



# A friendly user guide to ELF micropatching with Shiva (alpha v0.11)

## Shiva Manual

1 Introduction to Shiva.....	2
Shiva is LEJIT!.....	2
ELF program interpreters.....	3
Shiva is an ELF program interpreter.....	4
Systems and arch's that Shiva supports.....	4
2 Installing Shiva-AArch64.....	4
Download Source code.....	4
Build source code.....	4
Build and install libelfmaster.....	5
Build and install Shiva.....	5
Build and install example patches.....	5
3 Writing patches with Shiva.....	6
The Shiva workflow.....	6
ELF Program compilation and linking example.....	6
Static linking of objects.....	6
Dynamically linked executable.....	7
Dynamically patched executable.....	8

ELF Symbol interposition.....	8
Patching an .rodata string (Simple first patch).....	9
Building the patch.....	9
Testing our patch: ro_patch.o.....	10
Using shiva-ld to prelink the patch.....	10
Final outcome of patching test_rodata.....	13
Current limitations of patching const data.....	13
Symbol interposition on functions.....	14
Patch exercise 2: Replacing function foo().....	14
Shiva Post Linker.....	16
Relocations and symbol data in foo_patch.o.....	18
Shiva external re-linking of the ELF executable.....	20
Patch example 2: Apply Function interposing patch, replace foo().....	21
Shiva-ld tool. Prelink executable ./test_foo with foo_patch.o.....	22
Execute “test_foo” after shiva-ld has prelinked it.....	22
Symbol interposition on global data: .rodata, .bss, .data.....	23
Re-declaring the size of global variables.....	23
Fixing .bss overflow by increasing buffer size.....	23
Cross-Relocation with .bss relinking.....	25
The algorithm for cross relocations on .bss variables.....	27
Redeclaring global data types, signedness, and size.....	27
Redeclaring global integer signedness.....	27
Redeclaring the global storage type:.....	28
Redeclaring a global .data variable as a constant.....	28
Redeclaring a global constant into a .data variable.....	28
Initializing a .bss variable (Conversion from .bss variable to .data variable).....	29
Symbol interposing on STB_LOCAL symbols.....	29
Symbol interposition on stripped ELF binaries.....	30
Using the extern keyword in your patch code.....	30
Program Transformations: Advanced patching.....	31
Function splicing.....	31
High level technical details.....	32
Patch exercise 3: Patching a strcpy vulnerability.....	33
Demonstrate strcpy_vuln.o patch.....	40
Current limitations and the future of Transforms.....	41
In closing.....	42

## 1 Introduction to Shiva

**Shiva is LEJIT!**

Shiva is a LEJIT micropatching system for Linux ELF software. LEJIT (Linking & Encoding Just In Time). It's purpose is to allows developers to re-program and patch native ELF software.

## 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 0x00000000000010d40 0x00000000000010d40
                 0x00000000000002d0 0x000000000000042e0  RW      0x10000
  DYNAMIC        0x0000000000000d50 0x00000000000010d50 0x00000000000010d50
                 0x0000000000000200 0x0000000000000200  RW       0x8
  NOTE           0x000000000000021c 0x000000000000021c 0x000000000000021c
                 0x0000000000000044 0x0000000000000044  R        0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW       0x10
  GNU_RELRO      0x0000000000000d40 0x00000000000010d40 0x00000000000010d40
                 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".

## 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 three 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.

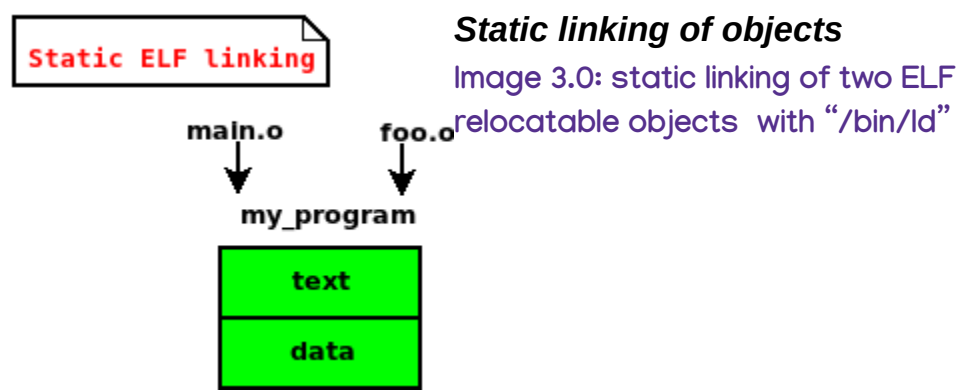
## 3 Writing patches with Shiva

### 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

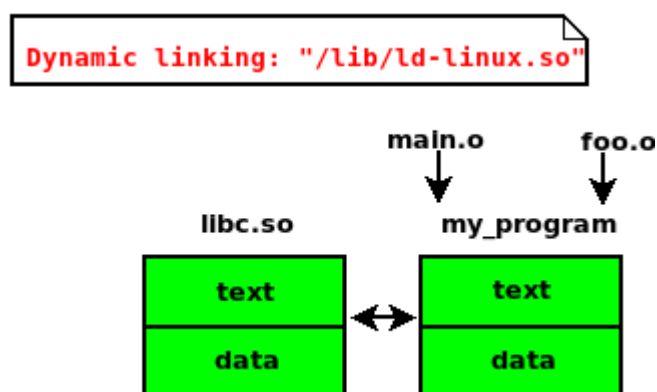
Standard program compilation begins with a compiler like gcc; the source code files (i.e. main.o and foo.o) are compiled and then get linked (By “/bin/ld”) into a final output ELF executable my\_program.



Additionally these ELF relocatable object files generally contain references to code and data 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 program is executed 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 is linked twice: once statically, and again dynamically.

### ***Dynamically linked executable***

Image 3.1: Dynamically link libc.so in at runtime

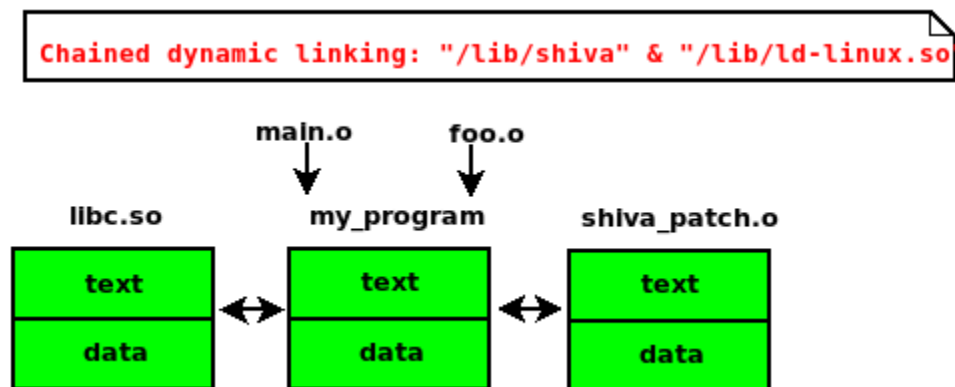


A Shiva patch can be written in C and compiled as an ELF relocatable object file. In the same sense that `"ld-linux.so"` loads and links shared libraries (ELF ET\_DYN files), Shiva will

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



As shown in **Image 3.2** there are two dynamic linkers within the same address space. The Shiva interpreter is responsible for loading and linking `shiva_patch.o` into a proper runtime image. Shiva is also responsible for loading the RTLD “`ld-linux.so`” into memory, and passing control to it. Shiva maintains symbolic resolution that is process-wide for handling relocations and program transformation.

## **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 by simply re-defining it in C.

Image 3.3: Original 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 many 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";
```

### ***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 -mmodel=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		
	0x00000000000001b	0x00000000000001b	R	0x1	
[Requesting program interpreter: /lib/shiva]					
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x0000000000000868	0x0000000000000868	R E	0x10000	
LOAD	0x0000000000000d78	0x0000000000001d78	0x0000000000001d78		
	0x0000000000000298	0x00000000000002a0	RW	0x10000	
DYNAMIC	0x0000000000003000	0x00000000000012000	0x00000000000012000		
	0x00000000000001d0	0x00000000000001d0	RW	0x8	
LOAD	0x0000000000003000	0x00000000000012000	0x00000000000012000		
	0x0000000000000259	0x0000000000000259	RWE	0x1000	
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x0000000000000000	0x0000000000000000	RW	0x10	
GNU_RELRO	0x000000000000d78	0x0000000000001d78	0x0000000000001d78		
	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
0x000000006fffffb (FLAGS_1)       Flags: NOW PIE
0x000000006fffffe (VERNEED)       0x430
0x000000006ffffff (VERNEEDNUM)    1
0x000000006fffff0 (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.

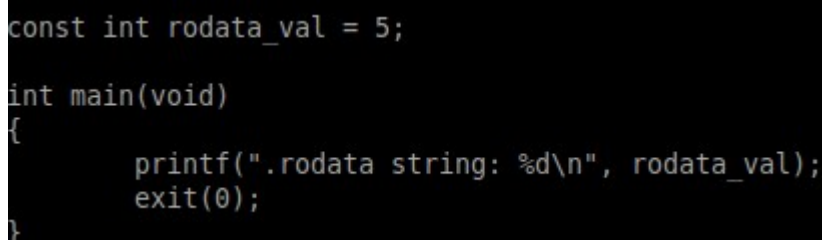
### ***Current limitations of patching const data***

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:
0x0000aaaaaaaa774 <+0>:      stp     x29, x30, [sp, #-16]!
0x0000aaaaaaaa778 <+4>:      mov     x29, sp
=> 0x0000aaaaaaaa77c <+8>:      mov     w1, #0x5                                // #5
0x0000aaaaaaaa780 <+12>:     adrp    x0, 0aaaaaaaa000
0x0000aaaaaaaa784 <+16>:     add     x0, x0, #0x840
0x0000aaaaaaaa788 <+20>:     bl      0aaaaaaaa660 <printf@plt>
0x0000aaaaaaaa78c <+24>:     mov     w0, #0x0                                // #0
0x0000aaaaaaaa790 <+28>:     bl      0aaaaaaaa610 <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 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 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 libraries into memory and processed 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 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 libraries, 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("Transferring 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
00000000000008      000200000113  R_AARCH64_ADR_PRE 0000000000000000   .text + 28
0000000000000c      000200000115  R_AARCH64_ADD_ABS 0000000000000000   .text + 28
00000000000014      000c0000011b  R_AARCH64_CALL26  0000000000000000   puts + 0
00000000000028      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 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`.*

*Definition: The relocation is in `.rela.text` and symbolically links to a `.text` section offset. This becomes important to Shiva when it's using more advanced features such as function splicing transforms. Therefore “`.text on .text relocation`” is a term that will be spoken about in this manual.*

- 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 `b/` instruction with the 26bit offset of 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, in fact... 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. The `.text` encodings are kept in -place by Shiva without moving them into any kind of `.got` table.*

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
I
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 re-link the process image. 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 their respective memory segments that Shiva creates with *mmap*; a text segment includes: *.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 after 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 *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 “userland-exec only, don’t apply patches”. The program “test\_foo” now executes

without the patches installed at runtime. 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**

Thus far Shiva has demonstrated the ability to re-write global functions and global data by name. ELF Symbolic data of any type can be re-defined in a Shiva patch, whether it be a function or a global variable. Let us illustrate a patch that re-writes both global data and several functions all in the same patch, as there is no limitation to the number of global variables and functions that can be interposed.

Shiva doesn't yet support re-writing thread local storage variables, but support for this is on the road map.

## **Re-declaring the size of global variables**

When a variable is re-declared or re-defined in a Shiva patch, the target program is re-linked to reference the new global variable which exists somewhere within the Shiva patch module. Therefore it stands to reason that a variable can be redefined in size, and type. In this example we will demonstrate a patch that re-sizes a *.bss* global buffer in order to fix a ridiculous and hand-crafted vulnerability which results in local privilege escalation.

### ***Fixing .bss overflow by increasing buffer size***

Change directories to "shiva\_examples/bss\_vuln". Let's take a look at the original source code for this vulnerable program.

*Image 3.18: bss\_vuln.c source code*

```

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>

#define MAX_LEN 32

uint8_t uid;
uint8_t bss_buffer[16];

int main(int argc, char **argv)
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file_payload>\n", argv[0]);
        return 0;
    }
    uid = getuid();

    printf("uid: %d\n", uid);

    strncpy(bss_buffer, argv[1], 32);

    printf("Setting uid: %d\n", uid);
    setuid(uid);
    system("/bin/bash");
}

```

In *image 3.18* we can see the original source code to the program “shiva\_examples/bss\_vuln/bss\_vuln.c”. This program illustrates how the function *strncpy* pads the remaining length of a buffer with null bytes and how this behavior can lead to a *.bss* overflow that overwrites the bss variable *uint8\_t uid* with the value 0, which is then passed to *setuid()* before a call to *system()*. In our somewhat unrealistic example, we aim to re-size the variable *bss\_buffer[16]* to *bss\_buffer[32]* to prevent the 0 padded *.bss* overflow.

### *Image 3.19: Running the vulnerable bss\_vuln program*

```

elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ ./bss_vuln hello
uid: 232
Setting uid: 0
root@esoteric-aarch64:~/amp/shiva_examples/bss_overflow# whoami
root
root@esoteric-aarch64:~/amp/shiva_examples/bss_overflow#

```

Running this program, and passing in a string length of less than 32 bytes will cause the bug to be triggered and for dramatic effect we can see the pop of a root shell in *Image 3.19*. And as already shown in the original source code, *bss\_buffer* is an uninitialized static global



variable that lives in the `.bss`. Let's confirm that this global variable does indeed exist within the `.bss` of the target executable before we demonstrate our patch.

*Image 3.20: Verifying `bss_buffer` variable exists in target executable*

```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ readelf -s bss_vuln | grep bss_buffer
76: 00000000000011018 16 OBJECT GLOBAL DEFAULT 22 bss_buffer
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ readelf -S bss_vuln | grep 22
[22] .bss NOBITS 0000000000011010 00001010
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$
```

In *Image 3.20* we confirm that a `.bss` variable `bss_buffer[16]` does exist and its bindings are `STB_GLOBAL`. We can fix this vulnerability by simply re-defining `bss_buffer` as a 32 byte length buffer. Patching the program `bss_vuln` is straightforward, watch...

*Image 3.21: Patching the `bss_vuln` vulnerability by re-defining `bss_buffer`*

```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ cat bss_patch.c
#include <stdint.h>

uint8_t bss_buffer[32]; // extend buffer from 16 to 32bits
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ make patch
gcc -mmodel=large -fno-pic -I ../ -fno-stack-protector -c bss_patch.c
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ shiva-ld -e bss_vuln \
> -i /lib/shiva -p bss_patch.o -s /opt/shiva/modules/ -o bss_vuln
[+] Input executable: bss_vuln
[+] Input search path for patch: /opt/shiva/modules/
[+] Basename of patch: bss_patch.o
[+] Output executable: bss_vuln
Writing out original interp path: /lib/ld-linux-aarch64.so.1
Finished.
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ sudo cp bss_patch.o /opt/shiva/modules
[sudo] password for elfmaster:
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ ./bss_vuln hello
uid: 232
Setting uid: 232
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$
```

As shown above in *Image 3.21* a simple patch re-declaring the size of `bss_buffer[16]` to `bss_buffer[32]` is compiled and pre-linked in just two commands. The program `bss_vuln` is now linked to the `bss_buffer[32]` and doesn't overflow anymore at the call to `strcpy()`.

## **Cross-Relocation with `.bss` relinking**

For those interested in understanding the internals of `.bss` variable relocation in Shiva, please keep reading to understand a key Shiva feature known as *Cross Relocations*.

The .bss is uninitialized data and doesn't take up any space in the ELF file. ELF PIE binaries use R\_AARCH64\_RELATIVE relocation's to fixup the .got with the correct address to the .bss variable after it's been allocated at runtime. In order for Shiva to re-link the binary to use the new .bss variable in the patch it must update the .got entry with the address to the new variable living within the patches runtime image. The RTLD "ld-linux.so" will process the R\_AARCH64\_RELATIVE relocation's and overwrite the .got entry for the .bss variable. because the RTLD runs after Shiva and therefore updates the .got after Shiva. This introduces a little problem that is solved by a concept that we call *Cross Relocation*. It is an almost inevitable by-product of chaining two dynamic linkers together. Cross relocation refers to the idea that Shiva and ld-linux.so (Two linkers in the same process) must synchronize relocation data and work together to solve a relocation. In the case of linking .bss variables Shiva modifies the relocation record in memory. Specifically the r\_addend value to reflect the location of the new .bss variable within the patch image. Let's examine this up close.

### Image 3.22: Lookup the address of the bss\_buffer symbol

```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ readelf -s bss_vuln | grep bss_buffer
76: 00000000000011018 16 OBJECT GLOBAL DEFAULT 22 bss_buffer
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$
```

The bss\_buffer variable is at address 0x11018 (As shown in Image 3.22 above). This address exists past the end of the data segment on disk and will be allocated by the kernel at ELF load time.

### Image 3.23: Find the relative relocation for bss\_buffer by address

```
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$ readelf -r bss_vuln | grep 11018
0000000010fc8 000000000403 R_AARCH64_RELATIV 11018
elfmaster@esoteric-aarch64:~/amp/shiva_examples/bss_overflow$
```

The R\_AARCH64\_RELATIVE relocation in Image 3.23 is describing the computation (B + A), which means base\_address + r\_addend.

The r\_offset (*The address to be patched*) is 0x10fc8. The r\_addend value is 0x11018. The base must be added with the r\_addend and stored at the location specified by r\_offset. The RTLD will apply this like so:

```
*(uint64_t *)&0x10fc8[0] = 0x11018 + base_address
```

Shiva wants to redirect all references to bss\_buffer from the old location 0x11018 to the new location within the patch object, who's address won't be known until runtime. At runtime Shiva calculates the offset from the executable base address to the patches bss\_buffer variable. Shiva overwrites the r\_addend field of the R\_AARCH64\_RELATIVE relocation from 0x11018 to the new offset of the patch bss\_buffer variable.

## ***The algorithm for cross relocations on .bss variables***

1. locate the R\_AARCH64\_RELATIVE relocation who's r\_addend is equal to the offset of the original .bss variable.
2. Calculate the offset of the new patch .bss variable from the base of the executable:  
 $\text{new\_var\_addr} - \text{executable\_base} - 4$
3. Store the offset as the updated r\_addend field in the relocation entry

## **Redeclaring global data types, signedness, and size**

It is common to change the data-type, signedness or size of a variable in order to fix a bug. This is therefore a common requirement in the world of micropatching binaries. Shiva will create a new variable matching the correct data-type and size and relink the program to the new variables. This may or may not have expected results since Shiva doesn't recompile the executable code that is reading or writing to that data. As it stands there are no limitations on changing the integer type and storage location of any global variable with the exception of TLS data. Any STT\_OBJECT with STB\_GLOBAL bindings can be re-linked to any other STT\_OBJECT/STB\_GLOBAL variable within a defined patch. Support for patching STB\_LOCAL binded variables and functions will be supported in the future.

### ***Redeclaring global integer signedness***

When a patch re-declares a global variable's signedness, it will create a new global variable within the patch, and all code within that same patch will be compiled into code that treats the variable by it's new signedness. The code within the executable being patched will be re-linked to use the new global variable, but it's code won't be updated to reflect the change in the variables signedness and therefore the final outcome is somewhat ambiguous.

**Old definition:** `int data_var = 5;` (Defined within the ELF executable program)

**New definition:** `unsigned int data_var = 5;` (Defined within the ELF patch)

**Outcome Results:** Original program code will still see the new variable as signed. New program code introduced by the patch will see the variable as unsigned.

**Outcome Description:** Shiva will allocate `sizeof(int)` bytes in the `.data` section of the patch and relink the executable to use the new `data_var` variable. The signedness was changed from signed to unsigned, but only the code within the patch itself (Assuming there is any) will respect the new signedness since it's code was compiled to see an *unsigned int* `data_var = 5;` whereas the code in the executable being patched was compiled when

*data\_var* was declared as a signed value, and therefore it's computations and read/write accesses will still be treating it as signed.

### ***Redeclaring the global storage type:***

There may be times when the patch operator wants to re-declare an uninitialized global variable (i.e. *.data*) as const (i.e. *.rodata*). Or change an uninitialized *.bss* variable to a *.data* variable, etc. Shiva allows for the re-linking of any STT\_OBJECT type symbols that are globally binded (STB\_GLOBAL).

### ***Redeclaring a global .data variable as a constant***

#### **Example 1:**

Original source code:    uint16\_t pid = 0;  
Patch source code:       const uint16\_t pid = 0;

**Outcome results:** The global initialized variable *pid* becomes read-only.

**Technical details:** Shiva creates an alternate *.rodata* section within the patch text segment, stores the *pid* variable there (*As specified by the patches symbol and relocation table*). Shiva then re-links the executable to use the new *pid* variable by re-encoding any instructions which reference the original *pid* variable.

#### **Example 2:**

Original source code:    char comm[] = "/bin/l\$";  
Patch source code:       const char comm[] = "/bin/l\$";

**Outcome results:** The global initialized variable *comm[]* becomes read-only.

**Technical details:** Shiva creates an alternate *.rodata* section within the patch text segment, stores the *comm[]* buffer there (*As specified by the patch objects symbol and relocation table*). Shiva then re-links the executable to use the new *comm[]* variable by re-encoding any instructions which reference the original *comm[]* variable.

### ***Redeclaring a global constant into a .data variable***

An integer variable in *.rodata* cannot be relinked to a *.data* integer due to the current limitations of Shiva discussed in "*Current limitations of patching .rodata-- page 13*". The good news is that global *.rodata* strings can be relinked to a global *.data* string.

**Supported conversion:** Converting a global constant string (stored in the `.rodata` section) to a non-constant string (stored in the `.data` section).

**Unsupported conversion:** Converting a global constant integer (stored in the `.rodata` section) to a non-constant integer (stored in the `.data` section).

**Original source code:** `const char string[] = "Hello world!";`

**Patch source code:** `char string[] = "Hello world!";`

**Outcome results:** The `string[]` variable will become writeable.

**Outcome description:** Shiva will create its own `.data` section within the patch data segment at runtime. A character array called `string[] = "Hello World!"`; will be stored within the `.data` section, and all references to the old `string[]` variable within the executable will be re-linked by Shiva to reflect the location of the new writable `string[]` within the patch `.data` section.

### ***Initializing a .bss variable (Conversion from .bss variable to .data variable)***

In some instances the patch developer may need to initialize a `.bss` variable (uninitialized data).

**Original source code:** `int count;`

**Patch source code:** `int count = 0;`

**Outcome results:** The `count` variable is moved to the `.data` section and initialized to 0.

**Outcome description:** Shiva creates a `.text`, `.data`, `.bss`, sections and more within the patch objects process image. The new `count` variable will be stored within the `.data` section of the patch, and all references to the original `count` variable will be re-linked to the new one.

## **Symbol interposing on STB\_LOCAL symbols**

At this present time Shiva does not support patching symbols that are locally bound (i.e. `STB_LOCAL`) functions and objects. Functions declared as static, or initialized/uninitialized static variables will be supported in the near future though! Awaiting the next big version of Shiva.

## Symbol interposition on stripped ELF binaries

There are many occasions where legacy ELF binaries have been stripped. In particular we are referring to the `binutils strip` utility which removes the `.symtab` section from the ELF section header table. There are more drastic versions of stripping an ELF binary which include removing the entire section header table, and thus the map for locating the symbol, relocation, and other various meta-data within the ELF binary. Fortunately though Shiva uses `libelfmaster` under the hood which forensically reconstructs ELF section headers and symbol tables. Currently `libelfmaster` only reconstructs the symbol data for the functions that from `.symtab`, and `.dynsym` symbol tables. It is on *libelfmasters* roadmap to be able to forensically reconstruct `STT_OBJECT` symbols as well.

## Using the extern keyword in your patch code

It is common to reference an external variable in your patch code. An external variable in Shiva is any variable that exists within the executable or it's needed shared libraries. The `extern` keyword (in patch source code) will create the symbol type as `STT_NOTYPE` in the patch object, thus instructing Shiva to search outside of the patch module and from the ELF executable. If Shiva cannot find the symbol within the executable then it searches the shared libraries.

### Implicit declaration of extern doesn't work in Shiva

```
static int array_1[10]; // will create a new variable instead of using external one
```

This will cause Shiva to generate a symbol type of `STT_OBJECT`, and therefore Shiva will interpose the original `array_1` symbol with this new one. This C declaration behavior slightly deviates from the traditional C convention in which case a developer can declare a global variable from another source object without using the `extern` keyword. `"/bin/ld"` and `"/lib/shiva"` have different behavior here, since Shiva must be able to interpose symbols whereas `"/bin/ld"` doesn't support interposition of symbols unless they have weak bindings. The RTLD `"/lib/ld-linux.so"` on the other hand will interpose global symbols of any type. We need Shiva to generate an `STT_NOTYPE` symbol in order for it to link to the external symbol.

### Explicitly use external variables

```
extern static int array_1[10]; // will use external variable called array_1[10]
```

This code will create a symbol type of `STT_NOTYPE` for `array_1`, and therefore Shiva will link the patch code to the external variable.

## Implicitly use external variables

A patch developer may choose not to explicitly declare an external variable, and instead simply reference the external variable implicitly by name. For example, assuming *int array\_1[10]* is a global variable in the executable being patched then the variable can simply be referenced by name in the patch source.

```
int i;  
  
for (i = 0; i < 10; i++)  
    array_1[i] = i * 100;
```

This patch code would generate a symbol of type STT\_NOTYPE in the ELF patch with relocations for symbol *array\_1[]*. If the symbol exists in the executable or one of its shared library dependencies (shared objects) then Shiva will resolve the symbol and apply the related relocations

## Program Transformations: Advanced patching

Although symbol interposing is a great strength; A natural grammar for re-writing and re-linking code and data that are represented by ELF symbols. Unfortunately symbol interposing cannot readily be applied to the instruction sequences in between all of the symbol definitions. For example, adding several bytes of code to an existing function requires that the function be extended in length. For example if we want to insert a patch that adds 8 extra bytes into the middle of a function, this would cause any instructions that live after the insertion to have invalid xref offsets for any calls or references to code and data that live before the insertion. Therefore a complete transformation of the function being is required. This is where Transforms come into play. They give the patch writer the mechanics necessary to handle patching more nuanced parts of the program code. Shiva implements these directives as macros and they are interpreted by the Shiva Pre-linker which compiles them into *Transform Records*.

## Function splicing

The current version of Shiva v0.11 (Alpha) is using the first iteration of Transforms which introduced the concept of transforming an existing function, aka *Function Splicing*. Function splicing allows a patch developer to splice relocatable code (*patch code*) anywhere into a function, optionally overwriting existing code. Transforms are a meta-data type which describe the information necessary to re-write parts of an existing function. Shiva splices in relocatable code with the help of transforms. Transforms are

## High level technical details

Shiva does not transform the function in-place. The function is often times extended in size when transforming it, and therefore Shiva transforms the function at runtime in a newly memory mapped executable region. The Transformed function is constructed of the original function code with an insertion of new code that possibly extends the length of the function in bytes. The end of the function may contain *.text* encodings per the patch objects large-code-model relocations. There are many unique challenges to function splicing. One such challenge is that the executable program being patched has no relevant relocation meta-data for relocating it's functions. Function splicing can be broken down into roughly seven steps:

1. **Transform-slice-1:** Copy the first half of the original function code (*Up until the patch-insertion-offset*) into the new memory region.
2. **Transform-slice-2 :** Copy the new code being spliced in (*From the patch*) to the offset right after the last insertion, which we call the *patch-insertion-offset*.
3. **Transform-slice-3:** Copy the remaining code from the original function down into the third slice of our new memory region.
4. **Intra-function-relinking:** There may be local branch instructions within the code of the first and third slices of the transformed function: (*i.e. b, b.eq, b.ne, b.gt, etc.*) that branch to offsets that are potentially erroneous depending on whether the size of the patch code increased the distance between slice-1 and slice-2.
5. **Relinking function calls:** Call instructions within slice-1 and slice-3 must be re-linked since all of the call-sites are now moved into the new memory region of the transformed function, thus changing the offsets to other functions.
6. **Relinking global variables:** Global variable references within slice-1 and slice-3 must be re-linked. The ADRP/ADD instructions work together to form the address of the variable being accessed. In such cases Shiva must re-encode the adrp offset reflect the new page offset to the global variable's segment (*i.e. data segment for initialized globals*).
7. **Relocate spliced in code:** Eventually the `apply_relocation()` function is called in Shiva and the relocatable code that was spliced into the function is properly relocated. Please note that the patch objects relocation entries will no longer have valid *r\_offset* and *r\_addend* values until there are computed with the transform offsets and lengths, and therefore the process of relocation is heavily modified by transformations.



### Patch exercise 3: Patching a strcpy vulnerability

Change directories to “shiva\_examples/strcpy\_vuln”. This patch directory contains a simple vulnerable program that uses strcpy(). Our goal is to fix it to use strncpy(). Our approach is to overwrite the relevant code with a call to strncpy(). We will splice our code into the function, thus transforming the function.

Instead of revealing the source code to the binary this time, let’s approach this as someone who no longer has the source code. Function splicing requires some basic reversing knowledge. “shiva\_examples/strcpy\_vuln/overflow” is an AARCH64 PIE ELF executable. Let’s do a little bit of reversing and try to define the bug that it has. I commented the relevant lines of code in the GDB output below.

Image 3.24: Disassembling copy\_string(char \*string)

```
Starting program: /home/elfmaster/shiva_examples/strcpy_vuln/overflow AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 4, copy_string (string=0xfffff7e97748 <__libc_start_main+136> "\200\t") at overflow.c:6
6      {
(gdb) disas
Dump of assembler code for function copy_string:
=> 0x0000aaaaaaaa7b4 <+0>:      stp     x29, x30, [sp, #-48]!
   0x0000aaaaaaaa7b8 <+4>:      mov     x29, sp
   0x0000aaaaaaaa7bc <+8>:      str     x0, [x29, #24] // *(uint64_t)(bp + 24) = (char *)src;
   0x0000aaaaaaaa7c0 <+12>:     add     x0, x29, #0x20 // x0 = (char *)dst;
   0x0000aaaaaaaa7c4 <+16>:     ldr     x1, [x29, #24] // x1 = (char *)src;
   0x0000aaaaaaaa7c8 <+20>:     bl      0xaaaaaaaa690 <strcpy@plt>
   0x0000aaaaaaaa7cc <+24>:     nop
   0x0000aaaaaaaa7d0 <+28>:     ldp     x29, x30, [sp], #48
   0x0000aaaaaaaa7d4 <+32>:     ret
End of assembler dump.
(gdb) c
~
```

From the GDB disassembly in Image 3.23:

First the procedure prologue which allocates 48 bytes on the stack and stores the bp and ip there (x29 and x30).

1. copy\_string + 0: “stp x29, x30, [sp, #-48]!”  
Stores the pair: base pointer and instruction pointer at sp – 48

2. `copy_string + 4: "mov x29, sp"`  
The base pointer is given the value of the stack pointer.
3. `copy_string + 8: "str x0, [x29, #24]"`  
Stores the first and only function argument onto the stack at `bp + 24`. The pointer to the source string.
4. `copy_string + 12: "add x0, x29, #0x20"`  
Adds `0x20` to the base pointer and assigns the value to register `x0`, as the pointer to the dst buffer: i.e. `x0 = x29 + 0x20`; The stack allocated a total of 48 bytes. So calculate  $48 - 32 = 16$ , thus the destination buffer being passed to `strcpy` is 16 bytes.
5. `copy_string + 16: "ldr x1, [x29, #24]"`  
Retrieves the src string pointer that was stored at `bp + 24`.
6. `copy_string + 20: "bl strcpy@plt"`  
Calls the `strcpy` function's PLT entry.

From this GDB state we have been able to reverse the function `copy_string`. From our analysis it looks something like this:

```
void copy_string(char *src)
{
    char dst[16];

    strcpy(dst, src);
    return;
}
```

This function is obviously vulnerable since it performs no bounds checking before calling `strcpy`. Copying over 16 bytes will begin to corrupt the stack. Let's watch the entire program run before we discuss our patch.

### Image 3.25: Vulnerable program crashes via stack overflow

```
Continuing.

Breakpoint 5, copy_string (string=0xffffffff786 'A' <repeats 37 times>, "a") at overflow.c:9
9      strcpy(buf, string);
(gdb) x/i $pc
=> 0xaaaaaaaa7c0 <copy_string+12>:      add      x0, x29, #0x20
(gdb) p/x $x29 + 0x20
$33 = 0xffffffff3d0
(gdb) p/x &buf
$34 = 0xffffffff3d0
(gdb) c
Continuing.

Breakpoint 6, 0x000aaaaaaaa7c8 in copy_string (string=0xffffffff786 'A' <repeats 37 times>, "a") at overflow.c:9
9      strcpy(buf, string);
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0041414141414141 in ?? ()
(gdb) p/x $x29
$35 = 0x4141414141414141
(gdb)
```

We have a crashing program and have identified the vulnerable function: `void copy_string(char *src)`. The argument `src` is passed into the register `x0`, and the destination buffer is located at `bp + 32`. In order to fix this we will replace the code that calls `strcpy()` with `strncpy()`. This is a great opportunity to explore the *Function Splicing* capabilities of Shiva. Function splicing is a feature that allows a developer to splice C code into an existing function with rich symbol interaction due to Shiva's specially developed linking and transformation capabilities.

Let's take a look at the `copy_string()` function again with `objdump` and determine which address range to patch.

Image 3.26: `objdump` of function `copy_string`

```
000000000000007b4 <copy_string>:
7b4: a9bd7bfd      stp     x29, x30, [sp, #-48]!
7b8: 910003fd      mov     x29, sp
7bc: f9000fa0      str     x0, [x29, #24]
7c0: 910083a0      add     x0, x29, #0x20
7c4: f9400fa1      ldr     x1, [x29, #24]
7c8: 97ffffb2      bl      690 <strcpy@plt>
7cc: d503201f      nop
7d0: a8c37bfd      ldp     x29, x30, [sp], #48
7d4: d65f03c0      ret
```

The disassembly output above is highlighted over the code that must be patched. The code must be patched from `0x7bc` to `0x7cc`. `0x7cc` being the address directly after the last instruction being patched. If the code we are splicing in is larger than the area we are overwriting, then Shiva will extend the function by which all code from `0x7cc` and after will be pushed forward to account for the extra patch length. Based on everything we've learned about the software bug and the crashing program we can write a patch pretty to solve this pretty easily by replacing the call to `strcpy()` with `strncpy()`. In the GDB output and disassembly examples above we were able to determine everything that we need in order to write a function splicing patch.

1. Location of live variables in registers or on the stack
  - `char *src` is stored in variable `x0`
  - `char buf[16]` is stored at `x29 + 32`
2. Virtual address of where splice code begins: `0x7bc`
3. Virtual address of where splice code ends + 4: `0x7cc`

Now we can take that information and use the `SHIVA_T_SPLICE_FUNCTION` macro to solve our patch problem and fix the buffer overflow.

Image 3.27: Function splicing patch to replace strcpy with strncpy

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

#include "/opt/shiva/include/shiva_module.h"

#define MAX_BUF_LEN 16

SHIVA_T_SPLICE_FUNCTION(copy_string, 0x7bc, 0x7cc)
{
    /*
     * The function copy_string(char *string) will
     * store the string arg in x0. Create a pointer
     * that is paired with the x0 register. We will
     * name it src. (It is char *string)
     */
    SHIVA_T_PAIR_X0(src);

    /*
     * The destination buffer is at bp + 32 in the function
     * we are patching.
     * char *dst = LEA(bp + 32);
     */
    SHIVA_T_LEA_BP_32(dst);

    /*
     * Call strncpy instead of strcpy to fix the overflow.
     */
    strncpy(dst, src, MAX_BUF_LEN);
}
```

Let's break this patch down step by step.

#### 1. SHIVA\_T\_SPLICE\_FUNCTION(func\_name, start\_addr, end\_addr)

This macro instructs the compiler to create a global function named `__shiva_splice_fn_name_<func_name>`. In our case it creates a symbol called `__shiva_splice_fn_name_copy_string` in the `.text` section like any other function. This macro also creates two global variables to store the start and end address. The end address is also known as the extend or extension address, because the end address may extend forward depending on the size of the patch code. The end address should always be set to the last line of code patched + `ARM_INST_LEN` (Which is 4). Which is the address that will be pushed forward (And everything after it) thus the extend address. Let's take a look at the macro source code from "shiva/modules/include/shiva\_module.h".

Image 3.28: The SHIVA\_T\_SPLICE\_FUNCTION macro from shiva\_module.h

```
#define SHIVA_T_SPLICE_FUNCTION(fn_name, insert, extend) \
static uint64_t __shiva_splice_insert_##fn_name __attribute__((section(".shiva.transform"))) = insert; \
static uint64_t __shiva_splice_extend_##fn_name __attribute__((section(".shiva.transform"))) = extend; \
void * __shiva_splice_fn_name_##fn_name(void)
```

The macro in Image 3.28 is simple, it essentially creates body of code that defines two global variables that hold the **insert** and **extend** values and stores them in the .shiva.transform section for analysis by Shiva's Transformer code which handles all of the function splicing. The macro above creates the following code for our patch in Image 3.27.

```
static uint64_t
__shiva_splice_insert_##fn_name __attribute__((section(".shiva.transform")));

static uint64_t
__shiva_splice_extend_##fn_name __attribute__((section(".shiva.transform")));

void * __shiva_splice_fn_name_copy_string(void)
{
    <empty body>
}
```

## 2. SHIVA\_T\_PAIR\_X0(variable\_name)

This macro simply uses the *register* keyword to instruct the compiler that any reference to int64\_t <variable\_name> is a reference to the x0 register. We use this in our patch because the first argument of copy\_string() is stored in x0, char \*src. Let's take a look at the macros for pairing variables in your patch to a specific register.

Image 3.29: SHIVA\_T\_PAIR\_X0 Macro in shiva\_module.h

```
#define SHIVA_T_PAIR_X0(var) register int64_t var asm("x0");
#define SHIVA_T_PAIR_X1(var) register int64_t var asm("x1");
#define SHIVA_T_PAIR_X2(var) register int64_t var asm("x2");
#define SHIVA_T_PAIR_X3(var) register int64_t var asm("x3");
#define SHIVA_T_PAIR_X4(var) register int64_t var asm("x4");
#define SHIVA_T_PAIR_X5(var) register int64_t var asm("x5");
#define SHIVA_T_PAIR_X6(var) register int64_t var asm("x6");
#define SHIVA_T_PAIR_X7(var) register int64_t var asm("x7");
```

We can see from Image 3.29 that the **register** keyword is used to associate an `int64_t` variable with a register. This variable can be used as a pointer or any other type with casting in C.

### 3. SHIVA\_T\_LEA\_BP\_32(variable\_name)

This macro declares a variable

`int64_t <variable_name>`

And then defines it by assigning the effective address of `bp+32` (In AArch64 the base pointer is `x29`). Below is the macro from `shiva_module.h`

Image 3.30: SHIVA\_T\_LEA\_BP\_32 macro in shiva\_module.h

```
#define SHIVA_T_LEA_BP_32(var) asm volatile ("mov x9, x29\n" \
                                             "add x9, x9, #32"); \
register int64_t var asm("x9");
```

We can see that the `add` instruction adds 32 to the base pointer and stores the result in `x9`, which is referenced in C by register `int64_t var asm("x9")`. Let's take a look at our relocatable patch object `"shiva_examples/strcpy_vuln/overflow_patch.o"`.

**NOTE:** *Macros such as these are using the same register over and over, x9 for example. These macros are somewhat prototype and will be expanded on in the future so that Shiva intelligently picks which register to use automatically, or the developer can optionally select the register. As it stands many of these macros can't be used more than once because they will clobber the x9 register. See "modules/include/shiva\_module.h".*

Image 3.31: Symbol table of overflow\_patch.o

```
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$ eu-readelf -s overflow_patch.o

Symbol table [ 8] '.symtab' contains 14 entries:
12 local symbols  String table: [ 9] '.strtab'

Num:      Value              Size Type Bind Vis      Ndx Name
 0: 0000000000000000          0 NOTYPE LOCAL DEFAULT UNDEF
 1: 0000000000000000          0 FILE  LOCAL DEFAULT ABS overflow_patch.c
 2: 0000000000000000          0 SECTION LOCAL DEFAULT 1
 3: 0000000000000000          0 SECTION LOCAL DEFAULT 3
 4: 0000000000000000          0 SECTION LOCAL DEFAULT 4
 5: 0000000000000000          0 SECTION LOCAL DEFAULT 5
 6: 0000000000000000          0 NOTYPE LOCAL DEFAULT 5 $d
 7: 0000000000000000          8 OBJECT LOCAL DEFAULT 5 __shiva_splice_insert_copy_string
 8: 0000000000000008          8 OBJECT LOCAL DEFAULT 5 __shiva_splice_extend_copy_string
 9: 0000000000000000          0 NOTYPE LOCAL DEFAULT 1 $x
10: 0000000000000000          0 SECTION LOCAL DEFAULT 7
11: 0000000000000000          0 SECTION LOCAL DEFAULT 6
12: 0000000000000000        48 FUNC   GLOBAL DEFAULT 1 __shiva_splice_fn_name_copy_string
13: 0000000000000000          0 NOTYPE GLOBAL DEFAULT UNDEF strcpy

elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$
```

The symbol table has three symbols whose names are prefixed with “\_\_shiva\_splice”. There are two global variables containing the start and end address, and a function containing the code to splice into the function being transformed; in this case it is copy\_string() that is being transformed. Let’s take a look at the disassembly of our patch code...



Image 3.32: Disassembly of overflow\_patch.o

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

Disassembly of section .text:

0000000000000000 <__shiva_splice_fn_name_copy_string>:
  0:  f81f0ffe      str     x30, [sp, #-16]!
  4:  aa1d03e9      mov     x9, x29
  8:  91006129      add     x9, x9, #0x18
  c:  aa0903e1      mov     x1, x9
 10:  aa0103e3      mov     x3, x1
 14:  d2800202      mov     x2, #0x10
 18:  aa0003e1      mov     x1, x0
 1c:  aa0303e0      mov     x0, x3
 20:  94000000      bl      0 <strncpy>
 24:  d503201f      nop
 28:  f84107fe      ldr     x30, [sp], #16
 2c:  d65f03c0      ret

Disassembly of section .shiva.transform:

0000000000000000 <__shiva_splice_insert_copy_string>:
  0:  000007bc      .word   0x000007bc
  4:  00000000      .word   0x00000000

0000000000000008 <__shiva_splice_extend_copy_string>:
  8:  000007cc      .word   0x000007cc
  c:  00000000      .word   0x00000000
```

The disassembly above in Image 3.32 shows the code “\_\_shiva\_splice\_fsplce\_copy\_string()” that is going to be spliced into the function copy\_string() from the ELF binary “shiva\_examples/strcpy\_vuln/overflow”. Please note that the procedure prologue should not be there, but aarch64-gcc doesn’t respect the naked attribute. The procedure prologue at offset 0x0 and the epilogue at offset 0x28-0x2c will be NOP’d out by Shiva before the patch is spliced in. Notice that the start and end address for patching are stored in their respective global variables in the .shiva.transform section.

### ***Demonstrate strcpy\_vuln.o patch***



The ELF binary `./overflow` crashes if more than 16 bytes are copied into stack variable `buf[16]` in `copy_string()` function. Our patch replaces `strcpy()` with `strncpy()`, let's check it out.

Image 3.33: Applying `overflow_patch.o` to fix the vulnerability

```
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$ ./overflow AAAA
The string: AAAA, was copied into buf!
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$ ./overflow AAAAAAAAAAAAAAAAAAAAAAAAAA
The string: AAAAAAAAAAAAAAAAAAAAAAAAAA, was copied into buf!
Segmentation fault (core dumped)
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$ shiva-ld -e overflow -p overflow_patch.o \
> -s /opt/shiva/modules -i /lib/shiva -o overflow
[+] Input executable: overflow
[+] Input search path for patch: /opt/shiva/modules
[+] Basename of patch: overflow_patch.o
[+] Output executable: overflow
Writing out original interp path: /lib/ld-linux-aarch64.so.1
Finished.
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$ sudo cp overflow_patch.o /opt/shiva/modules/
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$ ./overflow AAAAAAAAAAAAAAAAAAAAAAAAAA
The string: AAAAAAAAAAAAAAAAAAAAAAAAAA, was copied into buf!
elfmaster@esoteric-aarch64:~/shiva_examples/strcpy_vuln$
```

As shown in Image 3.33, we demonstrate how the program crashes before we patch it, and then go on to `shiva-prelink` the binary using `shiva-ld`. The next time we run `./overflow` with a string larger than 16 bytes we no longer get a crash. At least 6 other examples for function splicing in “`shiva/modules/aarch64_patches/fsplice/`” starting from simple to more complex examples.

## ***Current limitations and the future of Transforms***

Shiva is still in an early development and the transforms need work in a number of areas:

1. Function splicing only works when patching an area that is being overwritten. In other words, it doesn't yet handle the scenario of inserting patch code in between two contiguous addresses. i.e.

0x400000 <inst1>

← patch code inserted here, and extending 0x400004 forward

0x400004 <inst2>

If you were to select a `insert_address` `0x400000` and an end address of `0x400004`, Shiva would overwrite the first instruction and then push everything else after that instruction forward by `patch_len` bytes. However in such cases we would desire for Shiva to not overwrite the first instruction. This is high priority to fix by the next coming version of Shiva (Shiva v1.0 beta).

2. Shiva only allows one transformation per-function, you cannot splice code into multiple locations of a single function. Although you can apply one transform to any function.

<https://github.com/advanced-microcode-patching/shiva/issues/14>

3. A CFG is built at runtime by Shiva so that it can properly re-link the executable which doesn't hold the type of relocation data we need. This is slow, but necessary for transforms to work. In the coming version of Shiva, this phase of runtime CFG analysis will be available when running Shiva directly to load your patch, but the Shiva-Prelinker "shiva-ld" will parse and generate the CFG graph and generate the necessary relocations, all of which will be stored within the ELF object files `.shiva.cfg` section. <https://github.com/advanced-microcode-patching/shiva/issues/17>

4. Transformed functions do not currently handle recursion. If the patch writer splices in code that attempts to call the function that it's splicing into, the result will be that Shiva links the call to the original version of the function, which is incorrect. This will be fixed in the next Shiva release as well. <https://github.com/advanced-microcode-patching/shiva/issues/16>

## 4 In closing

Shiva is in early development and any feedback is welcomed. Shiva is still evolving, and many new features and support will be implemented during the coming year (And beyond).

Author:

[elfmaster@arcana-research.io](mailto:elfmaster@arcana-research.io)