Malware Engineering Part 0×2— Finding shelter for parasite

Abhinav Thakur Follow
Nov 14 · 13 min read

```
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffde50 --> 0x555555554670 (<__libc_csu_init>:     push   r15)
0008| 0x7fffffffde58 --> 0x7ffff7a05b97 (<__libc_start_main+231>:       mov    edi,eax)
0016| 0x7fffffffde60 --> 0x1
0024| 0x7fffffffde68 --> 0x7fffffffdf38 --> 0x7fffffffe293 ("/home/critical/hell/xxx")
0032| 0x7fffffffde70 --> 0x100008000
0040| 0x7fffffffde78 --> 0x55555555464a (<main>:        push   rbp)
0048| 0x7fffffffde80 --> 0x0
0056| 0x7fffffffde88 --> 0xf60622e9a1511b53
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x000055555555465a in main ()
gdb-peda$ x/5s 0x5555555546f4
0x5555555546f4: " Compilepeace Was Here x_x "
0x555555554710: "\001\033\003;<"
0x555555554716: ""
```

> A virus is a small infectious agent that replicates only inside the living cells of an organism ×_×

That's how Wikipedia defines a biological virus in a nutshell. Holding on the analogy, a computer virus comes under the category of a malware that infects host binaries (even memory) via some parasitic code injection technique. Parasite here is the code that gets injected, residing in the host binary to takeover the hijacked code flow of the host program. After infection, the host binary is trojanized to achieve further goals. Trojan is a software that has malicious intent but disguise to be a legitimate program. The basic idea of a virus is to hijack the code flow of the program and hand it over to the parasite code, which (after execution of its malicious code) silently transfers control to the host binary resuming the intended code execution. In this article we'll be discussing an approach towards virus design and the algorithm used for infection process.

NOTE : This article is the continuation of — *That magical ELF* malware engineering series where we kept our first foot into the world of ELF binaries.

Prerequisites

- Understanding of ELF file format is mandatory.
- A decent amount of C programming skills along with system programming concepts under Linux will greatly help in understanding the article to the fullest.
- Further, we'll be using data structures defined in elf.h (present in `/usr/include/` directory of your local Linux filesystem) to programmatically access and manipulate parts of the host binary. Therefore, I encourage the reader to go through the structures . Alternatively, these data structures can be referenced from *page 5* of the Linux manual — `$ man 5 elf`.

In the world of snipers and hand grenades, what do we intend to make?

A dagger, perhaps. What we're going to design is a silent weapon (infector program) with some poison(parasite) which could be used after the exploitation phase of the cyber kill-chain to stealthily maintain persistence over the victim machine via some commonly executed trojans.

In memory, a program is laid out in the form of segments (each segment constituting one or more sections). Let's look at Program Header Table (PHT) of the `/bin/ls` utility (which lists the content of the current directory) -

Above, we see that there are 2 loadable segments (marked as type LOAD) with segment permissions R-E (Read-Execute) and RW- (Read-Write). Looking at the section to segment mappings, it's safe to infer that they are the **CODE/TEXT** and **DATA** segment respectively.

Abstract memory view of a program

After an innocent program is loaded into memory, segments are page-aligned but the sections (comprising segments) rarely align with the page boundary, thereby leaving space in between current and the adjacent segment. This space is padded by the zero bytes when the binary is loaded

into memory. In the above picture, the padding is represented by a series of `P` which causes alignment in segments. We'll use this padding to provide residence to the parasite as it seems to be a nice comfy shelter. There are however cases where the padding area is smaller than the parasite size in which case this method of infection (a.k.a ***segment padding infection*** in UNIX world and ***code caving*** in windows world) would not be able to infect that binary. Bellow is the algorithm for infection mechanism.

```
-x-x- Load parasite from file on-disk into memory
1. Get parasite_size and parasite_code address (location in allocated
memory)

-x-x- Find the padding_size (unused space) between CODE segment and
the NEXT segment after CODE segment(usually data segment)
2. In the CODE segment PHT entry, increase the following-
-> p_filesz    (by parasite size)
-> p_memsz     (by parasite size)


Get and Set respectively,
-> padding_size  = (offset of next segment (after CODE segment)) -
(end of CODE segment)
-> parasite_offset = (end of CODE segment) or (end of last section of
CODE segment)
-> parasite_load_address = virtual address (vaddr) + sizeof CODE
segment (filesz)
```

```
-x-x- PATCH Host entry point
3. Save the original_entry_point of host binary.
4. Alter the host entry to point to the location
parasite_offset/parasite_load_address (i.e. the location where
parasite is to be injected into the host binary)

-x-x- PATCH SHT
5. Find the last section in CODE Segment and increase -
-> sh_size    (by parasite size)

-x-x- PATCH Parasite offset
6. Find and replace Parasite jmp-on-exit address/offset placeholder
with original_entry_point 0x???????????????

-x-x- Inject Parasite to Host (mapped @ host_mapping)
7. Inject parasite code @ (host_mapping + parasite_offset), i.e. to
the end of the last section (among all other sections in CODE
segment)
8. Write the infection to disk ×_×
```

NOTE : There's a drawback to this infection point. Since the CODE segment is **not** having write permissions, a self-modifying parasite code won't work.

Let the CODE speak for itself ×_×

With all the knowledge equipped about ELF's and our black hoodies on, its time to dive into actual implementation of infection algorithm in C

programming language. I'll walk through the source code of Kaal Bhairav's infection module.

Prologue

First, let's declare some variables. I hope the comments above would suffice as a lucid understand of what a variable stores.

Ingredients for spell

`ElfParser()` accepts a parameter named *filepath* (path to host binary) and follows each step of infection algorithm by coordinating with other utility functions.

And that's how an evil ELF was born

- (Line **84**) Abstracts the use of `mmap()` to map the host binary into memory for further modification. First byte of the host is mapped @ location specified by *host_mapping*.
- (Line **86–90**) It gets Elf header of the host binary into *host_header* (`struct Elf64_Ehdr` type specified in elf.h). It parses **ET_EXEC** and

ET_DYN (executable and shared objects) type of class 64-bit binaries skipping **ET_REL** (relocatable), **ET_CORE** (core) or binaries of class **ELFCLASS32** (32 bit binaries). Here **ET_EXEC**, **ET_DYN**, **ET_REL**, **ET_CORE** are macros defined in elf.h.

- (Line **93–94**) Calls `LoadParasite()` to get parasite into memory preparing it for injection into the host. Crafting parasite code (which is different for both the LSB executable and shared object type) will be explained in the next article.
- (Line **96–101**) Calls `GetPaddingSize()` which gets the padding size (shelter size) and checks if the host can accommodate the parasite into the padding.
- (Line **103–107**) Saves the *original_entry_point* of the host binary and modifies the host entry point by location where the parasite is to be injected (*parasite_offset* and *parasite_load_address* set up by `GetPaddingSize()`). Parasite location will be different for both executable and shared object binaries. In an ET_EXEC host, entry point is specified by an address whereas in shared objects(ET_DYN), entry point is specified by an offset since it contains PIC (**P**osition **I**ndependent **C**ode).

LSB executable and LSB shared object

- (Line **109–115**) `PatchSHT()` is our utility function that performs modification on **S**ection **H**eader **T**able to accommodate the parasite. Then we use `FindAndReplace()` to patch the jmp-on-exit address/offset of parasite code with *original_entry_point* of host binary, after which we inject the parasite into host via `memcpy()` and use `munmap()` to unmap the binary and write infected host mapping onto the disk.

mmapFile() — Keeping the host at a one-arm distance

mapping the host

The function `mmapFile()` accepts path to host as parameter, maps the host into current process address space and returns a pointer to first byte of the mapped host binary.

- (Line **317–321**) Opens the host binary in Read-Write mode
- (Line **323–328**) `lstat()` to get the information about the host, specifically size of host binary (on-disk size) to be used by mmap(). Read `$ man stat`.

- (Line **330–334**) `mmap()` maps the host binary into virtual address space of calling process (Kaal Bhairav in this case). The host is mapped at an address chosen by the kernel (specified by NULL as first parameter to mmap). **PROT_WRITE** here describes the memory protection of the mapping, i.e. pages in the mapping may be written. **MAP_SHARED** flag specifies that updates to the mapping will be reflected to file on disk as soon as any modification to mapping is performed (additionally, use of msync() is preffered). Read `$ man mmap`.
- (Line **336–337**) Finally `close()` the host binary and return the address to first byte of mapped host binary.

LoadParasite() — Poison in memory

Loading poison pill

The idea is to inject the parasite into host mapping (which has MAP_SHARED flag set). To do this, the parasite code needs to be somewhere into the process address space so that it is directly accessible when the time is right. `LoadParasite()` which takes path to file containing parasite code () as parameter and gets the parasite into memory defining the *parasite_size* and *parasite_code* global variables (which stores the size and location of parasite code in memory). It does so by -

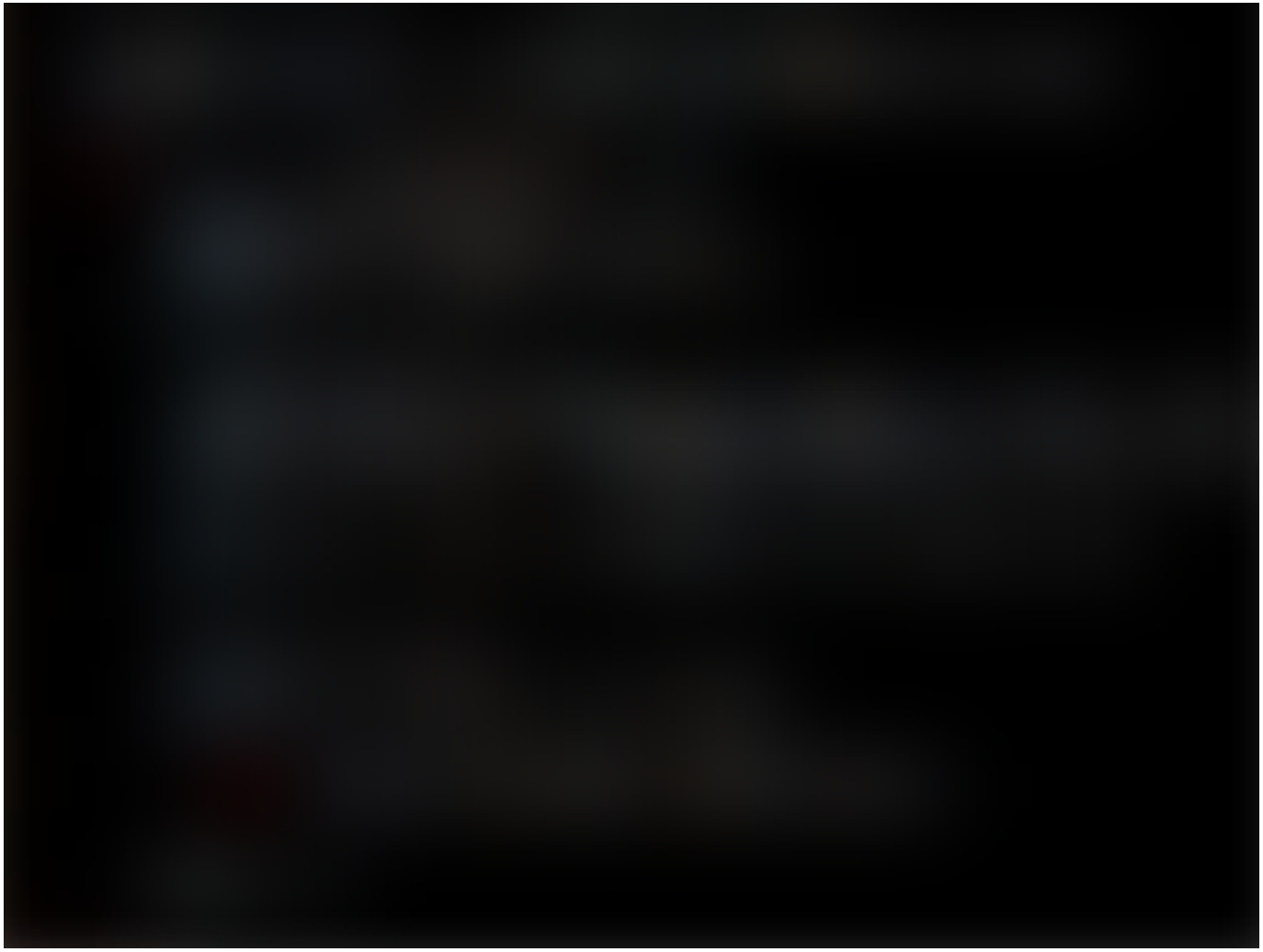
- (Line **270–282**) Opens the file containing parasite code in Read-Only mode and uses `lstat()` (previously used in `mmapFile()`) to get the

properties of the parasite file.

- (Line **284–290**) The code gets the size of parasite into *parasite_size* and allocates *parasite_size* bytes on heap segment via `malloc()` which returns the location of the allocated space to *parasite_code*. The address returned by `malloc()` is stored into parasite_code which is of type `int8_t *` (pointer to a byte value).
- (Line **292–296**) Finally we perform `read()` syscall to read *parasite_size* bytes from `parasite_fd` (parasite file descriptor returned by `open()`).

NOTE : We rely on heap segment to load parasite code, since it remains accessible until the end of program execution. If we load parasite code into an array (space for which gets allocated on the stack segment), it will get destroyed along with stack frame of `LoadParasite()` as soon as the function returns (during function epilogue).

GetPaddingSize() — That comfy shelter !



Parsing PHT

Till the point, we have mapped the host binary into process address space and we have our parasite code chilling out on the heap segment. Next, we

need to find out an accommodation into mapping of the host binary where our parasite code could sit and execute silently. The function `GetPaddingSize()` parses **P**rogram **H**eader **T**able (PHT) (of the host mapped @ *host_mapping*) to find the padding size between CODE segment and the NEXT segment after CODE segment (usually DATA segment) and returns size of padding (which lets us know whether or not the host is able to accommodate the parasite).

- (Line **211–213**) Gets the elf_header from host_mapping (which always starts from the 0th offset of the binary). From elf header, it gets and stores the offset of PHT into variable named *pht_offset* and the number of program headers (in PHT) into a variable named *pht_entry_count*.
- (Line **216**) Sets a variable named *phdr_entry* to point to the first entry in PHT which is @ (host_mapping + pht_offset)
- (Line **217–219**) Sets **CODE_SEGMENT_FOUND** to 0 which describes whether the CODE segment is found or not. Then, we start iterating through the entries of PHT starting from 0th program header.
- (Line **221–225**) Checks if the **CODE_SEGMENT_FOUND** is 0 (i.e. CODE segment is not found yet) and the segment type (specified by

p_type attribute of `struct Elf64_Phdr` defined in elf.h) is **PT_LOAD** (i.e. one of the LOAD segments of binary) and **p_flags** (permission flags attribute of program header) is set to bitwise OR of **PF_R** (Read) and **PF_X** (Execute bit) Read-Execute (R-X). If all three conditions are true, the segment is marked as CODE seg Patch parasite code to silently transfer control to original codement by setting **CODE_SEGMENT_FOUND** to 1.

- (Line **227–229**) Here the *phdr_entry* points to the current program header and the end of the segment is given by adding the **p_filesz** (size of segment on disk) to the **p_offset** (offset of current segment). Since we want to place parasite code at the end of CODE segment itself, we set *parasite_offset* to *code_segment_end_offset*. For LSB executables, we set *parasite_load_address* (i.e. location at which the parasite will be found during execution of host binary) to (p_vaddr + p_filesz) of CODE segment.
- (Line **231–232**) As per the 2nd step of infection algorithm, we increase the *filesz* (size of segment on disk) and *memsz* (size of segment in memory) attributes of current program header by *parasite_size*. This is done to accommodate parasite both on disk and in memory.

- (Line **235–240**) It is the block for DATA segment which executes if the **CODE_SEGMENT_FOUND** is set to 1 (i.e. CODE segment is already parsed and modified by above code) and segment type (`p_type` attribute of `struct Elf64_Phdr` defined in elf.h)is set to PT_LOAD (i.e. loadable segment) and segment permissions are set to Read-Write (RW-). It returns the padding_size which is calculated by subtracting offset to end of
  CODE segment from offset to DATA segment (i.e. *p_offset* (of DATA segment) — *code_segment_end_offset*).
- (Line **242–245**) It then increments *phdr_entry* to point to next program header entry. If the loop iterates through all the entries and code flow somehow reaches line number **245**, it means that we didn't get into if block of DATA segment and therefore returns 0 which marks as no space for parasite code.

NOTE : Here PF_R, PF_W and PF_X are macros which describe the access permissions of a segment and are defined in `/usr/include/elf.h`. **PF_X** is (1<<0), **PF_W** is (1<<1) and **PF_R** is (1<<2). We can use these flags by performing a bitwise OR on these macros as described in `$ man 5 elf`.

PatchSHT() — Long live our parasite !

Parsing PHT

We found a room for parasite inside host but how do we remove its insecurity of not getting thrown out of the shelter we found. There exist tools possessive to binaries (like `strip`) that don't like complementary content (i.e. anything other than what is necessary for program execution) to stay inside a binary. `Strip` discard symbols, debugging information and parts of the binary that doesn't contribute to the execution of a program.

Tools like `strip` is often used by software developers to reduce the size of binary (before releasing it for production) and also to make life harder for reverse engineers. Along with this, it removes any code/data that doesn't fall into any section, which seems scary to our parasite. To make our future trojanized host strip-safe, we'll have to parse the Section Header Table (SHT) of the binary and increase the size of the last section of CODE segment by *parasite_size* to deceive `strip` into believing that the parasite is a part of a section.

- (Line **145–150**) It sets the *sht_offset* (offset to Section Header Table) and *sht_entry_count* (number of entries in SHT) from the elf header of the host binary. Defines *section_entry* (which is of type `Elf64_Shdr *` defined in elf.h) to 1st entry of SHT.
- (Line **153–162**) Iterating over the SHT, we check if the offset to end current section (*current_section_end_offset*) is equal to the offset to end of code segment (*code_segment_end_offset*). If it is the last section of CODE segment, then increase its *sh_size* to accommodate parasite code. Later, increment the section_entry pointer to point to next entry in SHT.

FindAndReplace() — Patching the parasite

evil_elf.c

When an attacker infects a binary, he intends to hijack the code flow of the binary to make it do things which it was not intended to do (stunts performed by the parasite). The hijacking part of the process can be done via different methods of infection, depending on the infection point used by the malware author. It can be done prior to execution of main() (by **entry point modification technique** or by **hijacking constructors**), in between main()'s execution (by hijacking library functions — **GOT poisoning** or **PLT/GOT redirection**, infection via **function trampolines**) and also can be done after main() executes (**hijacking destructors**). Here, we'll be focus on entry point modification technique to hijack code flow of the host binary.

After we hijack the code flow and transfer code flow to the parasite, it is the responsibility of the parasite to silently transfer code flow back to the intended code execution of the host binary such that code is silently executed without creating any noise. To do this we need to somehow let the parasite know where it needs to transfer code flow after execution. Our utility function — `FindAndReplace()` performs this task for us. It replaces a placeholder value (inside parasite code) with the *original_entry_point* of host binary. Calling it - `FindAndReplace(parasite_code, 0xAAAAAAAAAAAAAAAA, original_entry_point);`.

- (Line **125–128**) Initializes a pointer `uint8_t *ptr` to parasite and loops through each byte of parasite code.
- (Line **130–135**) On x86–64 bit architectures, a memory address takes 8 bytes of space in memory. Keeping this in mind the *current_QWORD* is a long type variable which stores a 8-byte value. The `if statement` compares the values by XORing the *find_value* and *current_QWORD* (1 XOR 1 is 0) such that it overwrites 8 bytes of placeholder with *replace_value* and returns (i.e. *original_entry_point*)

ElfParser : Inject parasite code & unmap the binary

evil_elf.cParsing PHT

In `ElfParser()`, `memcpy()`, copies *parasite_size* bytes from location *parasite_code* (in heap segment) to (*host_mapping* + *parasite_offset*) which is the accommodation for parasite. `munmap()` unmaps the host binary from the process address space of Kaal Bhairav and infection is written to disk.

. . .

My beloved binary is sick

Sick enough to show up an abrupt behavior (at least). Infection process isn't complete until we learn to craft parasites. Parasite is what completes the infection by defining what good or evil happens after we hijack the code flow of the host. In the next article, we'll discuss parasite design which is different for both **ET_EXEC** and **ET_DYN** type binaries (i.e. for both LSB executable and shared objects).
The entire source code for the elf infector can be found here.

Cheers,
Abhinav Thakur
(a.k.a compilepeace)

Github : https://github.com/compilepeace
Linkedin : https://www.linkedin.com/in/abhinav-thakur-795a96157/

Programming | Malware | Hacking | Cybersecurity | Information Security

953 claps

WRITTEN BY
Abhinav Thakur
security researcher

Follow

Write the first response

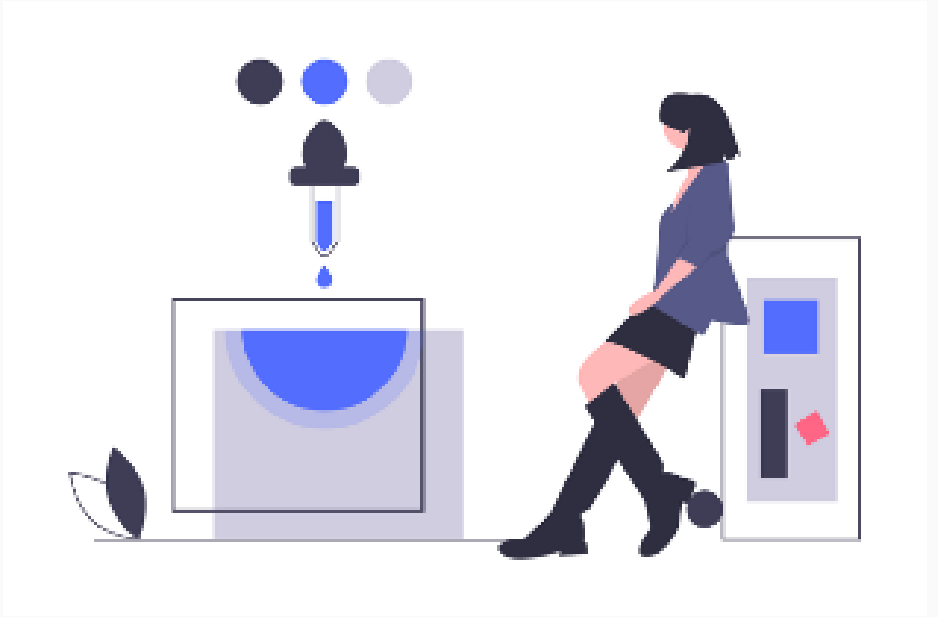
More From Medium

Also tagged Programming

Edit PDFs, Images, and Videos Using Quick Look in iOS 13

Anupam Chugh in Better Programming
Nov 14 · 4 min read ★

51



Also tagged Information Security

The Intersection Between Network Topology and Security

Some Dude Says in The Startup

Nov 14 · 7 min read ★

56

Top on Medium

The Unforgettable Relationship Advice My Ex-Girlfriend Gave Me

Nico Ryan in P.S. I Love You

Nov 6 · 5 min read ★

11.3K

Discover Medium

Welcome to a place where words matter. On Medium, smart voices and original

Make Medium yours

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade