

**PACKT** [FOLLOW](#)

Stay Relevant!

Reverse Engineering a Linux executable – hello world

Published Jan 25, 2019

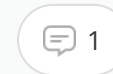
Learn how to reverse engineer a Linux executable – hello world in this article by Reginald Wong, a lead anti-malware researcher at Vipre Security, a J2 Global company, covering various security technologies focused on attacks and malware.

A lot of our tools work great in Linux. This article will discuss how to reverse an ELF file by exploring the reversing tools.

To begin with, let's create a hello world program. Before anything else, we need to make sure that the tools required to build it are installed. Open a Terminal and enter the following

password:

By using Codementor, you agree to our [Cookie Policy](#).

[ACCEPT](#)**Enjoy this post**Leave a like and comment for **PACKT**

```
sudo apt install gcc
```

The C program compiler, gcc, is usually pre-installed in Linux. Open any text editor and type the lines of following code, saving it as hello.c:

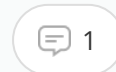
```
#include <stdio.h>
void main(void)
{
    printf ("hello world!\n");
}
```

You can use vim as your text editor by running vi from the Terminal. To compile and run the program, use the following commands:

The hello file is our Linux executable that displays a message in the console. Now, on to reversing this program.

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



As an example of good practice, the process of reversing a program first needs to start with proper identification. Let's start with file:

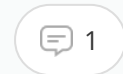
It is a 32-bit ELF file-type. ELF files are native executables on Linux platforms. Next stop, let's take a quick look at text strings with the strings command:

This command will produce something like the following output:

```
/lib/ld-linux.so.2
libc.so.6
_ IO_stdin_used
puts
__ libc_start_main
__ gmon_start__
GLIBC_2.0
PTRh
UWVS
t$,U
[^_ ]
hello world!
;* 2$(
GCC: (Ubuntu
```

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



```
__ do_global_dtors_aux
completed.7209
__ do_global_dtors_aux_fini_array_entry
frame_dummy
__ frame_dummy_init_array_entry
hello.c
__ FRAME_END__
__ JCR_END__
__ init_array_end
_ DYNAMIC
__ init_array_start
__ GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__ libc_csu_fini
_ ITM_deregisterTMCloneTable
__ x86.get_pc_thunk.bx
_ edata
__ data_start
puts@@GLIBC_2.0
__ gmon_start__
__ dso_handle
_ IO_stdin_used
__ libc_start_main@@GLIBC_2.0
__ libc_csu_init
_ fp_hw
__ bss_start
main
_ Jv_Register
```

By using Codementor, you agree to our
[Cookie Policy](#).

ACCEPT

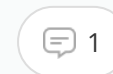
♥ 1

💬 1

```
.symtab  
.strtab  
.shstrtab  
.interp  
.note.ABI-tag  
.note.gnu.build-id  
.gnu.hash  
.dynsym  
.dynstr  
.gnu.version  
.gnu.version_r  
.rel.dyn  
.rel.plt  
.init  
.plt.got  
.text  
.fini  
.rodata  
.eh_frame_hdr  
.eh_frame  
.init_array  
.fini_array  
.jcr  
.dynamic  
.got.plt  
.data  
.bss  
.comment
```

By using Codementor, you agree to our
[Cookie Policy](#).

ACCEPT



The strings are listed in order from the start of the file. The first portion of the list contained our message and the compiler information. The first two lines also show what libraries are used by the program:

```
/lib/ld-linux.so.2  
libc.so.6
```

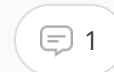
The last portion of the list contains names of sections of the file. We only know of a few bits of text that we placed in our C code. The rest are placed there by the compiler itself, as part of its code that prepares and ends the graceful execution of our code.

Disassembly in Linux is just a command line away. Using the `-d` parameter of the `objdump` command, we should be able to show the disassembly of the executable code. You might need to pipe the output to a file using this command line:

```
objdump -d hello > disassembly.asm
```

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



The output file, `disassembly.asm`, should contain the following code:

If you notice, the disassembly syntax is different from the format of the Intel assembly language that we learned. What we see here is the AT&T disassembly syntax. To get an Intel syntax, we need to use the `-M intel` parameter, as follows:

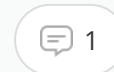
```
objdump -M intel -d hello > disassembly.asm
```

The output should give us this disassembly result:

The result shows the disassembly code of each function. In summary, there were a total of 15 functions from executable sections:

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



Disassembly of **section .init**:

080482a8 <_init>:

Disassembly of **section .plt**:

080482d0 <puts@plt-0x10>:

080482e0 <puts@plt>:

080482f0 <__libc_start_main@plt>:

Disassembly of **section .plt.got**:

08048300 <.plt.got>:

Disassembly of **section .text**:

08048310 <_start>:

08048340 <__x86.get_pc_thunk.bx>:

08048350 <deregister_tm_clones>:

08048380 <register_tm_clones>:

080483c0 <__do_global_ctors_aux>:

080483e0 <frame_dummy>:

0804840b <main>:

08048440 <__libc_csu_init>:

080484a0 <__libc_csu_fini>:

Disassembly of **section .fini**:

080484a4 <_fini>:

By using Codementor, you agree to our
[Cookie Policy](#).

ACCEPT

♥ 1

💬 1

The disassembly of our code is usually at the .text section. And, since this is a GCC-compiled program, we can skip all the initialization code and head straight to the main function where our code is at:

I have highlighted the API call on puts. The puts API is also a version of printf. GCC was smart enough to choose puts over printf for the reason that the string was not interpreted as a C-style formatting string. A formatting string, or formatter, contains control characters, which are denoted with the % sign, such as %d for integer and %s for string. Essentially, puts is used for non-formatted strings, while printf is used for formatted strings.

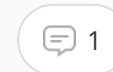
What have we gathered so far?

Assuming we don't have any idea of the source code, this is the information we have gathered so far:

- The file is a 32-bit ELF executable.
- It was compiled using GCC.
- It has 15 executable functions, including the main() function.
- The code us

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



message.

- The program is expected to display the message using puts.

Dynamic analysis

Now let's do some dynamic analysis. Remember that dynamic analysis should be done in a sandbox environment. There are a few tools that are usually pre-installed in Linux that can be used to display more detailed information. We're introducing ltrace, strace, and gdb for this reversing activity.

Here's how ltrace is used:

The output of ltrace shows a readable code of what the program did. ltrace logged library functions that the program called and received. It called puts to display a message. It also received an exit status of 13 when the program terminated.

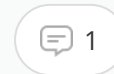
The address 0x804840b is also the address of the main function listed in the disassembly results.

strace is another

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT

result of



strace logged every system call that happened, starting from when it was being executed by the system. `execve` is the first system call that was logged. Calling `execve` runs a program pointed to by the filename in its function argument. `open` and `read` are system calls that are used here to read files. `mmap2`, `mprotect`, and `brk` are responsible for memory activities such as allocation, permissions, and segment boundary setting.

Deep inside the code of `puts`, it eventually executes a `write` system call. `write`, in general, writes data to the object it was pointed to. Usually, it is used to write to a file. In this case, `write`'s first parameter has a value of 1. The value of 1 denotes `STDOUT`, which is the handle for the console output. The second parameter is the message, thus, it writes the message to `STDOUT`.

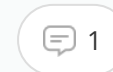
Going further with debugging

First, we need to install `gdb` by running the following command:

```
sudo apt install gdb
```

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



The installation should look something like this:

Then, use gdb to debug the hello program, as follows:

```
gdb ./hello
```

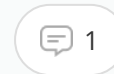
gdb can be controlled using commands. The commands are fully listed in online documentation, but simply entering help can aid us with the basics.

You can also use gdb to show the disassembly of specified functions, using the disass command. For example, let's see what happens if we use the disass main command:

Then, again we have been given the disassembly in AT&T syntax. To set gdb to use Intel syntax, use the following command:

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



```
set disassembly-flavor intel
```

This should give us the Intel assembly language syntax, as follows:

To place a breakpoint at the main function, the command would be `b * main`.

After placing a breakpoint, we can run the program using the run command. We should end up at the address of the main function:

To get the current values of the registers, enter `info registers`. Since we are in a 32-bit environment, the extended registers (that is, EAX, ECX, EDX, EBX, and EIP) are used. A 64-bit environment would show the registers with the R-prefix (that is, RAX, RCX, RDX, RBX, and RIP).

Now that we are at the main function, we can run each instruction with `step into` (the `stepi` command) and `step over` (the `nexti` command). Usually, we follow this with the `info registers`

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



Keep on entering si and disass main until you reach the line containing call 0x80482e0 puts@plt. You should end up with these disass and info registers result:

The => found at the left side indicates where the instruction pointer is located. The registers should look similar to this:

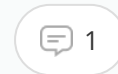
Before the puts function gets called, we can inspect what values were pushed into the stack. We can view that with x/8x \$esp:

Next, we need to do a step over (ni) the call instruction line. This should display the following message:

But if you used si, the instruction pointer will be in the puts wrapper code. We can still go back to where we left off using the until command, abbreviated as u. Simply using the until command steps in one instruction. You'll have to indicate the address location where it will stop. It is like a temporary breakpoint. Remember to place an asterisk before

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



The remaining 6 lines of code restore the values of ebp and esp right after entering the main function, then returning with ret. Remember that a call instruction would store the return address at the top of the stack, before actually jumping to the function address. The ret instruction will read the return value pointed to by the esp register.

The values of esp and ebp, right after entering the main function, should be restored before the ret instruction. Generally, a function begins by setting up its own stack frame for use with the function's local variables.

Reverse engineering

Linux

Debugging

Report

Here's a table showing the changes in the values of the esp, ebp, and ecx registers after the instruction at the given address:

Enjoy this post? Give **PACKT** a like if it's helpful.

♥ 1

💬 1

🔗 SHARE

You can either continue exploring the cleanup code after ret, or just make the program eventually end by using continue or its abbreviation, c, as follows:



PACKT

Stay Relevant!

If you found this article interesting, you can explore [Mastering Reverse Engineering](#)

FOLLOW

🗨️ 1 Reply

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT

♥ 1

💬 1

Leave a reply



shai ben shalom 8 months ago



great article!

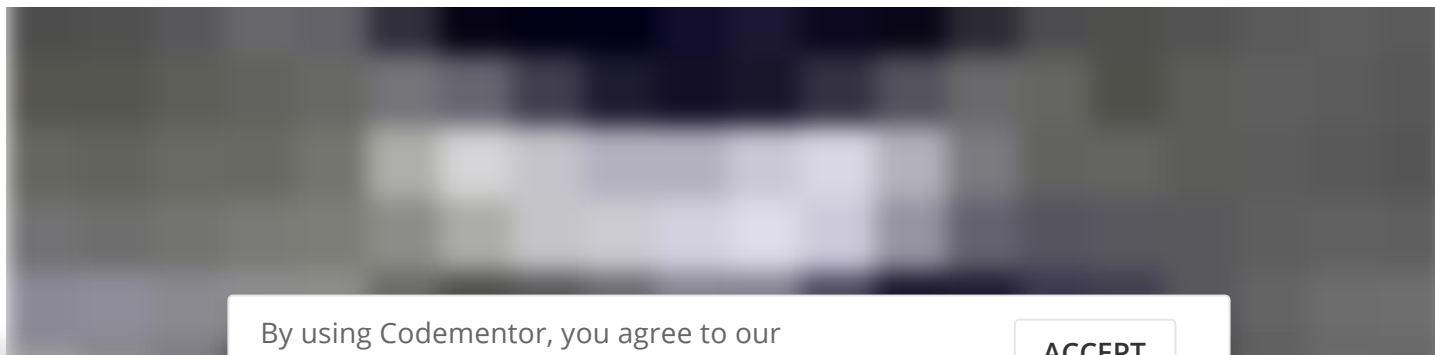


Reply



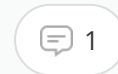
LikeGeeks


MySQL on Linux (Beginners Tutorial)



By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT





In this post, we will cover many aspects of MySQL on Linux. First of all, how to install it, how to perform basic CRUD operations, how to import & export data, play with the MySQL engine itself such as setting the root user password and much more.

MySQL is one of the most popular relational databases engines in the world. It has earned its fame for being open source and quite stable.

It is also compatible with most known programming languages. Of course, it's possible to install it and use it on many different distributions that exist, for example, ...

[READ MORE](#)

By using Codementor, you agree to our [Cookie Policy](#).

ACCEPT



1



1