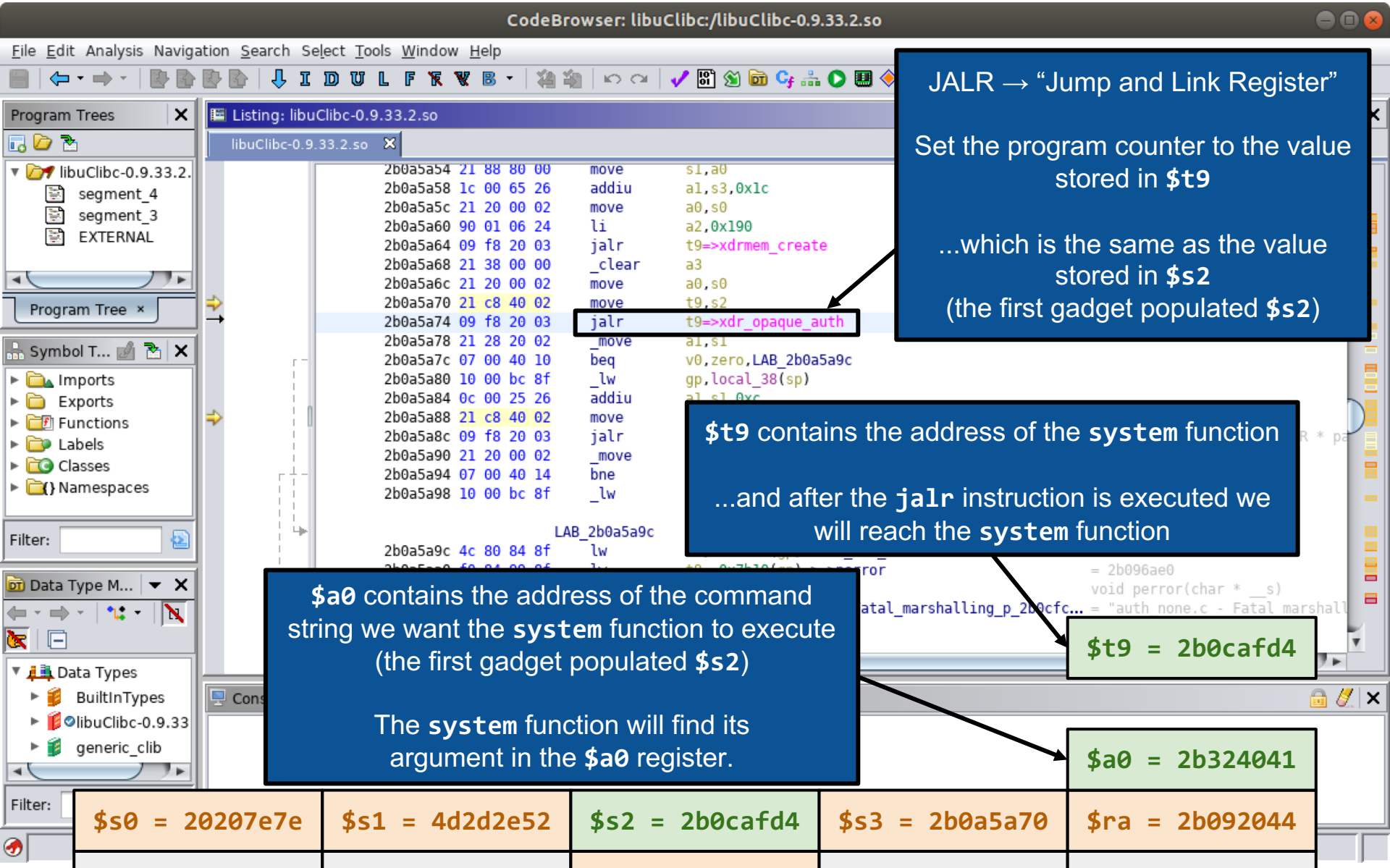


$s0 = 20207e7e	$s1 = 4d2d2e52	$s2 = 2b0cafd4	$s3 = 2b0a5a70	$ra = 2b092044
$s4 = dontcare	$s5 = dontcare	$s6 = 2b324041	$s7 = dontcare	$s8 = dontcare

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



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



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 `0x2b0cafd4` 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 `0x2b0cafd4` 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]. A[vailable: 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].