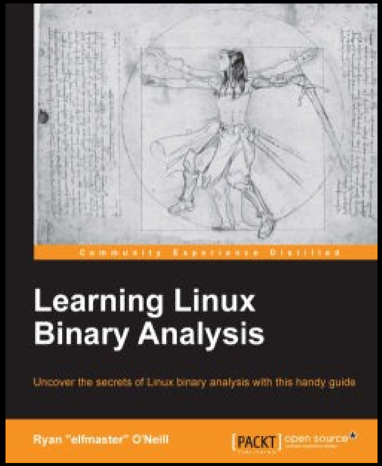
Custom ELF Dynamic linkers & Loaders For fun, profit, and Security solutions

Ryan “ElfMaster” O’Neill

Introduce myself…

Ryan “ElfMaster” O’Neill
Author: Phrack, POC||GTFO, Vxheaven, tmp.0ut, bitlackeys.org



Today’s topics

- A primer on ELF linkers and loaders
- Custom ELF dynamic linking with Shiva: https://github.com/advanced-microcode-patching/shiva
- Custom dynamic linkers for Virus infection
- Using Shiva to build security modules
  - *Granular ASLR* implementation demo
- State of the art binary patching solutions with Shiva
  - Game cheats with Pacman demo
  - NASA patch from the DARPA AMP program

Defining ELF loaders

- Software which loads ELF objects into a memory image
- Generic userland loaders (i.e. ulexec)
  - Known as “Userland Exec” technique
  - https://grugq.github.io/docs/ul_exec.txt
  - https://ulexec.github.io/post/2021-04-08-shelfloading/
- Kernel loaders
  - src/linux/fs/binfmt_elf.c: See load_elf_binary()
- The dynamic linker ld-linux.so is a loader
  - Loads shared libraries for dynamic linking
  - Optionally can load and execute ELF executable files directly

Loaders map code and data into memory segments

- ELF loaders “Generally” map the PT_LOAD segments into memory
  - ELF types: ET_EXEC and ET_DYN
  - PT_LOAD segments describe contiguous memory regions in an ELF file
- Userland ELF loaders (i.e. ulexec)
  - Binary protectors/packers have userland ELF loaders
  - Various types of anti-forensics techniques and malware loaders
  - ld-linux.so can load executables and load shared objects
  - Shiva can load executables directly and load relocatable objects

Ld-linux.so as an interpreter

```
elfmaster@arcana-laptop:~$ readelf -l /bin/ls

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x6aa0
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000003458 0x0000000000003458  R      0x1000
  LOAD           0x0000000000004000 0x0000000000004000 0x0000000000004000
                 0x0000000000013091 0x0000000000013091  R E    0x1000
  LOAD           0x0000000000018000 0x0000000000018000 0x0000000000018000
                 0x0000000000007458 0x0000000000007458  R      0x1000
  LOAD           0x000000000001ffd0 0x0000000000020fd0 0x0000000000020fd0
                 0x00000000000012a8 0x0000000000002570  RW     0x1000
```

Ld-linux.so loading example

```
elfmaster@arcana-laptop:~/dartmouth_talk$ /bin/ls /home
elfmaster  ryan
elfmaster@arcana-laptop:~/dartmouth_talk$ /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 /bin/ls /home
elfmaster  ryan
elfmaster@arcana-laptop:~/dartmouth_talk$ 
```

- The kernel loader: load_elf_binary() loads the program “/bin/ls” in the first command and ld-linux.so (As the interpreter) loads the shared libraries internally
- The dynamic linker “ld-linux.so” can be executed directly and used as the primary executable loader.
- In the second command ld-linux.so has a built-in userland implementation of execve() that maps the executable into memory

Defining Linkers

- Linkers are essentially simple binary patchers that stitch code together
- Linkers use relocation meta-data to determine which code or data to patch
  - Relocation meta-data tells the linker how to resolve symbolic references into symbolic definitions
- There are two primary ELF linkers that we are mostly concerned about
  - The linker “/bin/ld”: links ELF relocatable objects into an executable format ready for program execution
  - The dynamic linker “/lib/ld-linux.so”: links ELF shared objects (ET_DYN) to executable via PLT/GOT and GOT relocations

Linker example: R_X86_64_GOTPC64

This code satisfies the R_X6_64_GOTPC64 relocation type…
GOT_ADDRESS – POSITION (RIP) + ADDEND
Results in offset to base of GOT (Global Offset Table)

```
switch(rel_entry->rel.type) {
case R_X86_64_GOTPC64: /* GOT - P + A */
        if (elf_section_by_name(&ctx->elfobj, ".got", &got) == false) {
                fprintf(stderr, "elf_section_by_name() failed on .got\n");
                return false;
        }
        rel_val = ELF_RUNTIME_BASE(got.address) - rel_addr + rel_entry->rel.addend;
        *(uint64_t *)r_ptr = rel_val;
        break;
```

It patches an instruction like this
16:    49 bb 00 **00 00 00 00**   movabs $0x0,%r11

Evil dynamic linker ELF Virus

- “Preloading the linker for fun and profit” tmp0ut #2
  - https://tmpout.sh/2/6.html
- An evil dynamic linker which loads a modular Virus in the form of an ELF relocatable object
  - The evil dynamic linker loads an ELF relocatable object that infects other binaries PT_INTERP with the evil linker
- This research ultimately formed into the project now known as “Shiva”

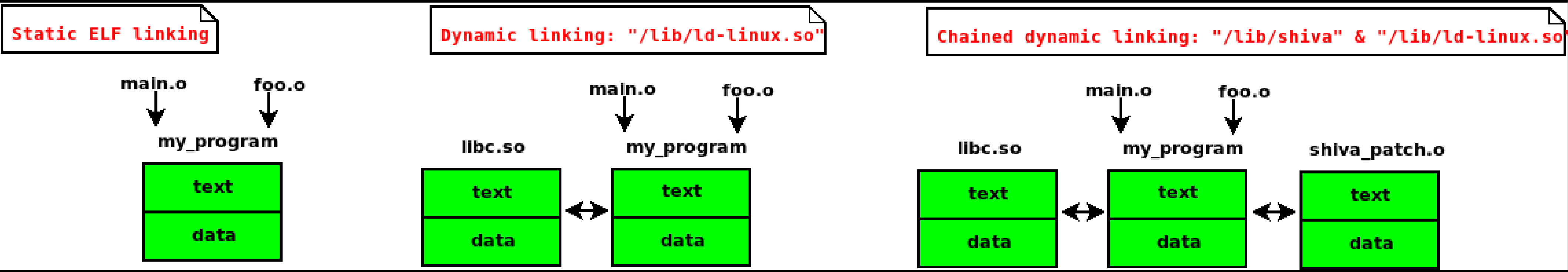
The power of ELF interpreters

- Fast In-process instrumentation
- Faster than PTRACE hot-patching
- Can transform, link, and instrument the program
- ELF relocatable objects (ET_REL) contain granular relocation meta-data

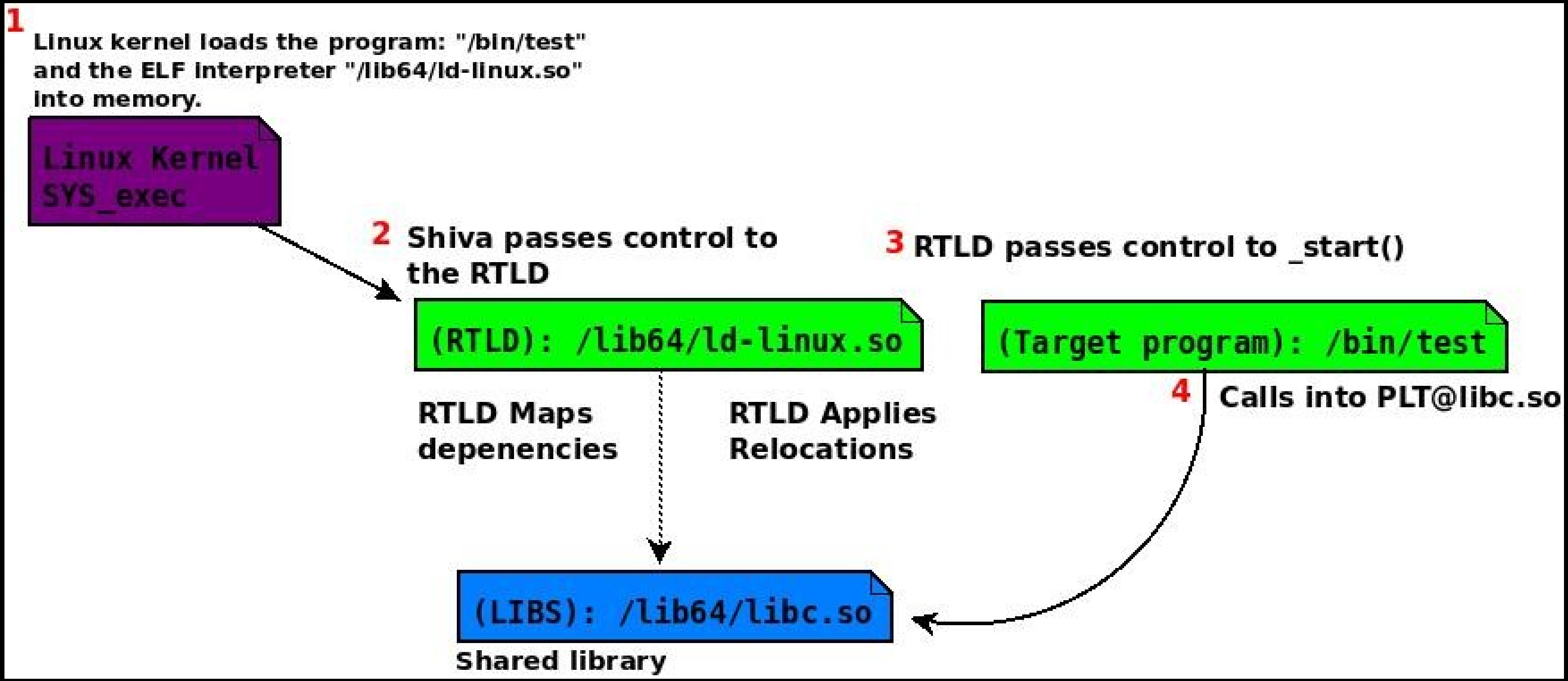
What is Shiva?

- Custom ELF interpreter for Linux AArch64 and X86_64
- Hybridized linking technology
- A dynamic binary-rewriting engine
- Works seamless with the ELF ABI Toolchain
- Modular patching environment (REL objects)
- DARPA AMP program phase 2 and 3.

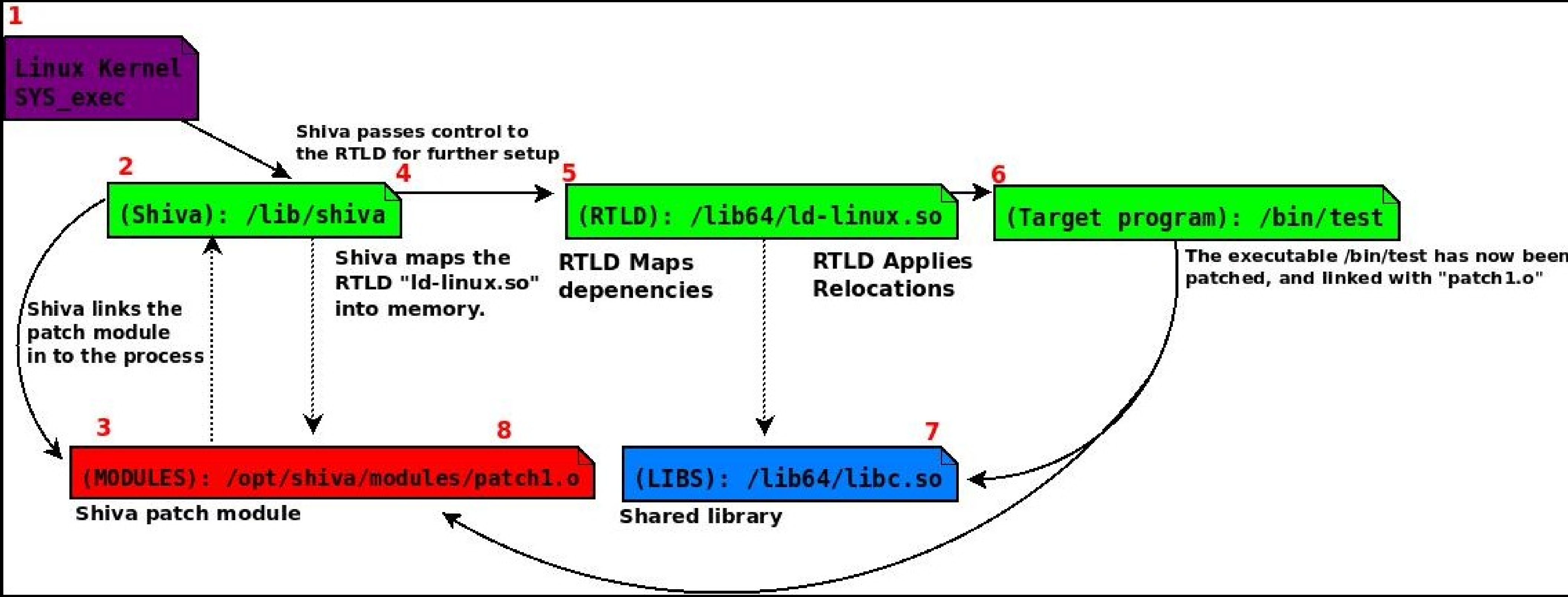
Shiva's ELF linking workflow



Standard ELF dynamic linking workflow



ELF "Linker chaining" diagram



Shiva is a custom dynamic linker

- Shiva has two modes: Module mode & Patch mode.
- Shiva can load “Modules” into the process address space
  - In-process debugging technologies
  - Security hardening modules (See gASLR demo)
  - Fuzzing harnesses
  - Dynamic analysis, tracing engines, reversing tools

- Shiva offers state of the art ELF binary patching with evolving DWARF support
  - NASA patch challenges demo
  - Pacman game-cheats

Shiva uses libelfmaster under the hood

- Libelfmaster is an intelligent ELF parsing library
  - Forensically reconstructs section headers internally
  - Forensically reconstructs symbol tables internally
  - Advanced API capabilities for ELF parsing and modification
  - API has never been properly documented (But easy to use)
  - Supports x86_64 and AArch64
- https://github.com/elfmaster/libelfmaster

- Shiva is symbolically driven but is still able to patch stripped binaries in x86_64 due to libelfmaster symbol table reconstruction

Shiva binary patching

- Shiva offers flexible binary patching capabilities
- Performs load-time program transformation
- Symbol interposition
  - Rewrite global code and data on the fly by symbol name.
  - Think LD_PRELOAD, except with the ability to re-write functions and global data within the main executable
- Function splicing
  - Advanced program transformation based on extended ELF ABI
  - Splice code into an existing function with Shiva’s *Transformation Capabilities*
  - New DWARF support for patch-by-line-number and live variable resolution
- Pre-linking tool “shiva-ld” with dependency injection capabilities via DT_NEEDED

Function splice patches can be hard to apply

- Function splicing patches are powerful but can be complex to apply
  - Require low-level RE knowledge by the patch developer
  - Require knowledge of registers and stack for live variable access
  - Require knowledge of which memory address ranges to patch

Transformations begin where relocations leave off

- ELF Relocations
  - ELF meta-data that describe simple patching operations
  - Re-encoding instructions and symbolically resolving definitions between code and data
  - No larger than 4 to 8 byte patches
- ELF Transformations
  - ELF meta-data that describe complex patching operations
  - Function Transformation: Granular function re-writing
  - Splice code is fully relocatable with rich symbolic access
  - Designed on top of ELF relocations

Function splicing

- Splice code into existing function
  - Re-write any portion of a function
  - Relocatable code is spliced into arbitrary locations with the help of *ELF Transformations (Short name: Transforms)*
  - Rich symbolic access to symbols process wide
  - No limit to the amount of code being spliced in, Shiva will extend the size of the function
- Transform macros
  - Help the developer accomplish granular patching controls
  - Generate custom ELF meta-data called Transform records.
  - Designed on top of ELF relocations as a second layer of abstraction to program transformation

Fixing a strcpy vulnerability

```
#define MAX_BUF_LEN 16

void copy_string(char *src)
{
        char buf[MAX_BUF_LEN];

     - strcpy(buf, src);
 }
```

```
#define MAX_BUF_LEN 16

void copy_string(char *src)
{
        char buf[MAX_BUF_LEN];

     + strncpy(buf, src, MAX_BUF_LEN);
     + buf[MAX_BUF_LEN - 1] = '\0';
 }
```

```
00000000000007b4 <copy_string>:
 7b4:   stp     x29, x30, [sp, #-48]!
 7b8:   mov     x29, sp
-7bc:   str     x0, [x29, #24]
-7c0:   add     x0, x29, #0x20
-7c4:   ldr     x1, [x29, #24]
-7c8:   bl      690 <strcpy@plt>
 7cc:   nop
 7d0:   ldp     x29, x30, [sp], #48
 7d4:   ret
```

```
00000000000007b4 <copy_string>:
 7b4:   stp     x29, x30, [sp, #-48]!
 7b8:   mov     x29, sp
+7bc:   mov      x9, x29
+7c0:   add     x9, x9, #0x20
+7c4:   mov     x1,x9
+7c8:   mov     x3, x1
+7cc:   mov     x2, #0xf
+7d0:  mov      x1, x0
+7d4:  mov      x0, x3
+7d8:  bl       8d4 <strncpy@patch.plt>
 7dc:   nop
 7e0:   ldp     x29, x30, [sp], #48
 7e4:   ret
```

Example of traditional splice without DWARF support: stb-resize-server

```c
#include <stdint.h>
#include <stdio.h>
#include "shiva_module.h"

#define RING_BUFFER_NUM_ENTRIES 0x14c // offset of ring_buffer_num_entries member in struct sbir__info
#define ALLOC_RING_BUFFER_NUM_ENTRIES 0x1d0 // offset of alloc_ring_buffer_num_entries member in struct sbir__info

SHIVA_T_SPLICE_FUNCTION(stbir__alloc_internal_mem_and_build_samplers, 0x81e2f, 0x81e35)
{
        register char *info asm ("rbx");
        int ring_buffer_num_entries = *(int *)(info + RING_BUFFER_NUM_ENTRIES);
        int alloc_ring_buffer_num_entries = *(int *)(info + ALLOC_RING_BUFFER_NUM_ENTRIES);

        if (alloc_ring_buffer_num_entries < ring_buffer_num_entries) {
                asm("mov $0, %rax\n"
                    "add $0x118, %rsp\n"
                    "pop %rbx\n"
                    "pop %r12\n"
                    "pop %r13\n"
                    "pop %r14\n"
                    "pop %r15\n"
                    "pop %rbp\n"
                    "ret");
        }
}
```

Recent DWARF enhancements to replace code by source line number

- Patch developers can replace a source line by line number
  - Specify source line number
  - Specify the function name
  - Replace a single source line with an arbitrary amount of new lines of code

NASA ground-control challenge (DARPA AMP program)

- Ground control software fails to properly parse science data
  - processSciencePacket(char *buf, int len)
  - Fails to handle streams that include more than one header
  - [header][pdu][pdu][pdu][header][pdu][pdu][pdu]
  - Header length is 6 bytes and PDU’s are 6 bytes

```c
#define HEADER_LEN 6
#define PDU_LEN 6

SHIVA_T_SPLICE_FUNCTION_REPLACE_SRCLINE(processSciencePacket, 11)
{
        static int ret, i, len;
        static char *p, *sp, *newp, new[1024];
        static size_t new_len, header_len;
        static size_t delta_len = 0, dst_buf_len = 0, iter = 0, total_header_len = 0;
        SHIVA_T_PAIR_RSI(lenarg);
        SHIVA_T_PAIR_RDI(realbuf);
        char *buf = (char *)realbuf;
        len = lenarg;

        for (newp = new, p = buf; (p - buf) < len; ) {
                uint16_t packet_len = *(uint16_t *)&p[4];
                packet_len =  __builtin_bswap16(packet_len);
                packet_len += 1;
                new_len = packet_len + HEADER_LEN;
                header_len = (iter++ == 0) ? 0 : HEADER_LEN;
                memcpy(newp, p + header_len, new_len - header_len);
                newp += new_len - header_len;
                p += new_len;
                dst_buf_len += new_len - header_len;
                total_header_len += header_len;
        }
        if (dst_buf_len >= len)
                return -1;
        ret = SHIVA_HELPER_CALL_EXTERNAL_ARGS2(processSciencePacket,
            new, len - total_header_len);
        exit(0);
}
```

Function splice to replace line 11 in the function ProcesSciencePacket() with code that re-packs the network stream buffer with only one header before the PDU stream

Shiva demo: Fix NASA ground control processSciencePacket() function

- Run ./science_dp_integrated before patch
- See how all headers after the first one are misconstrued as science data
- Apply function splice patch and see how it is fixed

NASA patch2: Symbol interposition approach to fix processSciencePacket()

```
int processSciencePacket(char *buf, int len)
{
        int ret, new_len;
        char *p;

        /*    6      6    6    6     6      6    6
         * [header][pdu][pdu][pdu][header][pdu][pdu]
         *///\------------------\  \---------------- etc. \

        /*
         * Process initial header
         */
        for (p = buf; (p - buf) < len; ) {
                uint16_t packet_len = *(uint16_t *)&p[4];
                packet_len =  __builtin_bswap16(packet_len);
                packet_len += 1;
                new_len = packet_len + HEADER_LEN;
                ret = SHIVA_HELPER_CALL_EXTERNAL_ARGS2(processSciencePacket,
                    p, new_len);
                p += new_len;
        }
        return ret;
}
```

Fun with Gamecheats & Pacman

Shiva patch to give yourself 31337 lives in Pacman

```
elfmaster@arcana-laptop:~/amp/shiva/modules/x86_64_patches/pacman$ cat pacman_patch2.c

/*
 * Pacman patch to give 31337 lives! Simply sets
 * the global variable: int lives, to 31337.
 */

extern int lives = 31337;
elfmaster@arcana-laptop:~/amp/shiva/modules/x86_64_patches/pacman$
```

Fun with Gamecheats & Pacman

Shiva patch to keep enemies in “Frightened” state

```
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include "/opt/shiva/include/shiva_module.h"

extern uint8_t fgState;
extern bool frighten;
extern int frightenTick;

void my_idle(void)
{
        frighten = true;
        frightenTick = 0;
        _Z4idlev();
}

void glutIdleFunc(void)
{
        uint8_t *fptr = (uint8_t *)&fgState + 104;

        *(uint64_t *)fptr = &my_idle;
}
```

Shiva’s Module Mode

- Shiva modules are mapped into the process image early on (before ld-linux.so)

- Module entry point: shiva_init(shiva_ctx_t *ctx)
  - AT_ENTRY is hooked to point to shiva_init
  - [kernel]→[shiva]→[ld-linux.so]→[module]→[program]

From a *module mode* perspective what is Shiva?

- A weird machine: programmable runtime engine
- Custom dynamic linker:
  - Loads ELF micro-programs into the address space
- Shiva provides a rich API for program tranfsormation, tracing, hooking, debugging
- Think “LKM’s (loadable kernel modules) for userland processes”
- A programmable debugging engine that does NOT require the ***ptrace syscall***

What can Shiva modules be used for?

- Building debuggers and tracers
- Designing process security modules
  - Anti-exploitation, sandboxing, process hardening
- In-memory fuzzing harnesses
- Software profiling
- Virus detection engines
- Malware unpackers
- Hot-patching code and data
- Userland-rootkit detection & disinfection
- Extremely fast instrumentation, hooking, and process injection of all types (Without the use of SYS_ptrace)

From a blackhat perspective

- Shiva can be used to crack software
- Designing powerful in-process rootkits, backdoors
- Perform modular virus infection: “Preloading the linker for fun and profit”: https://tmpout.sh/2/6.html
- Create ELF binary protection engines

Understanding Shiva modules

- Shiva modules are loaded from: “/opt/shiva/modules”
- Modules can resolve symbols to:
  - All loaded shared libraries (libc.so, etc.)
  - libcapstone
  - libelfmaster
  - ShivaTrace API

- Similar to LKM (Loadable kernel module):
  - Shiva modules are ELF relocatable objects
  - gcc -mcmodel=large -c my_module.c
  - Shiva modules must have an init routine
    - int shiva_init(shiva_ctx_t *ctx)

Shiva Module: shiva_ctx struct

- The context struct “shiva_ctx_t”
- Passed into the modules shiva_init() function
- Contains:
  - Linking data
  - Register state
  - Process state
  - Thread state
  - Control flow data
  - Signal data
  - Shiva-Trace API specific data structures
  - Full access to elfobj_t of target executable

Shiva modules execute in-process

- Shiva modules are executed in-process
- **In-process** means:
  - The shiva runtime engine is in the same process as the target program
  - The module also executes within the same process address space
- Instrumentation and hooking is extremely fast
- Module has an ***initialization function***
- The module init function sets hooks and breakpoints within the target program
- Handler functions are callbacks for hooks and breakpoints

Harden system() against cmd injection attacks

```
#include "../shiva.h"

int n_system(const char *s)
{

        printf("s: %s\n", s);
        if (strstr(s, ";") != NULL || strstr(s, "|") != NULL) {
                printf("Detected possible OS command injection attack '%s'\n", s);
                abort();
        }
        return system(s);
}

int
shakti_main(shiva_ctx_t *ctx)
{
        bool res;
        shiva_error_t error;

        res = shiva_trace(ctx, 0, SHIVA_TRACE_OP_ATTACH,
            NULL, NULL, 0, &error);
        if (res == false) {
                printf("shiva_trace failed: %s\n", shiva_error_msg(&error));
                return -1;
        }
        res = shiva_trace_register_handler(ctx, (void *)&n_system,
            SHIVA_TRACE_BP_PLTGOT, &error);
        if (res == false) {
                printf("shiva_register_handler failed: %s\n",
                    shiva_error_msg(&error));
                return -1;
        }
        res = shiva_trace_set_breakpoint(ctx, (void *)&n_system,
            0, "system", &error);
        if (res == false) {
                printf("shiva_trace_set_breakpoint failed: %s\n", shiva_error_msg(&error));
                return -1;
        }
        return 0;
}
```

In Shiva's patch mode we could do this:

```c
int system(const char *s)
{
	if (strstr(s, ":") != NULL || strstr(s, "|") != NULL) {
		printf("Detected possible OS command injection attack '%s'\n", s);
		abort();
	}
	return SHIVA_HELPER_CALL_EXTERNAL_ARGS1(system, s);
}
```

We can simply use natural C to write a new version of the function
And the Shiva linker will interpose the original system@PLT
With our new one at runtime.

I have built several module prototypes

- Backwards edge CFI on PLT entries (mitigates ret2PLT attacks)
- A Function tracer that is thousands of times faster than ltrace
- A fuzzing harness that fuzzes ld-linux.so
- And most recently gASLR (Granular address space layout randomization)

Granular ASLR

- Standard ALSR only randomizes the base address
- Granular ASLR randomizes the location of every function, PLT entry, and global data var
- Current module only randomizes the functions though

About my gASLR implementation

- Program requirements
  - Must be PIE (i.e. ET_DYN type)
  - Must be compiled with a large code model
  - Must be compiled with –emit-relocs flag (Which preserves the .rela.text section)

- The Shiva module for gASLR consumes the .rela.text section for function relocation
- The gASLR module moves every function to a new address via mmap()
- Future versions will give each function a random offset from its new base

gcc -mcmodel=large test.c -o test

```
elfmaster@arcana-laptop:~/amp/shiva/modules/x86_64_modules/aslr$ cat test.c
#include <stdio.h>
#include <stdlib.h>

int test1(void)
{
        int i = 0;
        printf("Hello\n");
        return 0;
}

static int ignore_me(void)
{
        int i = 7;
        return 3;
}

int main(void)
{
        char *p = malloc(10);
        if (p == NULL) {
                perror("malloc");
                exit(0);
        }
        printf("base address: %p\n", (unsigned long)&ignore_me & ~4095);
        printf("main() is at %p\n", &main);
        printf("test1() is at %p\n", &test1);
        test1();
}
```

How to install gASLR on a program

```
$ shiva-ld -e ./test -p gASLR.o -i /lib/shiva -s /opt/shiva/modules -o test.patched
$ sudo cp gASLR.o /opt/shiva/modules
```

OpenSource MIT license

https://github.com/advanced-microcode-patching/shiva

$ git clone git@github.com:advanced-microcode-patching/shiva
$ cd shiva; git checkout x86_64_port