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

Shiva Manual

1 Introduction to Shiva

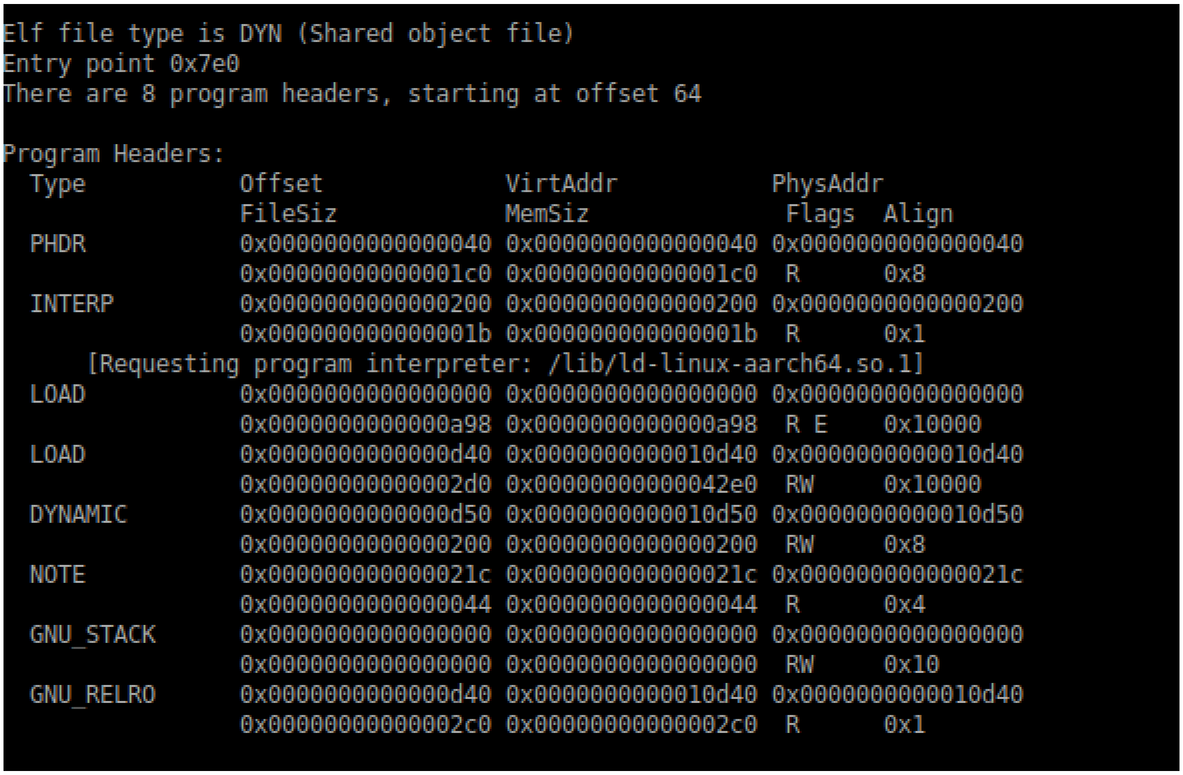
Shiva is LEJIT!

Shiva is a LEJIT micropatching system for Linux ELF software. **LEJIT (Linking & Encoding Just In Time).**

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 “ld-linux.so”



```
Elf file type is DYN (Shared object file)
Entry point 0x7e0
There are 8 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000000200 0x0000000000000200
                 0x000000000000001b 0x000000000000001b  R      0x1
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000a98 0x0000000000000a98  R E    0x10000
  LOAD           0x0000000000000d40 0x0000000000010d40 0x0000000000010d40
                 0x00000000000002d0 0x00000000000042e0  RW     0x10000
  DYNAMIC        0x0000000000000d50 0x0000000000010d50 0x0000000000010d50
                 0x0000000000000200 0x0000000000000200  RW     0x8
  NOTE           0x000000000000021c 0x000000000000021c 0x000000000000021c
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000d40 0x0000000000010d40 0x0000000000010d40
                 0x00000000000002c0 0x00000000000002c0  R      0x1
```

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

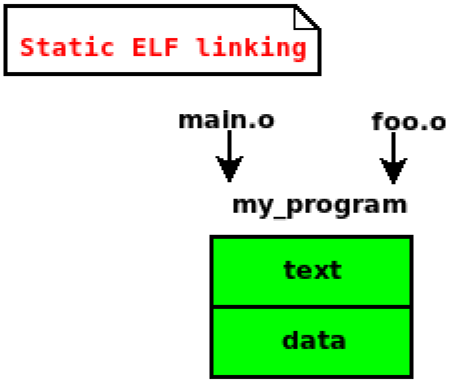
Before delving in to the use of Shiva, let us first explain the workflow of what the ELF compilation and linking process looks like before and after Shiva.

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) are compiled and then get linked (By “/bin/ld”) into a final output ELF executable my_program.

Static linking of objects

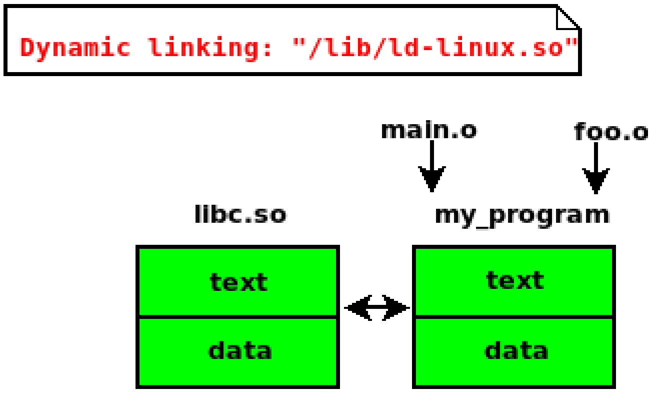
Image 3.0: static linking of two ELF relocatable objects with “/bin/ld”



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.

Dynamically linked executable

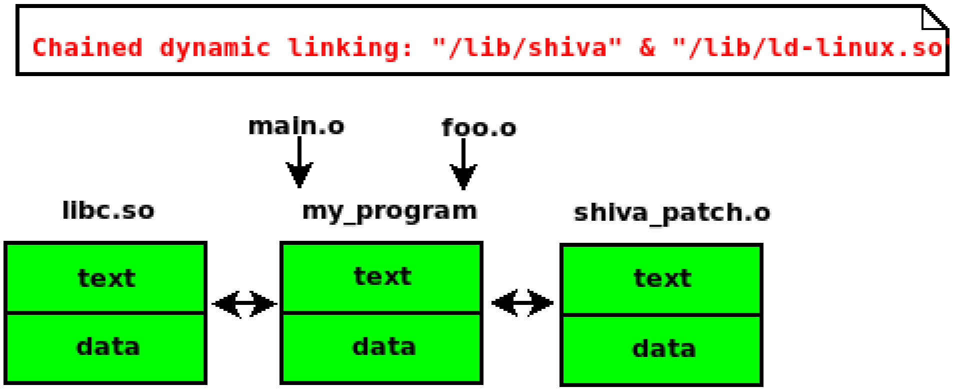
Image 3.1: Dynamically link libc.so in at runtime



A Shiva 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.

Dynamically patched executable

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.

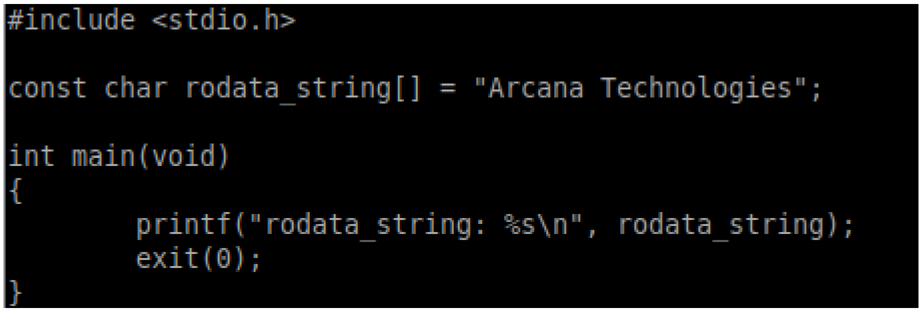
Citation 3.0: From "Oracle linkers and libraries guide"

"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)

This 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.

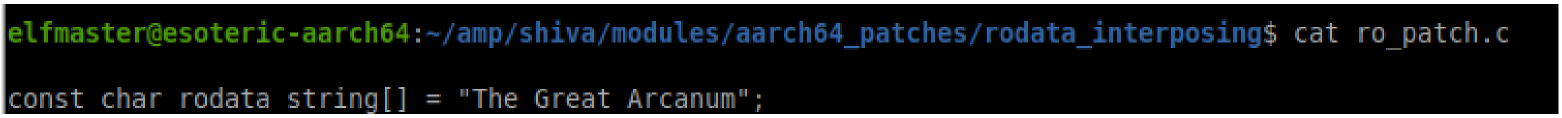
Image 3.3: Source code of program rodata_test.c



```
#include <stdio.h>

const char rodata_string[] = "Arcana Technologies";

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

NOTE: The source code to rodata_test.c is only being shown for the sake of illustrating what the program does that we are patching. In may workflows the source code wouldn't even exist, and Shiva doesn't depend on or use the source in any way.

Image 3.4: Source code of patch object: ro_patch.c



```
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 mo

Image 2.2: "rodata_interposing" Makefile with comments

Building the patch

The patch source code "modules/aarch64_patches/rodata_interposing/ro_patch.c" is a single line patch, redefining the symbol "rodata_string[]". To build this patch we can simply run the following gcc command to build the patch:

```
$ gcc -mcmodel=large -fno-pic -fno-stack-protector -c ro_patch.c -o ro_patch.o
```

The ELF ET_REL object: ro_patch.o describes a symbol with the same exact name as the symbol **rodata_string** in the target executable. Shiva will see this when linking and it will link in the version of the symbol from our patch. This perfectly demonstrates the concept of symbol interposition in Shiva.

Testing our patch: ro_patch.o

In this example we are patching the AArch64 ELF PIE executable: "modules/aarch64_patches/rodata_interposing/test_rodata" Let's first run the program normally.

```
$ ./test_rodata
rodata_string: Arcana Technologies
$
```

The program simply prints the string "rodata_string: Arcana technologies". To see if our patch worked we can invoke Shiva directly to load and link the executable with the patch at runtime. We must specify the path to our patch object with the SHIVA_MODULE_PATH environment variable.

```
$ SHIVA_MODULE_PATH=./ro_patch.o /lib/shiva ./test_rodata
rodata_string: The Great Arcanum
$
```

When executing ./test_rodata with "/lib/shiva" as the primary loader it installs the specified patch, and the new rodata_string[] value "The Great Arcanum" is printed to stdout. When we run ./test_rodata without "/lib/shiva" it does not install the patch and the original string "Arcana Technologies" is printed.

Using shiva-ld to prelink the patch

The reality is that a user doesn't want to invoke "/lib/shiva" every single time they want to run a program that needs a patch. This is why "/lib/shiva" is designed to be an ELF interpreter like "ld-linux.so". We can use the Shiva prelinker "/bin/shiva-ld" to install the appropriate meta-data into the program we are patching:

- Update PT_INTERP with the path to "/lib/shiva" (Replacing "/lib/ld-linux.so").
- Add several custom entries to PT_DYNAMIC describing the patch basename: "ro_patch.o" and the module search path "/opt/shiva/modules".

The following command will install the Shiva interpreter path and patch information into the ./test_rodata binary.

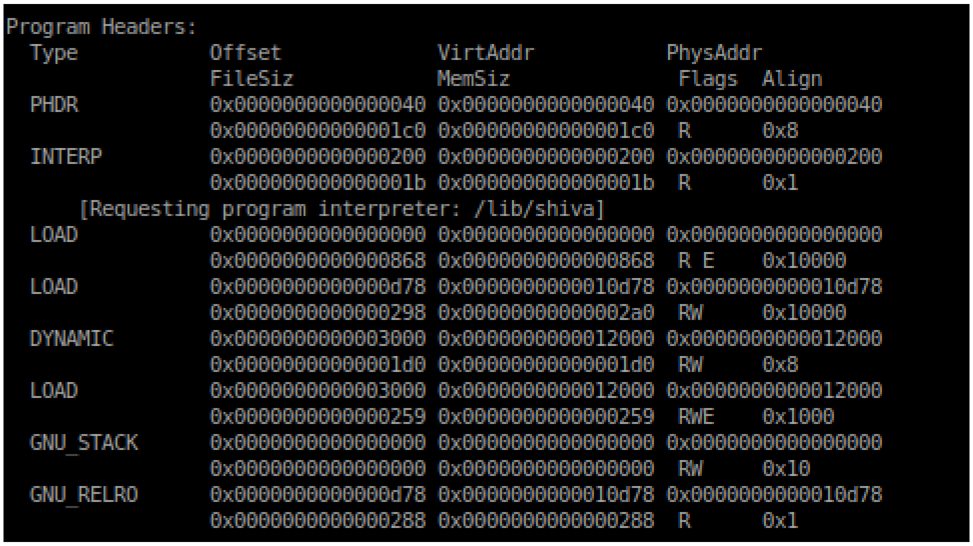
```
$ shiva-ld  -e test_rodata -p ro_patch.o -i /lib/shiva -s /opt/shiva/modules
```

This next command copies the patch file into the correct search path: "/opt/shiva/modules"

```
$ sudo cp ro_patch.o /opt/shiva/modules
```

Before we run ./test_rodata, let's take a quick look to see what modifications shiva-ld made to the executable:

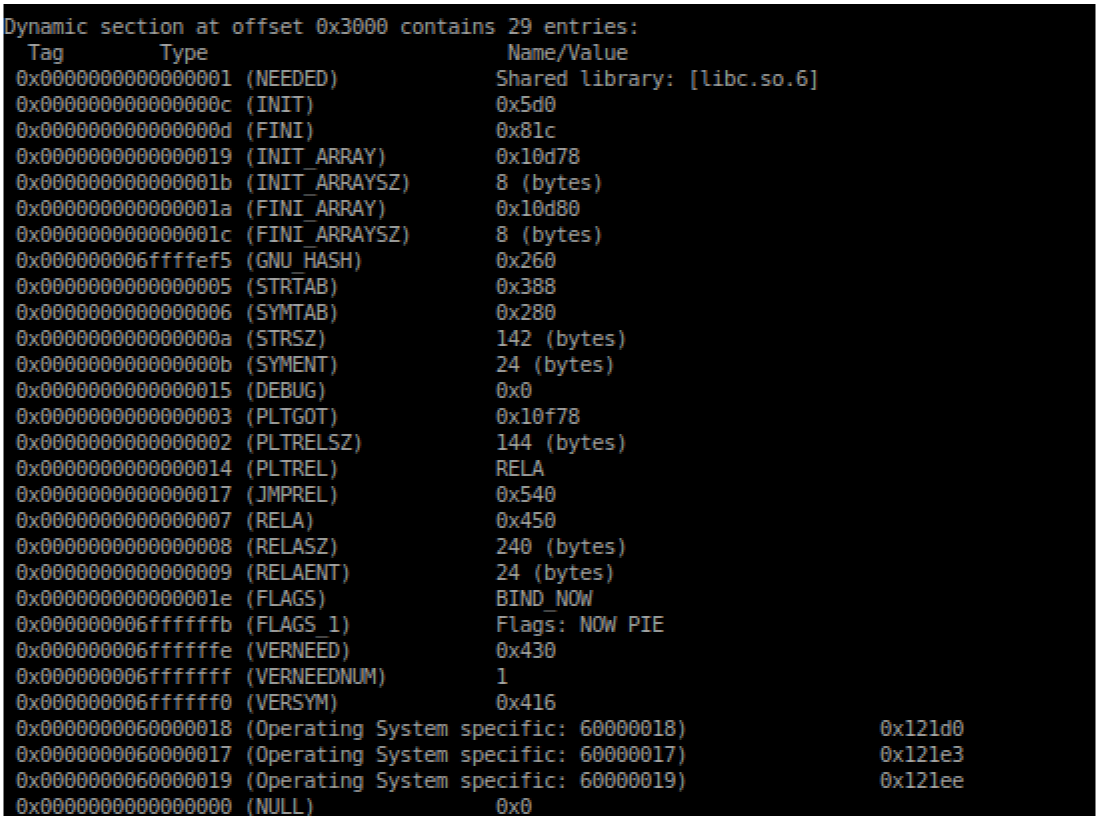
Image 3.5: Requesting program interpreter is "/lib/shiva"



```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001c0 0x00000000000001c0  R      0x8
  INTERP         0x0000000000000200 0x0000000000000200 0x0000000000000200
                 0x000000000000001b 0x000000000000001b  R      0x1
      [Requesting program interpreter: /lib/shiva]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000868 0x0000000000000868  R E    0x10000
  LOAD           0x0000000000000d78 0x0000000000010d78 0x0000000000010d78
                 0x0000000000000298 0x00000000000002a0  RW     0x10000
  DYNAMIC        0x0000000000003000 0x0000000000012000 0x0000000000012000
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  LOAD           0x0000000000003000 0x0000000000012000 0x0000000000012000
                 0x0000000000000259 0x0000000000000259  RWE    0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000d78 0x0000000000010d78 0x0000000000010d78
                 0x0000000000000288 0x0000000000000288  R      0x1
```

In image 3.5 we can see that "/lib/shiva" is set as the new Interpreter path. The astute reader will also notice that there is a third PT_LOAD segment marked RWE. The shiva-ld tool creates a new dynamic segment (As shown below in Image 2.3). In order to make room for this new and larger dynamic segment shiva-ld creates a new PT_LOAD segment and updates the PT_DYNAMIC segment program header so that it points into the new PT_LOAD segment at 0x3000. The updated PT_DYNAMIC segment is moved and stored in this new segment

Image 3.6: New dynamic segment contains 3 custom entries



```
Dynamic section at offset 0x3000 contains 29 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x5d0
 0x000000000000000d (FINI)               0x81c
 0x0000000000000019 (INIT_ARRAY)         0x10d78
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x10d80
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x260
 0x0000000000000005 (STRTAB)             0x388
 0x0000000000000006 (SYMTAB)             0x280
 0x000000000000000a (STRSZ)              142 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x10f78
 0x0000000000000002 (PLTRELSZ)           144 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x540
 0x0000000000000007 (RELA)               0x450
 0x0000000000000008 (RELASZ)             240 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000000000001e (FLAGS)              BIND_NOW
 0x000000006ffffffb (FLAGS_1)            Flags: NOW PIE
 0x000000006ffffffe (VERNEED)            0x430
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x416
 0x0000000060000018 (Operating System specific: 60000018)                0x121d0
 0x0000000060000017 (Operating System specific: 60000017)                0x121e3
 0x0000000060000019 (Operating System specific: 60000019)                0x121ee
 0x0000000000000000 (NULL)               0x0
```

The above *image 3.6* shows the dynamic segment of the ELF binary: test_rodata, after we ran the shiva-ld tool on it. Notice the three *Operating System specific* entries at the tail end. The readelf utility doesn't know how to parse these custom entries. Here is what the actual entry types and values are:

(SHIVA_NEEDED)	**Patch object: [ro_patch.o]**
(SHIVA_SEARCH)	**Search path: [/opt/shiva/modules]**
(SHIVA_ORIG_INTERP)	**Original RTLD: [/lib/ld-linux-aarch64.so.1]**

This data is needed by "/lib/shiva" at runtime so that it can locate and load the patch object, and the original dynamic linker (See Chained Linking in section x.x).

Final outcome of patching test_rodata

Now every time that the test_rodata executable is ran it will invoke the Shiva interpreter which will in turn load the patch object "/opt/shiva/modules/ro_patch.o".

```
$ ./test_rodata
rodata_string: The Great Arcanum
$
```

We can observe that the program test_rodata is now printing the patched version of *rodata_string []* when it is ran. This patch may appear permanent, but in effect the patch is actually being installed at runtime by “/lib/shiva” everytime the program runs.

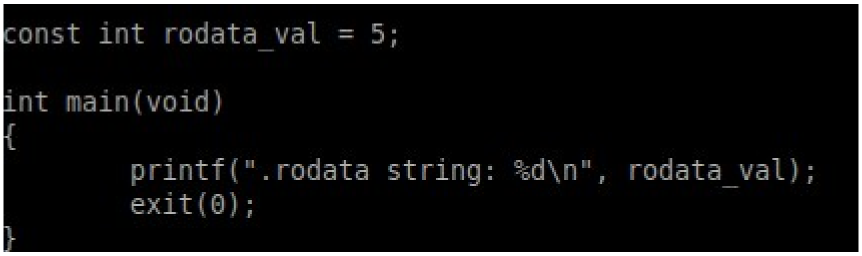
Current limitations of patching const data

There are patching scenarios with *const* data that fail with symbol interposition due to code optimizations with read-only data. In the previous example we illustrated how to patch a read-only global variable defined in C as:

const char rodata_string[] = “Arcana Technologies”;

The target program gets re-linked to use the patch version of *rodata_string*. Using symbol interposition to overwrite *.rodata* variables works perfectly unless the read-only value is optimized out of memory and stored only in a register. Let’s look at the following example:

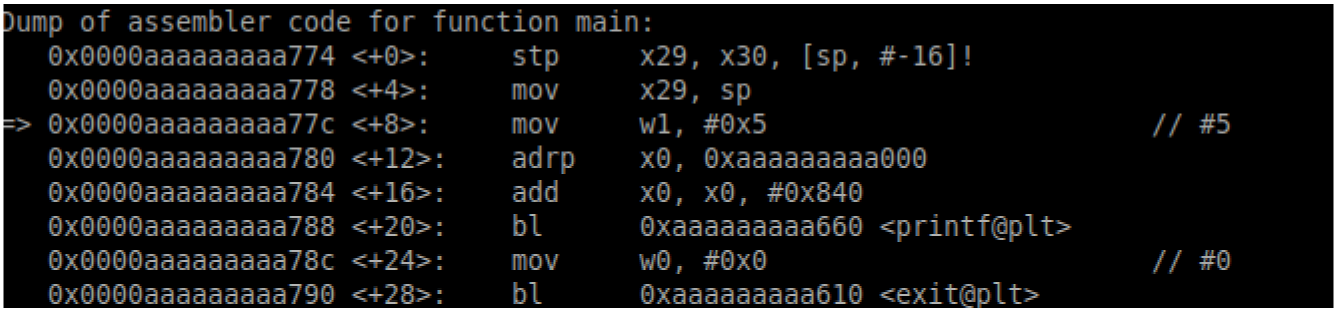
Image 3.7: C code illustrating a read-only global variable



```
const int rodata_val = 5;

int main(void)
{
        printf(".rodata string: %d\n", rodata_val);
        exit(0);
}
```

After compiling the code above with gcc, even after all optimizations are disabled, you will get code similar to the following AArch64 assembly.

Image 3.8: Code optimization of read-only variable in aarch64 assembly



```
Dump of assembler code for function main:
   0x0000aaaaaaaaa774 <+0>:     stp     x29, x30, [sp, #-16]!
   0x0000aaaaaaaaa778 <+4>:     mov     x29, sp
=> 0x0000aaaaaaaaa77c <+8>:     mov     w1, #0x5                        // #5
   0x0000aaaaaaaaa780 <+12>:    adrp    x0, 0xaaaaaaaaa000
   0x0000aaaaaaaaa784 <+16>:    add     x0, x0, #0x840
   0x0000aaaaaaaaa788 <+20>:    bl      0xaaaaaaaaa660 <printf@plt>
   0x0000aaaaaaaaa78c <+24>:    mov     w0, #0x0                        // #0
   0x0000aaaaaaaaa790 <+28>:    bl      0xaaaaaaaaa610 <exit@plt>
```

In the code above we can see that the value 5 is being copied as an immediate value into a register. This is an optimization that removes the need for an adrp/add/ldr instruction trio, thus eliminating two instructions and the need for memory access. This optimization is inauspicious for symbol interposition based patching since the actual *.rodata* variables memory is not being accessed and therefore there is no linking code to update. Instead there is only an instruction: "mov w1, #0x5". Clearly this instruction can be re-encoded rather simply, but what would a patch look like for this? For the curious reader, move ahead to the section titled *"ELF Transformations & Function splicing".*

Symbol interposition on functions

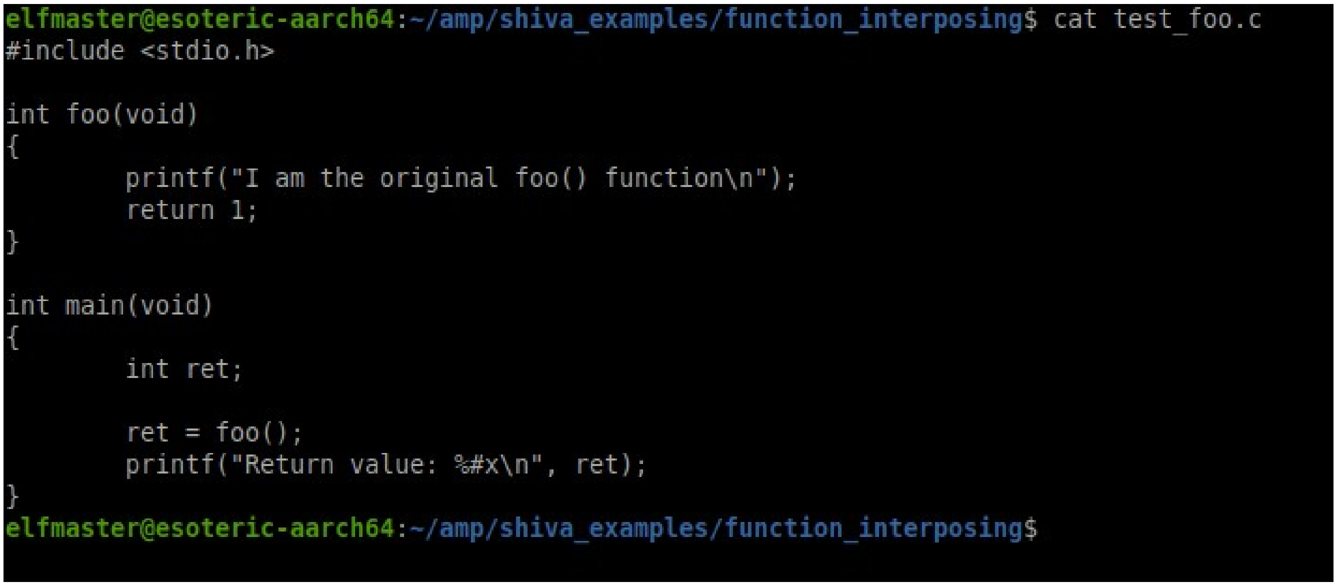
Shiva gives patch writers the ability to replace a function trivially using symbol interposition. We've all seen userland rootkits that re-write libc.so functions by introducing a preloaded library with a new version of the function. In the same spirit Shiva allows patch writers to replace any function in the target ELF executable with a symbol of type STT_FUNC and symbol bindings of STB_GLOBAL or STB_WEAK. The only exception is that the function main() cannot be replaced, although future support is on the way, as it depends on the glibc initialization code that transfers control to main from _start.

Let's get started with a simple example of replacing a function called *foo()*.

Patch exercise 2: Replacing an entire function, foo()

```
$ cd shiva_examples/function_interposing
```

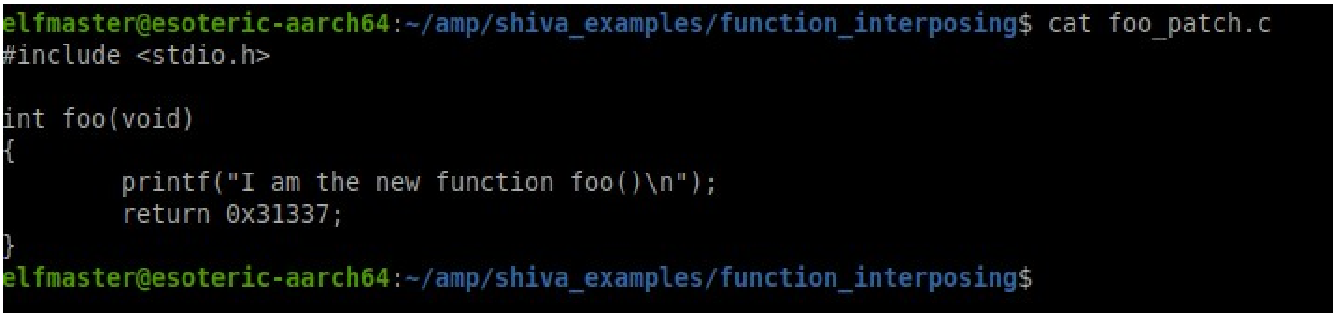
Image 3.9: Original source code of the binary *test_foo*



```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ cat test_foo.c
#include <stdio.h>

int foo(void)
{
        printf("I am the original foo() function\n");
        return 1;
}

int main(void)
{
        int ret;

        ret = foo();
        printf("Return value: %#x\n", ret);
}
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$
```

From the original source code shown in *Image 3.9* we can clearly see that foo() is a global function called by main(). In order to replace foo() with our own version we can simply re-write the function by name in our patch source code.

Image 3.10: foo_patch.c source code



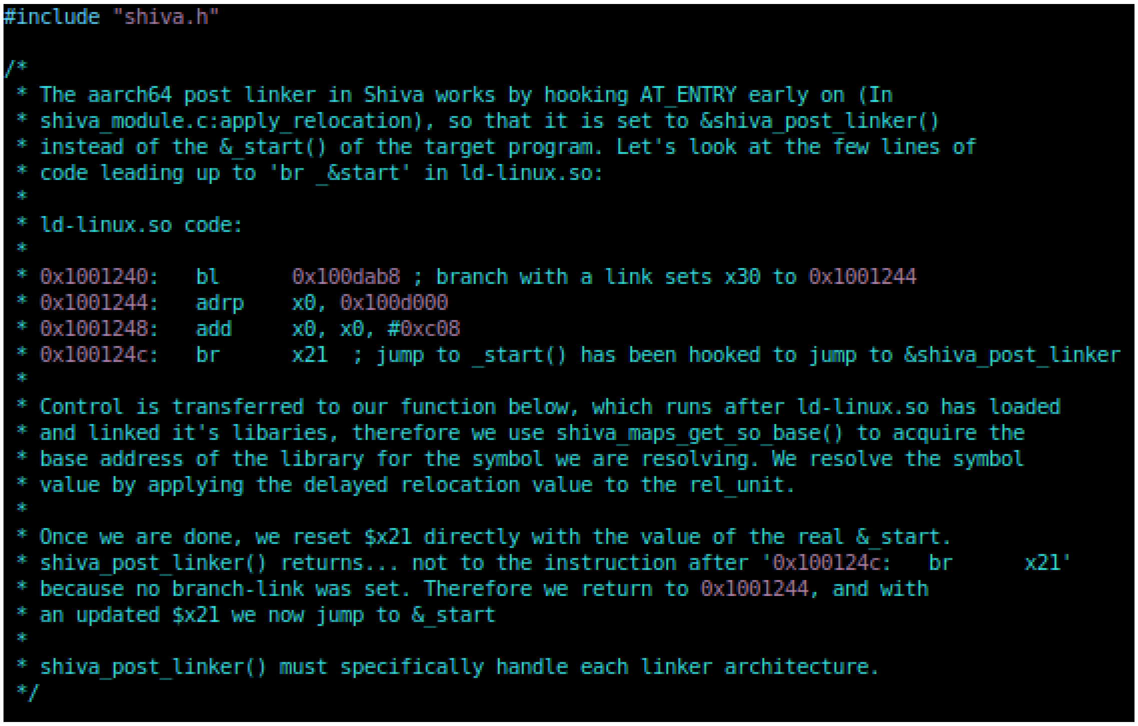
```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ cat foo_patch.c
#include <stdio.h>

int foo(void)
{
        printf("I am the new function foo()\n");
        return 0x31337;
}
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$
```

Shiva handles handles all of the linking at runtime for the patch, including shared library calls. Shiva works in conjunction with the ld-linux.so interpreter to accomplish these goals. All call instructions *(i.e. branch with link)* are given a PLT entry within the patch module at runtime. The patch has it's own PLT/GOT and the GOT is filled in at runtime with the address to every function *(Both in and out of the module)* that are being called. Shiva handles all of the runtime relocations prior to passing control to ld-linux.so except for when the relocation references symbolic code or data that live within a shared library. A call to libc:printf() from within the patch code have it's own respective GOT entry. The address of printf() won't be known until ld-linux.so has mapped the shared libaries into memory and procesed their relocations. To handle this scenario Shiva incorporates a ***Post Linker*** which waits until ld-linux.so has processed all of the symbol relocations for each shared library, and then finalizes fixing up the patch modules GOT with the addresses to each respective shared library symbol.

Shiva Post Linker

The Shiva Post Linking Phase is a great example of what Shiva refers to as Cross-ELF-Relocations ***(Defined as a relocation who's computation relies on the results of a secondary relocation solved by an entirely separate runtime linker - One linker must wait on the other to satisfy the Cross-Relocation)*** In particular though, the Shiva-Post-Linker is a linking mechanism that delays certain relocations from being processed until the standard RTLD "ld-linux.so" has finished mapping in each shared library and processing their respective relocations. In the event that Shiva has to fill in the PLT/GOT entry to resolve a function, such as printf() in the patch code above for foo(), it must rely on the dynamic linker to load and link libc.so:printf() first. After *ld-linux.so* is finished, it must somehow control must somehow be transferred back to Shiva, particularly to the function shiva_post_linker(). To make this happen we actuallyTo give some context, here is a screenshot of the shiva post linkers source code.

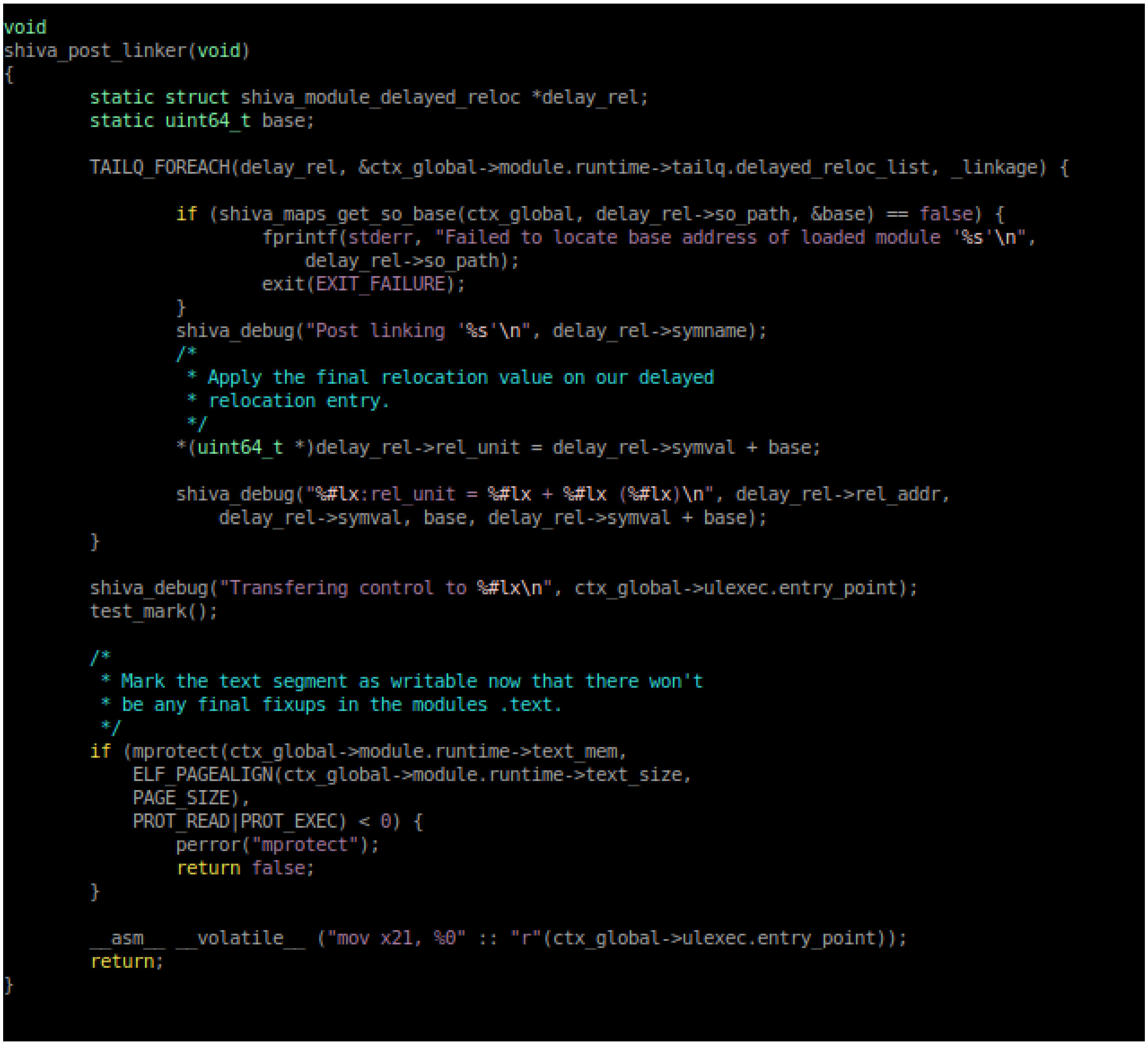
Image 3.11: shiva_post_linker.c comments



```
#include "shiva.h"

/*
 * The aarch64 post linker in Shiva works by hooking AT_ENTRY early on (In
 * shiva_module.c:apply_relocation), so that it is set to &shiva_post_linker()
 * instead of the &_start() of the target program. Let's look at the few lines of
 * code leading up to 'br _&start' in ld-linux.so:
 *
 * ld-linux.so code:
 *
 * 0x1001240:   bl      0x100dab8 ; branch with a link sets x30 to 0x1001244
 * 0x1001244:   adrp    x0, 0x100d000
 * 0x1001248:   add     x0, x0, #0xc08
 * 0x100124c:   br      x21  ; jump to _start() has been hooked to jump to &shiva_post_linker
 *
 * Control is transferred to our function below, which runs after ld-linux.so has loaded
 * and linked it's libaries, therefore we use shiva_maps_get_so_base() to acquire the
 * base address of the library for the symbol we are resolving. We resolve the symbol
 * value by applying the delayed relocation value to the rel_unit.
 *
 * Once we are done, we reset $x21 directly with the value of the real &_start.
 * shiva_post_linker() returns... not to the instruction after '0x100124c:   br      x21'
 * because no branch-link was set. Therefore we return to 0x1001244, and with
 * an updated $x21 we now jump to &_start
 *
 * shiva_post_linker() must specifically handle each linker architecture.
 */
```

In the above *image 3.11* the comments describe how Shiva actually replaces the auxiliary vector value for AT_ENTRY on the stack, which normally contains the address to _start() in the ELF executable that is running, and ld-linux.so passes final control to this address. Shiva takes advantage of this and hooks AT_ENTRY with the address of &shiva_post_linker(); The post linker introduces the concept of *Delayed relocations*. It is simply the concept that the linker notes a relocation entry for delayed fixups; meaning delayed until the ld-linux.so finishes loading and linking all of the shared libraries, at which point control is passed back to &shiva_post_linker() to apply the delayed relocation entries.

Image 3.12: shiva_post_linker() source code



```
void
shiva_post_linker(void)
{
        static struct shiva_module_delayed_reloc *delay_rel;
        static uint64_t base;

        TAILQ_FOREACH(delay_rel, &ctx_global->module.runtime->tailq.delayed_reloc_list, _linkage) {

                if (shiva_maps_get_so_base(ctx_global, delay_rel->so_path, &base) == false) {
                        fprintf(stderr, "Failed to locate base address of loaded module '%s'\n",
                            delay_rel->so_path);
                        exit(EXIT_FAILURE);
                }
                shiva_debug("Post linking '%s'\n", delay_rel->symname);
                /*
                 * Apply the final relocation value on our delayed
                 * relocation entry.
                 */
                *(uint64_t *)delay_rel->rel_unit = delay_rel->symval + base;

                shiva_debug("%#lx:rel_unit = %#lx + %#lx (%#lx)\n", delay_rel->rel_addr,
                    delay_rel->symval, base, delay_rel->symval + base);
        }

        shiva_debug("Transfering control to %#lx\n", ctx_global->ulexec.entry_point);
        test_mark();

        /*
         * Mark the text segment as writable now that there won't
         * be any final fixups in the modules .text.
         */
        if (mprotect(ctx_global->module.runtime->text_mem,
            ELF_PAGEALIGN(ctx_global->module.runtime->text_size,
            PAGE_SIZE),
            PROT_READ|PROT_EXEC) < 0) {
                perror("mprotect");
                return false;
        }

        __asm__ __volatile__ ("mov x21, %0" :: "r"(ctx_global->ulexec.entry_point));
        return;
}
```

In *Image 3.11* above, The shiva_post_linker() source code illustrates how it handles delayed relocations. Particularly (S + B) which is symbol value + base address. The final relocation value is stored in the patch modules GOT entry for the shared library function that is being called. The execution flow goes something like this:

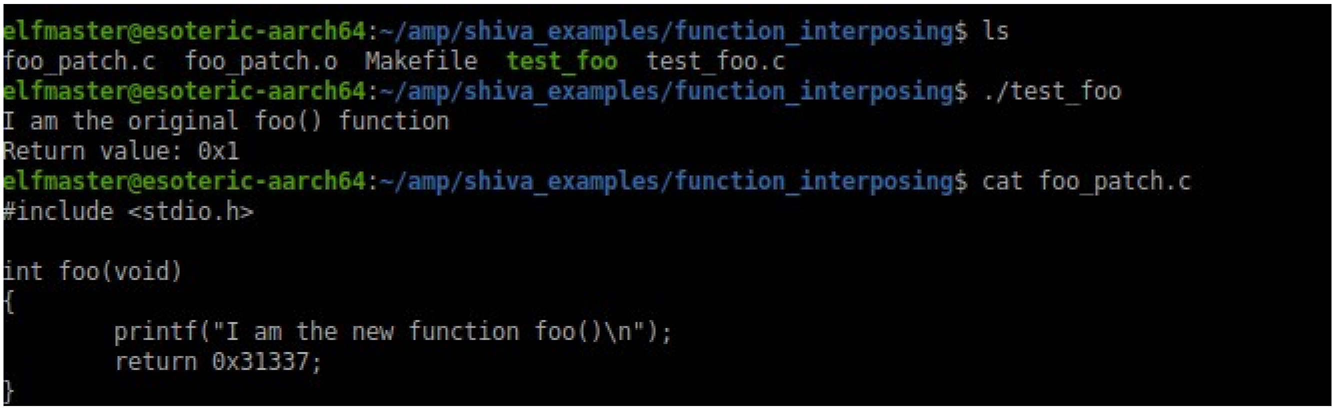
[shiva runtime linker] → [ld-linux.so linker] → [shiva_post_linker] → [_start() in exe]

We reset the value in register x21 back to the original entry point of the executable in &_start().

Relocations and symbol data in foo_patch.o

Without further contemplation lets jump right into the patch directory: "shiva_examples/function_interposing"

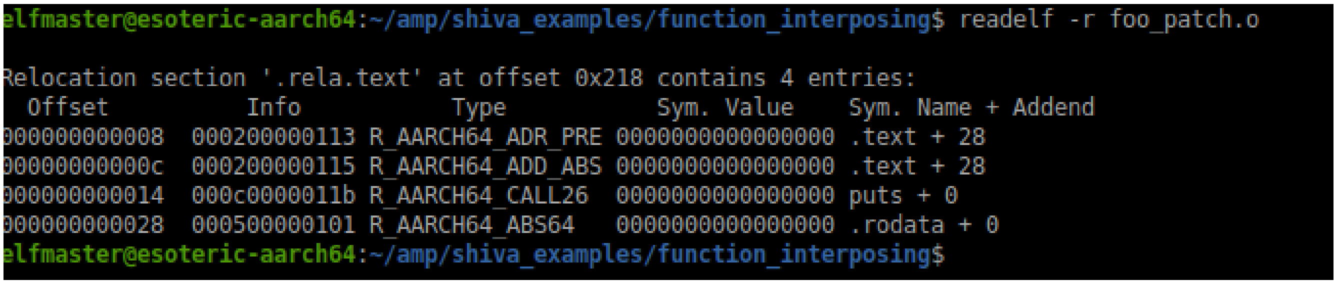
Image 3.12: View the patch files in directory "function_interposing"



```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ ls
foo_patch.c  foo_patch.o  Makefile  test_foo  test_foo.c
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ ./test_foo
I am the original foo() function
Return value: 0x1
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ cat foo_patch.c
#include <stdio.h>

int foo(void)
{
        printf("I am the new function foo()\n");
        return 0x31337;
}
```

Notice the patch above 'foo_patch.c' is simply a re-write of the function foo(). At runtime Shiva will place this new version of foo() into the patch modules text segment. A quick look at it's relocation entries:

Image 3.13: Relocation entries for program ./test_foo



```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ readelf -r foo_patch.o

Relocation section '.rela.text' at offset 0x218 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000008  000200000113 R_AARCH64_ADR_PRE 0000000000000000 .text + 28
00000000000c  000200000115 R_AARCH64_ADD_ABS 0000000000000000 .text + 28
000000000014  000c0000011b R_AARCH64_CALL26  0000000000000000 puts + 0
000000000028  000500000101 R_AARCH64_ABS64   0000000000000000 .rodata + 0
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$
```

The relocations for our patch foo_patch.o will be processed and applied by Shiva at runtime. Here is an explanation of the relocations for our patch, while not yet discussing the re-linking of the target executable that has no viable relocation meta-data for Shiva.

- The first relocation is to to be applied at offset 0x8 within the *.text* section of our module. This relocation is symbolically referencing *.text* + 28, which is going to eventually contain the address to *.rodata + 0*, where the string "I am the new function foo()" lives. To be specific, this relocation is re-encoding an *adrp* instruction to return the page address of the *.text* encoding at 0x28

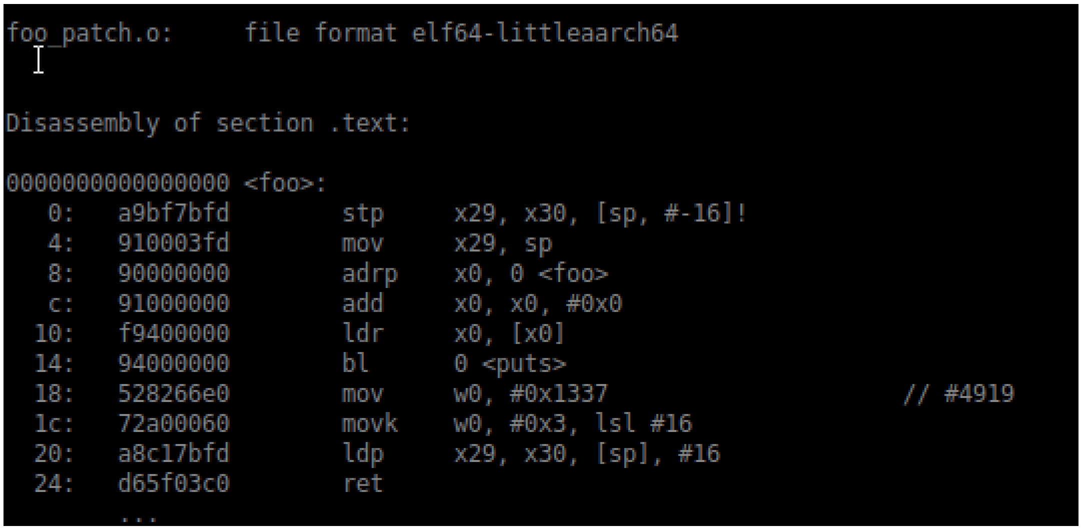
  NOTE: *Shiva refers to a relocation such as this, as a .text on .text relocation. Meaning that the relocation (.rela.text) table applies to the .text section fixups, and when one of those .text fixups references an offset into the .text; I.e. symbolically referencing another place within the .text. This becomes important to Shiva when it's using more advanced features such as function splicing transforms. Therefore ".text on .text relocation" :)*

- The second relocation is adding the offset of the *.text* encoding so that the following *ldr* instruction can indirectly access the address from the *.text* encoding and store it into a register as a pointer to the string "I am the new function foo()".
- The third relocation is a CALL26 to re-encode the *bl* instruction with the 26bit offset to the patches PLT entry for libc.so:puts().
- The last relocation is the one that fills in the *.text* encoding that should contain the address of *.rodata + 0x28.*

  NOTE: *The .text encodings often act as an indirect reference to something like a GOT[] does, infact...The binutils linker "/bin/ld" generally moves these .text encodings into the .got[] when the final executable is built. However it is not a necessary linking procedure. Shiva modules do actually have a designated GOT in the modules data segment at runtime and is used exclusively for PLT/GOT linking of the CALL26 relocations.*

Let's take a look at the patch object file disassembly for function foo() and compare the instruction offsets to those specified in the relocations above.

Image 3.14: Disassembly of foo_patch.o:foo() before relocation happens



```
foo_patch.o:     file format elf64-littleaarch64


Disassembly of section .text:

0000000000000000 <foo>:
   0:   a9bf7bfd        stp     x29, x30, [sp, #-16]!
   4:   910003fd        mov     x29, sp
   8:   90000000        adrp    x0, 0 <foo>
   c:   91000000        add     x0, x0, #0x0
  10:   f9400000        ldr     x0, [x0]
  14:   94000000        bl      0 <puts>
  18:   528266e0        mov     w0, #0x1337                     // #4919
  1c:   72a00060        movk    w0, #0x3, lsl #16
  20:   a8c17bfd        ldp     x29, x30, [sp], #16
  24:   d65f03c0        ret
        ...
```

The code shown above in *Image 3.14* is the objdump output of our replacement *foo() function.* Resolving relocations to build our own modules runtime image is based on the relocatable nature of the object file and the symbolic scope that Shiva uses to link references and calls against. This code can be relocated fairly easy by Shiva's standard linking capabilities. The next part to solving the patching at runtime is re-linking the code in the target executable (*in memory*) so that any calls to *foo()* are replaced with calls to the new location of *foo()* within the patches executable image. External re-linking can be a trifle difficult at times due to a lack of relocation records describing how to re-link *.text* code.

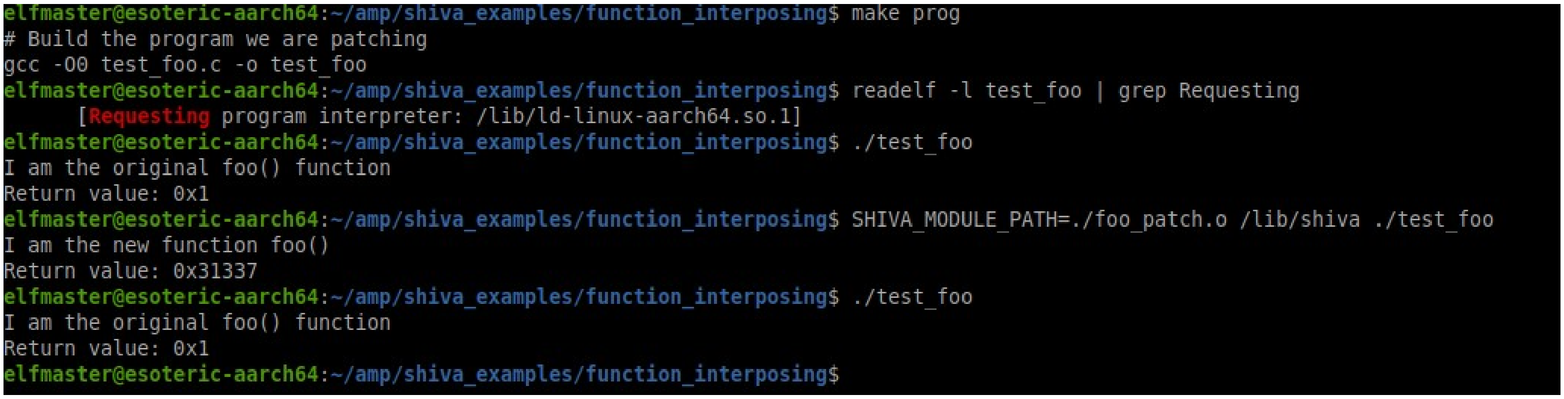
Shiva external re-linking of the ELF executable

Shiva's initial linking job is to ensure that a runtime image can be setup for the patch module, in this case it is foo_patch.o. The ELF sections in this file are copied into heir respective memory segments that Shiva creates with mmap; a text segment (*.text, .plt, .rodata, .shiva_transform, .shiva_CFG)* and a data segment (*.got, .data, .bss*). Relocations are driven by section offsets, and thus Shiva creates an internal data structure pointing to where each section lives within the patches executable environment. *(i.e. .text, .data, .bss section within the modules text segment)*. Shiva sets up an executable process image for the patch right next to the target executable in memory. Now that the patch code is ready for execution, the target program we are patching hasn't yet been re-linked. Historically ELF executables have never stored the more granular ET_REL style relocations because they aren't necessary as the program has already been linked into an executable. The only relocation data in an executable is for the initialization code or dynamic linker to resolve global symbols from shared objects. In this particular case Shiva needs to know the location of each call instruction to foo() and re-link the instruction to the replacement foo() from our foo_patch.o module. Ideally Shiva would have in this case an R_AARCH64_CALL26 relocation, but we haven't really needed such relocation types as this one outside of linking object multiple files. Shiva re-links object files into executables at runtime, as it is a patching engine. Shiva scans the *.text* section at runtime to build all of the CFG data. See **shiva_analyze.c:shiva_find_calls()** which very early on parses every instruction in the target executables *.text* section generating calls/xrefs to code and data. This can be seen in the following function **shiva_module.c:apply_external_patch_links() --** re-linking the target executable's instructions to patch data and patch code. By the time you read this user-manual though, all of the CFG analysis will be generated in the Prelinker: "shiva-ld" a tool which stores all of the control flow graph data into a custom ELF section in the patch module (i.e. ELF section called *.shiva.CFG*). At runtime Shiva will simply read the CFG data in from this custom section who's meta-data provides the correct relocations for where in the target executable we must re-link. This CFG and relocation data is generated by the Shiva prelinker: shiva-ld, which only has to run once vs. everytime Shiva runs. This will reduce Shiva's load time dramatically.

Patch example 2: Apply Function interposing patch, replace foo()

We've spent the last little while discoursing on Shiva's internals to give some insights into how ELF symbolic resolution, relocations, and cross relocations work together to support the JIT re-writing, re-encoding and transformation of the process image: Merging the patch and the executable into a single process image, eternally betrothed through linkage.

Image 3.15: Patch example 2, Replacing function foo()



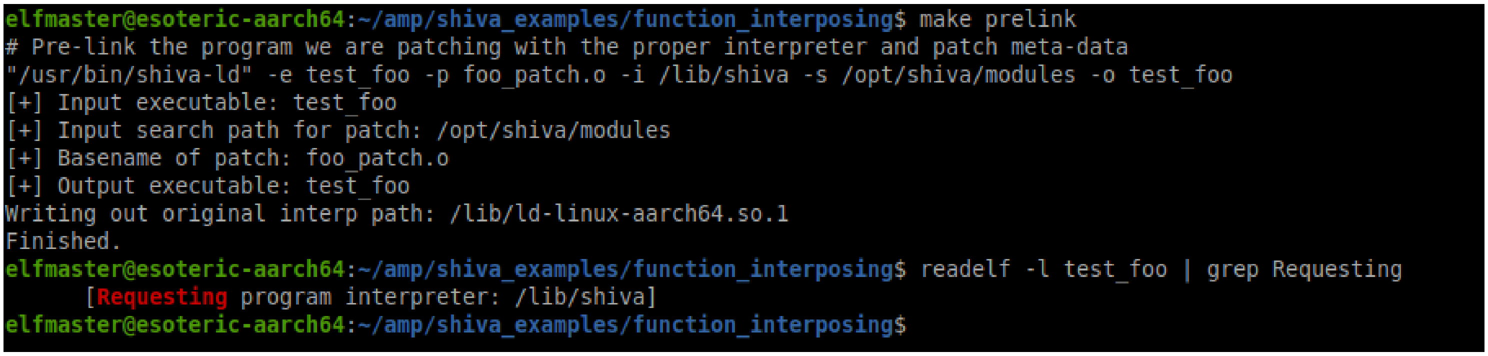
```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ make prog
# Build the program we are patching
gcc -O0 test_foo.c -o test_foo
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ readelf -l test_foo | grep Requesting
      [Requesting program interpreter: /lib/ld-linux-aarch64.so.1]
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ ./test_foo
I am the original foo() function
Return value: 0x1
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ SHIVA_MODULE_PATH=./foo_patch.o /lib/shiva ./test_foo
I am the new function foo()
Return value: 0x31337
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ ./test_foo
I am the original foo() function
Return value: 0x1
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$
```

In the example above, *Image 3.15*, we illustrate that the target ELF binary: test_foo, is not using Shiva as the interpreter, but rather the standard *"/lib/ld-linux.aarch64.so.1"*. We want to test our patch out before prelinking the patch meta-data with Shiva-ld. Again we can invoke "/lib/shiva" directly and it assumes responsibility for loading the executable you are patching, rather than the kernel doing it

NOTE: *See grugqs classic paper on engineering userland execve implementation for anti-forensics. So much mileage out of this great technique over the years.*

Shiva-ld tool. Prelink executable ./test_foo with foo_patch.o

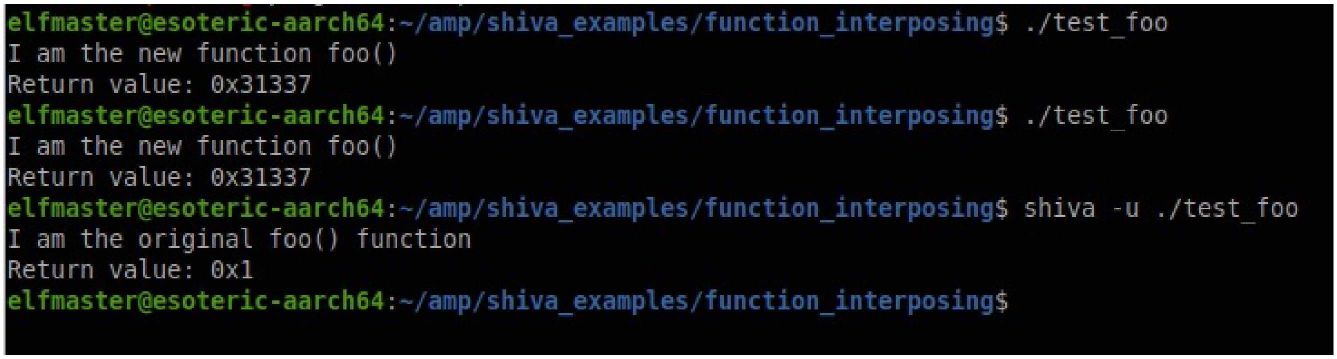
Image 3.16: Prelinking the patch: foo_patch.o to the executable: test_foo



```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ make prelink
# Pre-link the program we are patching with the proper interpreter and patch meta-data
"/usr/bin/shiva-ld" -e test_foo -p foo_patch.o -i /lib/shiva -s /opt/shiva/modules -o test_foo
[+] Input executable: test_foo
[+] Input search path for patch: /opt/shiva/modules
[+] Basename of patch: foo_patch.o
[+] Output executable: test_foo
Writing out original interp path: /lib/ld-linux-aarch64.so.1
Finished.
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ readelf -l test_foo | grep Requesting
      [Requesting program interpreter: /lib/shiva]
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$
```

Execute "test_foo" after shiva-ld has prelinked it

Image 3.17: Executing test_foo with Shiva as ELF interpreter



```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ ./test_foo
I am the new function foo()
Return value: 0x31337
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ ./test_foo
I am the new function foo()
Return value: 0x31337
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$ shiva -u ./test_foo
I am the original foo() function
Return value: 0x1
elfmaster@esoteric-aarch64:~/amp/shiva_examples/function_interposing$
```

Notice from above in *Image 3.17* that the ELF program test_foo executes and calls the replacement *foo()* function correctly from within the foo_patch.o module. The test_foo binary has been shiva-prelinked and contains the updated PT_INTERP segment with "/lib/shiva" replacing "/lib/ld-linux-aarch64.so.1". Every time the program executes the kernel invokes Shiva as the interpreter, which in turn performs its LEJIT operations (Linking & Encoding, Just in time). The last command in *Image 3.17* is note worthy too:

```
$ shiva -u ./test_foo
I am the original foo() function
Return value: 0x1
$
```

Notice that Shiva is invoked directly this time and we specify the -u flag which essentially means "undo any patch prelinking". The program "test_foo" now executes without the patches installed again. This illustrates the flexibility of a modular patching system well, as it's workflow is more resilient than static patching.

Symbol interposition on global data: .rodata, .bss, .data

Initialized global variables (that are not const) are stored within the *.data* section.

```
$ cd modules/aarch64_patches/dataonly_interposing
```

Original source code of binary

Redefining global data types

Patching STB_LOCAL variables

Using the extern keyword in your patch code

Symbolic patching of stripped ELF's