A friendly user guide to ELF microcode patching with Shiva (alpha v1)

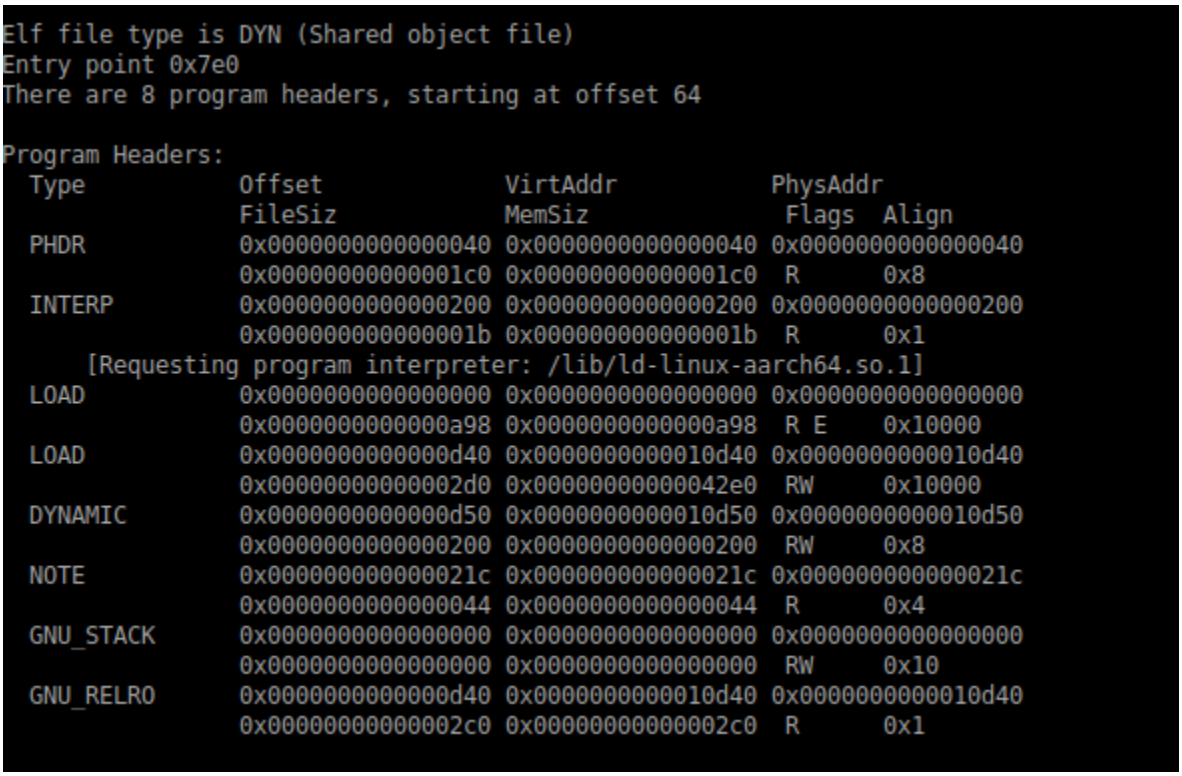
Shiva Manual

1 Introduction to Shiva

ELF program interpreters

Shiva is an ELF microcode-patching technology that works similarly to the well known (*but little understood*) ELF dynamic linker "/lib/ld-linux.so". Every ELF program that is dynamically linked has a program header segment type called PT_INTERP that describes the path to the *program interpreter.* The program interpreter is a separate ELF program loaded by the kernel that is responsible for helping to build the process runtime image by loading and linking extra modules containing code and data. Traditionally these modules are called shared libraries, and the program interpreter is "/lib/ld-linux.so". On aarch64 it is "/lib/ld-linux-aarch64.so.1"

Image 1.0: Requesting program interpreter



Just like any other interpreter, such as Python, or Java, the ELF program interpreter is responsible for performing various types of loading, patching and transformation.

Shiva is an ELF program interpreter

Shiva is a newly designed program interpreter who's sole purpose is to load ELF microcode patches that are written in C and compiled into Shiva modules (ELF relocatable objects).

Shiva modules are relocatable objects and by nature already contain the ELF meta-data *(sections, symbols relocations)* necessary to re-build much of the process image. Additionally Shiva has introduced some new concepts such as *Transformations* which are an extension to traditional ELF relocations, allowing for complex patching operations such as function splicing.

The core purpose of Shiva is to create a microcode patching system that seamlessly fits into the existing ELF ABI toolchain of compilers, linkers, and loaders. Shiva aims to load and link patches that are written 100% in C, in a way that is natural for developers who may not be reverse engineers but are experts in C. Shiva is in-itself a linker and a loader that works alongside and in conjunction with the existing Dynamic linker "ld-linux.so". To understand more technical details see "shiva_final_desing.pdf" within the documentation directory.

Systems and arch's that Shiva supports

Currently Shiva's AMP tailored-microcode patching system runs on **Linux AArch64**, and supports **AArch64 Linux ELF PIE binaries**. Future support for ARM32, x86_64/x86_32 and other requested architectures. At the present moment Shiva doesn't support ET_EXEC ELF binaries (non-pie) due to time constraints, but it's on the road map.

2 Installing Shiva-AArch64

Download Source code

git clone git@github.com:advanced-microcode-patching/shiva
git clone github.com/elfmaster/libelfmaster

Build source code

Build and install libelfmaster

Shiva requires a custom libelfmaster branch that has not been merged into main yet. Specifically make sure to use the *aarch64_support* branch.

cd libelfmaster
git checkout aarch64_support
cd ./src

sudo ./make.sh

Build and install Shiva

Commands:

cd shiva
make
make shiva-ld
make patches
sudo make install

Build results:

Shiva is a static ELF executable and installed to "/lib/shiva". The Shiva modules are typically stored in "/opt/shiva/modules". The example patches all live within "/git/shiva/modules/aarch64_patches/".

Build and install example patches

Please see the "shiva/modules/aarch64_patches/" directory. All of the example patches live here. Let us illustrate an example of building and testing one such patch on a program called 'test_rodata'.

Commands:

cd shiva/modules/aarch64_patches/rodata_interposing
make
sudo make install

Test the patch by setting the patch path and running "/lib/shiva" manually.

$ SHIVA_MODULE_PATH=./ro_patch.o /lib/shiva ./test_rodata

Notice that the software prints the new string "The great arcanum", per the patch. Yet when the program is executed without Shiva it executes with its old behavior.

$ ./test_rodata

Install the patch permanently

You may also install the patch permanently with the Shiva prelinking tool. This tool updates the PT_INTERP segment from "/lib/ld-linux.so" to "/lib/shiva", and updates the dynamic segment with several new tags describing the path to the patch that should be loaded and linked at runtime: "/opt/shiva/modules/ro_patch.o".

$ /bin/shiva-ld -e test_rodata -p ro_patch.o -i /lib/shiva -s /opt/shiva/modules -o test_rodata.patched

Shiva now installs the patch in memory everytime it's executed

$ ./test_rodata.patched

The only discernible difference between test_rodata.patched and test_rodata are the PT_INTERP segment modification and the PT_DYNAMIC addition of two new entries describing the patch path. No actual code or data has changed within the ELF binary. Shiva performs all of the patching at runtime.

3 Writing patches with Shiva

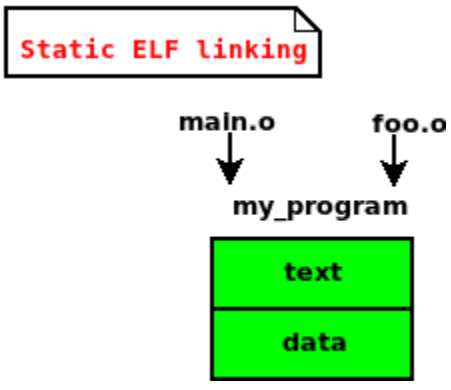
The Shiva workflow

Before delving in to the use of Shiva, let us first explain the workflow of what the ELF compilation and linking process looks like before and after Shiva.

ELF Program compilation and linking example

Standard program compilation begins with a compiler like gcc; the source code files (I.e. main.o and foo.o) get linked into a final output ELF executable my_program.

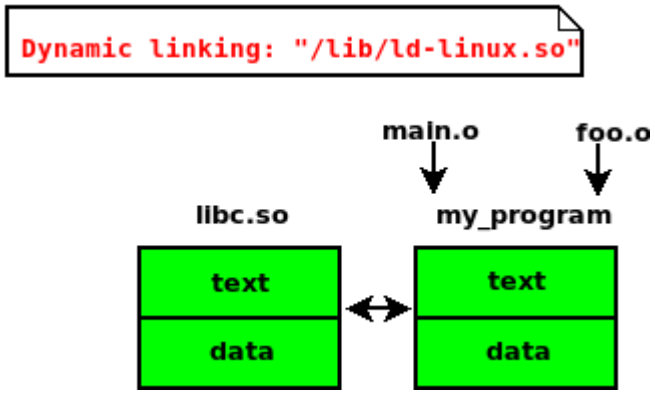
Image 3.0: static linking of two ELF relocatable objects



Additionally these ELF relocatable object files contain calls to functions that live within ELF shared libraries. Once the object files (foo.o and main.o) are linked into an ELF executable (my_program), additional Dynamic linking meta-data is created and stored into what is known as the PT_DYNAMIC segment of the program header table. The PT_DYNAMIC

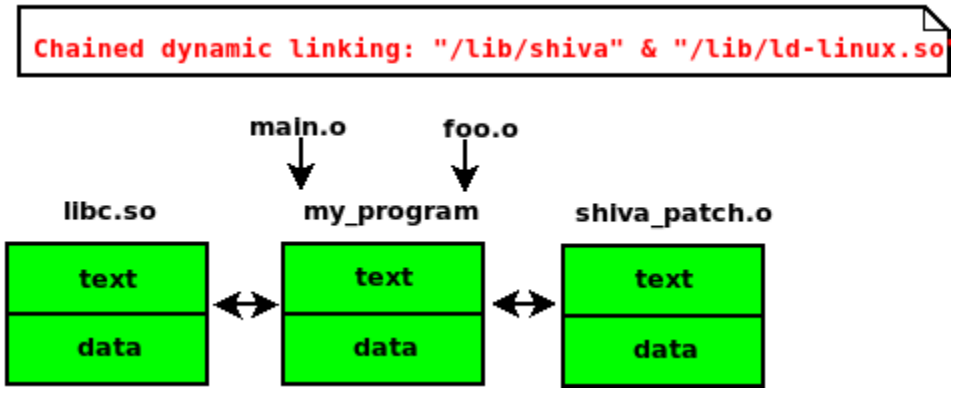
segment contains additional meta-data that is to be passed along to the Program interpreter at runtime. The program interpreter (aka: dynamic linker) is specified as a file path in the PT_INTERP segment: “/lib/ld-linux.so”. When the executable is run, the kernel loads and executes the interpreter program first: “/lib/ld-linux.so”. The interpreter loads and links the needed shared libraries (i.e. libc.so). The program has now been compiled, and linked twice: once statically, and again dynamically.

Image 3.1: Dynamically link libc.so in at runtime



At this point if the program “my_program” needs to be patched, a patch can be written in C, and compiled as an ELF relocatable object file. In the same sense that “ld-linux.so” loads and links shared libraries (ELF ET_DYN files), Shiva will load and link patch objects (ELF ET_REL files) to re-write the program in memory just before execution.

Image 3.2: Chained linking with Shiva



ELF Symbol interposition

The ELF format uses symbols to denote code and data. Program functions generally live in the .text section, and global variables live within the .data, .rodata, and .bss sections. Shiva makes use of the ELF relocation, section, and symbol data to allow patch developers to *interpose* existing symbols; in other words re-link functions and data that have symbols associated with them. As patch developers we will find this as a natural and beautifully convenient way to patch code and data.

Citation 2.0: From "Oracle linkers and libraries guide"

"Interposition can occur when multiple instances of a symbol, having the same name, exist in different dynamic objects that have been loaded into a process. Under the default search model, symbol references are bound to the first definition that is found in the series of dependencies that have been loaded. This first symbol is said to interpose on the other symbols of the same name."

Patching an .rodata string (Simple first patch)

Patching exercise lives in "modules/aarch64_patches/rodata_interposing".
In the following example we are going to look at a simple program that prints a constant string "Arcana Technologies". Constant data is read-only and is usually stored in the .rodata section of a program. The .rodata section is generally put into text segment, or a read-only segment that is adjacent to the text segment. A shiva module has it's own .rodata section within it's mapped text segment at runtime. Our goal with the following patch is to replace a read-only string with another by simply re-defining it in C.

Image 2.0: Source code of program rodata_test.c

```c
#include <stdio.h>

const char rodata_string[] = "Arcana Technologies";

int main(void)
{
        printf("rodata_string: %s\n", rodata_string);
        exit(0);
}
```

This program simply prints *rodata_string[]* which is a string stored within the .rodata section. Our goal is to change the string from "Arcana Technologies" to "The Great Arcanum" without modifying the programs source code. As a C developer it would be natural to define a new read-only string of the same name that holds our new string value. This is a great example of where interposing one symbol for another can be very helpful in writing natural C code patches.

Image 2.1: Source code of ro_patch.c (The patch code)

```
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ cat ro_patch.c

const char rodata_string[] = "The Great Arcanum";
```

This simple patch is re-defining a variable named *rodata_string[]*. At runtime Shiva will build a module process image which will contain the patch .rodata section with the updated version of *rodata_string[]*. Shiva will re-link the executable to use the new version of *rodata_string[]* that lives within the loaded modules .rodata section This is the concept of symbol interposition; one global symbol taking precedence over another with the same symbol name and type. Furthermore Shiva allows the user to interpose symbols of different types as well (Including STB_LOCAL symbols) as seen in later examples.

Compile this patch as an AArch64 Linux ELF relocatable object using a large code model. Each patch example, such as "modules/aarch64_patches/rodata_interposing", contains a Makefile that illustrates how to build a patch object. Generally speaking, a Shiva patch should contain the include path to "modules/include" which contain the "shiva_module.h" header file, necessary to build many patches.

Image 2.2: "rodata_interposing" Makefile with comments

```
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ cat Makefile
INTERP_PATH="/lib/shiva"
SHIVA-LD_PATH="../../../tools/shiva-ld/shiva-ld"
all:
        # Build the module ro_patch.c with a large code model
        gcc -mcmodel=large -fno-pic -I ../ -fno-stack-protector -c ro_patch.c

        # Build the program we are patching
        gcc -O0 test_rodata.c -o test_rodata

        # Pre-link the program we are patching with the proper interpreter and patch meta-data
        $(SHIVA-LD_PATH) -e test_rodata -p ro_patch.o -i /lib/shiva -s /opt/shiva/modules -o test_rodata.patched

clean:
        rm -f test_rodata test_rodata.patched ro_patch.o

elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$
```

The Makefile builds the patch, the executable, and pre-links the executable with the patch meta-data, creating a final output executable called "test_rodata.patched". Though we may test our patch by specifying the module path and invoking "/lib/shiva" directly to load our executable and patch.

Image 2.3: Invoking Shiva directly on test_rodata program

```
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ SHIVA_MODULE_PATH=./ro_patch.o /lib/shiva ./test_rodata
rodata_string: The Great Arcanum
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ ./test_rodata
rodata_string: Arcana Technologies
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$
```

Notice in the example here above that when running ./test_rodata the first time, with "/lib/shiva" it installs the specified patch, and the new *rodata_string[]* value "The great Arcanum" is shown. When we run ./test_rodata the second time without "/lib/shiva" it does not have the patch installed and the original string "Arcana technologies" is printed.

Alternatively and more commonly we can execute the prelinked executable directly which will indirectly invoke Shiva as the interpreter and locate the patch object in "/opt/shiva/modules/ro_patch.o", therefore installing the patch at runtime everytime the program is executed.

Image 2.4: Executing test_rodata.patched with Shiva as the interpreter

```
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ readelf -l test_rodata | grep Requesting
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ readelf -l test_rodata.patched | grep Requesting
      [Requesting program interpreter: /lib/shiva]
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ ./test_rodata
rodata_string: Arcana Technologies
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$ ./test_rodata.patched
rodata_string: The Great Arcanum
elfmaster@esoteric-aarch64:~/amp/shiva/modules/aarch64_patches/rodata_interposing$
```

In the above example notice that we illustrate which program interpreter is being used by ./test_rodata vs. ./test_rodata.patched. When executing ./test_rodata.patched the Shiva interpreter is used.