1.1 Introduction

Computer software is, essentially, a list of instructions given to a computer to execute. At a high-level, these can instruct a computer to connect to the internet, open a file for reading or playback music. They can also instruct a computer to delete important files, encrypt their contents or send file contents to unknown entities. Software that performs malicious activities is called malware. While there are behavioral differences at a high-level, malware and benign software have no inherent differences at a low-level, i.e., their machine code. To the untrained eye, their disassembly provides no clues about intent. The disassembly of a software program is the representation of its binary content in machine-code instructions form. Fig. 1 shows the disassembly of the `main` function of a Hello World program as viewed in IDA disassembler.[1] Threat actors generally do not release the source code for their malware. As such, malware analysts have to rely on the disassembly to determine a malware's intent. This introduces the assumption that the disassembly being analyzed is actually related to the malware, and this is where we get into muddy waters.

[1] https://hex-rays.com/ida-pro/

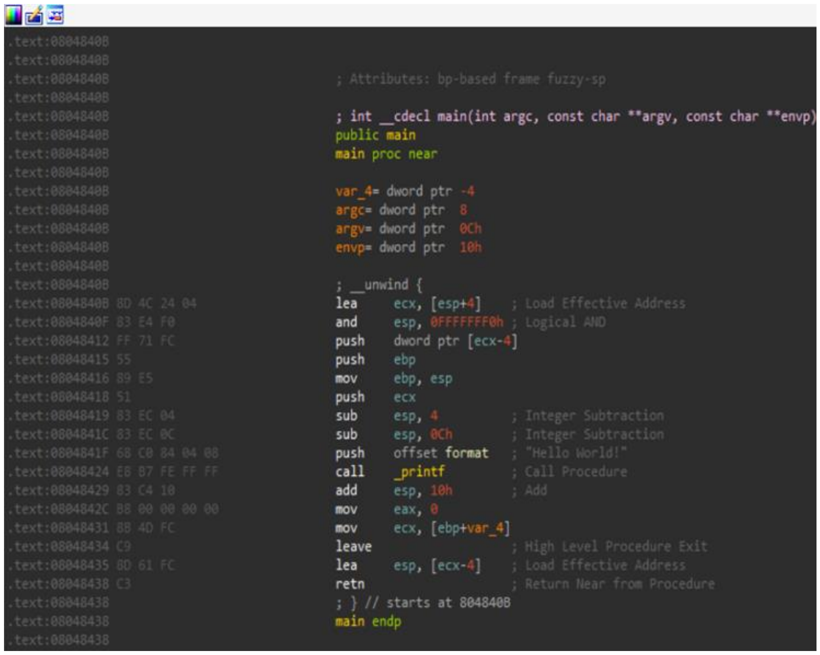

Figure 1: x86 Disassembly as viewed in IDA

Packers are software that are used to obfuscate a program's contents. They can be leveraged for both benign and malicious software. In general, a packer performs two actions: it compresses or encrypts the contents of the original binary; and it adds an unpacking code (or stub) which is executed first when the packed binary is run.[2] A common method employed by packers is to add an unpacking stub that reverses the packing process at runtime and then executes the contents of the original binary. When a malware analyst views the disassembly of a packed

[2] https://encyclopedia.kaspersky.com/glossary/packer/; https://www.welivesecurity.com/2008/10/27/an-introduction-to-packers/

binary, they are not looking at the code of the original binary. Instead, they are looking at code related to the packer, likely the unpacking stub.

When malware analysts find themselves in this situation, they have multiple options for the next step. They can place the malware under a debugger, such as x64dbg[3], wait for the moment when the unpacked binary is available in memory and then dump it to disk. Another option is to find an open-sourced tool that is capable of unpacking the said binary. The last option is to analyze the disassembly and produce an unpacking program based on the results of the analysis. There are pros and cons to each of these options. The first option is not scalable. If given a hundred binaries, an analyst would need to spend an inordinate amount of time to place each binary under a debugger to get the unpacked version. The second option requires that the open-sourced tool support the packer at hand. It can only unpack the packed binary if it knows the unpacking code for that packer. This is the best option for well-known packers, such as UPX, which are generally supported by open-source unpackers. The third option may require significant time investment depending on the sophistication of the packer. However, the produced unpacking program is scalable, and if open-sourced is

---

[3] https://x64dbg.com

also available for the community to use. This is the option we have chosen to follow, since ELFuck does not have an open-sourced unpacking tool.

ELFuck is a packer for 32-bit ELF binaries and is written mostly in C and x86 assembly.[4] It uses custom code to load the original binary into memory for execution. NRV2E algorithm is used to compress the loader and the original binary contents.[5] The unpacking stub decompresses and executes the loader and the original binary contents at runtime in memory. ELFuck has three main features which can be used in combination with each other: compression with NRV2E; polymorphic scrambler; and a password-based binary locking mechanism.[6] This paper describes the packing technique and the polymorphic scrambler used by ELFuck, and contributes an unpacking program written in the Python language. The unpacking tool leverages the Qiling framework for emulating the packed binary.[7] We do not explore the password-based binary locking feature because it is generally not leveraged by malware, which are intended to execute autonomously. We also primarily focus on malware which are targeted to little-endian systems.

---

[4] https://github.com/timhsutw/elfuck
[5] https://github.com/korczis/ucl/blob/master/src/n2_99.ch#L372
[6] https://github.com/timhsutw/elfuck/tree/master/doc
[7] https://qiling.io; https://github.com/qilingframework/qiling

1.2 ELF Format

The Executable and Linkable Format (ELF) is a standard file format for Linux-based executables, object code, shared libraries and core dumps.[8] Each ELF file contains four important parts: ELF header; program header table; section header table; and file data. More information about the ELF format is available in the Linux manual page.[9] Debugging tools such as gdb[10], readelf[11] and pyelftools[12] are capable of parsing an ELF file and returning information about it. For example, the snippet below shows readelf being leveraged to display the ELF header of an ELF binary. The ELF header describes metadata about the file such as the type of ELF file, the architecture it targets, and more.

```
$ readelf -h hello_world_dynamic
ELF Header:
    Magic:                                 7f 45 4c 46 01
01 01 00 00 00 00 00 00 00 00 00
    Class:                                 ELF32
    Data:                                  2's complement,
```

8 https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
9 https://man7.org/linux/man-pages/man5/elf.5.html
10 https://www.sourceware.org/gdb/
11 https://man7.org/linux/man-pages/man1/readelf.1.html
12 https://github.com/eliben/pyelftools

```
little endian
    Version:                           1 (current)
    OS/ABI:                            UNIX - System V
    ABI Version:                       0
    Type:                              EXEC
(Executable file)
    Machine:                           Intel 80386
    Version:                           0x1
    Entry point address:               0x8048310
    Start of program headers:          52 (bytes into
file)
    Start of section headers:          6116 (bytes
into file)
    Flags:                             0x0
    Size of this header:               52 (bytes)
    Size of program headers:           32 (bytes)
    Number of program headers:         9
    Size of section headers:           40 (bytes)
    Number of section headers:         31
    Section header string table index: 28
```

It is also important to note that these debugging tools assume that the integrity of the ELF header is not compromised. Parsing an ELF binary fails if some fields, such as `e_shoff`, in the ELF header are corrupted. In one ELF binary, we modified the value of the `e_shoff` field such that it pointed beyond the file. According to the Linux manual page, `e_shoff` holds the section header table's file offset in bytes. readelf was unable to parse the section header table due to this corruption as can be seen in the snippet below:

```
$ readelf -S hello_world
There are 31 section headers, starting at offset
0xffffffa:
readelf: Error: Reading 1240 bytes extends past end of
file for section headers
```

The program header table describes segments which contain information required by the Linux kernel to load and execute the ELF file. Only segments of type, `PT_LOAD` are loaded into memory. All other segments are mapped into one of the `PT_LOAD` segments. The snippet below shows the program header table of an ELF binary:

```
$ readelf -l hello_world_dynamic
```

Elf file type is EXEC (Executable file)

Entry point 0x8048310

There are 9 program headers, starting at offset 52

Program Headers:

  Type   Offset   VirtAddr   PhysAddr   FileSi

z MemSiz Flg Align

  PHDR   0x000034 0x08048034 0x08048034

0x00120 0x00120 R E 0x4

  INTERP   0x000154 0x08048154 0x08048154

0x00013 0x00013 R   0x1

      [Requesting program interpreter: /lib/ld-

linux.so.2]

  LOAD   0x000000 0x08048000 0x08048000

0x005c8 0x005c8 R E 0x1000

  LOAD   0x000f08 0x08049f08 0x08049f08

0x00114 0x00118 RW 0x1000

  DYNAMIC   0x000f14 0x08049f14 0x08049f14

0x000e8 0x000e8 RW 0x4

  NOTE   0x000168 0x08048168 0x08048168

0x00044 0x00044 R   0x4

```
  GNU_EH_FRAME   0x0004d0 0x080484d0 0x080484d0
0x0002c 0x0002c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000
0x00000 0x00000 RW  0x10
  GNU_RELRO      0x000f08 0x08049f08 0x08049f08
0x000f8 0x000f8 R   0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01      .interp
   02      .interp .note.ABI-tag … … .text .fini …
   03      .init_array .fini_array .jcr .dynamic .got
.got.plt .data .bss
   04      .dynamic
   05      .note.ABI-tag .note.gnu.build-id
   06      .eh_frame_hdr
   07
   08      .init_array .fini_array .jcr .dynamic .got
```

Note that the virtual address of the first `PT_LOAD` segment (in bold above) is the same as the base address of the ELF binary in memory. Consequently, the ELF header is also part of the first `PT_LOAD` segment. Different segments may have different access flags. For example, the first `PT_LOAD` segment above has read-execute permissions (likely contains executable instructions), while the other has read-write permissions (likely contains data). It is difficult to enforce access attributes if two such segments are in the same page in memory. For this reason, segments are aligned with the system page size (usually 4 KB).[13]

The section header table describes sections which contain information required for relocations (handled by the ELF static linker) and the ELF dynamic linker. It is primarily useful for debugging tools such as gdb, readelf and pyelftools which use it to locate section information, notably the .symtab (symbol table) and .shstrtab (section name strings) sections. It is important to note that the section header table is not required to successfully load and execute an ELF binary. The snippet below shows the section header table of an ELF binary:

```
$ readelf -S hello_world_dynamic
There are 31 section headers, starting at offset
0x17e4:

Section Headers:
```

[13] https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments/

```
  [Nr]
Name              Type            Addr     Off    Size
  ES Flg Lk Inf Al
  [ 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
   …
   …
  [14] .text            PROGBITS        08048310
000310 000192 00  AX  0   0 16
   …
   …
  [28] .shstrtab        STRTAB          00000000
0016d9 00010a 00      0   0  1
```

```
  [29] .symtab           SYMTAB          00000000
001054 000450 10     30  47  4
  [30] .strtab           STRTAB          00000000
0014a4 000235 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S
(strings), I (info),
  L (link order), O (extra OS processing required), G
(group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E
(exclude),
  p (processor specific)
```

1.3 ELFuck Features

1.3.1 Packing via Compression

The Linux kernel requires data in `PT_LOAD` segments to successfully load and execute an ELF binary. The ELF dynamic linker (relevant for dynamically-linked ELF binaries) requires the interpreter string (refers to an ELF binary and outlined in the `PT_INTERP` segment) to perform dynamic relocations at load time. Consequently, ELFuck extracts the contents of only the `PT_LOAD` and

PT_INTERP segments of the original ELF binary. It keeps track of the lowest and the highest virtual address at which the contents of these segments exist on-disk and in-memory. The interpreter, if any, is copied to the end of ELFuck's ELF loader. If PT_INTERP segment is not found, such as for statically-linked ELF binaries, the interpreter loading code in ELFuck's ELF loader is zeroed.

ELFuck creates a single page-aligned PT_LOAD segment. Its size is equal to the sum of the loader; interpreter string, if any; and difference of the highest and lowest page-aligned virtual addresses of the PT_LOAD segments of the original ELF binary. The ELF loader and interpreter string, if any, are first copied into this memory region. They are followed by the contents of the PT_LOAD segments. This memory region is then compressed with NRV2E algorithm.

ELFuck then places an auxiliary vector containing information required by the ELF dynamic linker. This information includes the address of the program header table, number of entries in it and entry point of the original ELF binary. Finally, an unpacking stub and banner are added prior to the memory address containing compressed data. It is possible to skip the addition of the banner through a command-line argument to the ELFuck program. An ELF header and program header table are constructed and added before the banner. A rough memory layout of the packed binary is shown below:

ELF Header
Program Header Table
ELFuck Banner
Unpacking Stub
Compressed Data
Auxiliary vector for ELF Dynamic Linker

Table 1: Memory layout of packed binary

The `EI_CLASS` and `EI_DATA` fields in the packed binary's ELF header are left empty (aka, zero) which causes parsing issues for debugging tools such as gdb and pyelftools as shown in the snippets below:

```
$ ipython
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
Type 'copyright', 'credits' or 'license' for more
information
IPython 7.30.1 -- An enhanced Interactive Python. Type
'?' for help.

In [1]: from elftools.elf.elffile import ELFFile
```

```
In [2]: data =
ELFFile(open('hello_world_dynamic_packed', 'rb'))
---------------------------------------------------------
----------------------
ELFError                                  Traceback
(most recent call last)
<ipython-input-2-68a6730f2411> in <module>
----> 1 data =
ELFFile(open('hello_world_dynamic_packed', 'rb'))

~/.local/lib/python3.8/site-
packages/elftools/elf/elffile.py in __init__(self,
stream)
     71     def __init__(self, stream):
     72         self.stream = stream
---> 73         self._identify_file()
     74         self.structs = ELFStructs(
     75             little_endian=self.little_endian,

~/.local/lib/python3.8/site-
```

```
packages/elftools/elf/elffile.py in
_identify_file(self)
    488             self.elfclass = 64
    489         else:
--> 490             raise ELFError('Invalid EI_CLASS
%s' % repr(ei_class))
    491
    492         ei_data = self.stream.read(1)

ELFError: Invalid EI_CLASS b'\x00'


$ gdb -q ./hello_world_dynamic_packed
“./hello_world_dynamic_packed”: not in executable
format: file format not recognized
(gdb) quit
```

The C-based code that performs the above functions is available in ELFuck’s GitHub repository.[14]

[14] https://github.com/timhsutw/elfuck/blob/master/src/stubify.c

1.3.2 Polymorphic Scrambler

ELFuck uses a polymorphic scrambler to produce a unique packed binary each time. This is effective for evading hash and pattern-based detection systems. It achieves this by adding junk instructions and encrypting the unpacking stub, compressed data and auxiliary vector with different keys each time. Encryption keys are generated using C `rand()` function and the seed to `rand()` is the epoch time at the time of packing. The memory layout of an ELF binary packed by leveraging ELFuck polymorphic features is shown below:

ELF Header
Program Header Table
ELFuck Banner
Polymorphic Descrambler (Mix of Junk bytes / instructions and decryption instructions)
Encrypted Unpacking Stub, Encrypted Compressed Data, Encrypted Auxiliary vector for ELF Dynamic Linker

Table 2: Memory layout of packed program with polymorphic scrambler

The snap below shows an example of junk instructions interleaved with instructions related to the decryption algorithm. Instructions at address `0x80460AF` and `0x80460BF` are related to the decryption algorithm (moves decryption keys into registers) while the others are junk instructions. Although this kind of polymorphism is unsophisticated, it presents an additional layer of difficulty to the novice analyst.

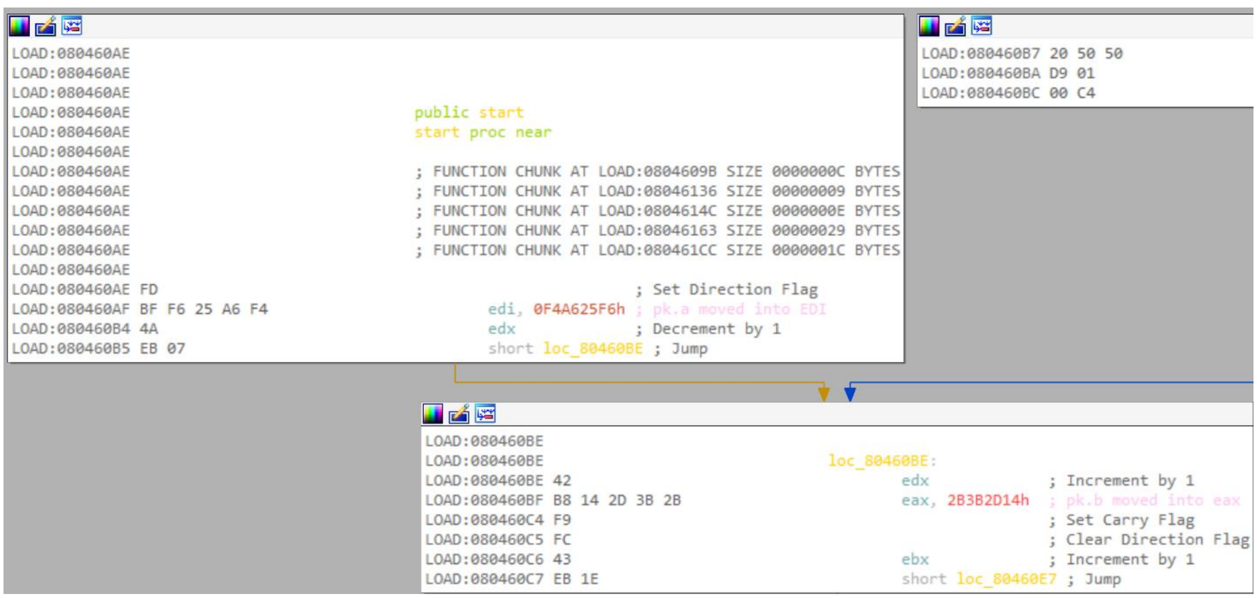

Figure 2: Polymorphic Junk Instructions + Decryption Algorithm Instruction

A C-based encryption code is used to encrypt the unpacking stub, compressed data and auxiliary vector. It is available in the ELFuck repository.[15] The code below shows the Python version:

```
len_data = (len(data_to_compress) + 3) >> 2
for i in range(len_data):
    data[i] += key2
    data[i] ^= key1
    key1 += key2
```

The decryption code is shown below:

```
for i in range(len(encrypted_data)):
    encrypted_data[i] ^= key1
    encrypted_data[i] -= key2
    key1 += key2
```

1.3.3 ELFuck ELF Loader

Before diving into ELFuck's custom loader, we wanted to present some contextual information. The stack structure[16] of a loaded and initialized process is

[15] https://github.com/timhsutw/elfuck/blob/master/src/poly.c
[16] https://articles.manugarg.com/aboutelfauxiliaryvectors.html

shown below. This structure allows a newly running program to figure out where information on the stack is located.

```
position            content                     size (bytes) + comment
  ------------------------------------------------------------------------
  stack pointer ->  [ argc = number of args ]     4
                    [ argv[0] (pointer) ]         4   (program name)
                    [ argv[1] (pointer) ]         4
                    [ argv[..] (pointer) ]        4 * x
                    [ argv[n - 1] (pointer) ]     4
                    [ argv[n] (pointer) ]         4   (= NULL)

                    [ envp[0] (pointer) ]         4
                    [ envp[1] (pointer) ]         4
                    [ envp[..] (pointer) ]        4
                    [ envp[term] (pointer) ]      4   (= NULL)

                    [ auxv[0] (Elf32_auxv_t) ]    8
                    [ auxv[1] (Elf32_auxv_t) ]    8
                    [ auxv[..] (Elf32_auxv_t) ]   8
                    [ auxv[term] (Elf32_auxv_t) ] 8   (= AT_NULL vector)

                    [ padding ]                   0 - 16

                    [ argument ASCIIZ strings ]   >= 0
                    [ environment ASCIIZ str. ]   >= 0

  (0xbffffffc)      [ end marker ]                4   (= NULL)

  (0xc0000000)      < bottom of stack >           0   (virtual)
```

Figure 3: Stack structure of initialized process

For our purposes, the auxiliary vector is the most important. It lies immediately after the environment variable values on the stack and is primarily used by the program interpreter. The auxiliary vector comprises of an array of structures of type `Elf32_auxv_t`:[17]

```
typedef struct
{
    uint32_t a_type;              /* Entry type */
    union
    {
        uint32_t a_val;           /* Integer value */
        /* We use to have pointer elements added here.
We cannot do that,
           though, since it does not work when using
32-bit definitions
           on 64-bit platforms and vice versa.  */
    } a_un;
} Elf32_auxv_t;
```

[17] https://sourceware.org/git/?p=glibc.git;a=blob;f=elf/elf.h#l1138

Acceptable entry types are declared in `auxvec.h`[18]. Earlier, we mentioned that during the packing process ELFuck places an auxiliary vector in the packed binary. However, it only places the value of each entry and not the type. This works because the values are placed in order, so ELFuck's loader known which values belong to which type. Now that we have some contextual information, we'll explore how ELFuck sets up the stack to get the packed binary up and running.

Statically-linked ELF binaries do not require a program interpreter to perform additional linking (aka dynamic linking), so ELFuck's loading process is fairly straightforward. It modifies the values of `AT_PHDR`, `AT_PHNUM`, `AT_ENTRY` fields in the auxiliary vector to reflect that of the original binary[19] and then jumps to the entry point of the original ELF binary.

The algorithm is more involved for dynamically-linked executables. Earlier in section 1.3.1, we mentioned that the interpreter string is copied to the end of ELFuck's ELF loader. This string points to the Linux ELF loader on disk, which in itself is and must be an ELF binary. ELFuck's loader opens this file and reads the first 4096 bytes. It loads all `PT_LOAD` segments of the Linux ELF interpreter

[18] https://github.com/torvalds/linux/blob/v3.19/arch/ia64/include/uapi/asm/auxvec.h; https://github.com/torvalds/linux/blob/v3.19/include/uapi/linux/auxvec.h
[19] https://github.com/timhsutw/elfuck/blob/master/src/execelf.S#L271-L283

into memory. It modifies the values of `AT_PHDR`, `AT_PHNUM` fields in the auxiliary vector to reflect that of the original binary. The value of `AT_ENTRY` field in the auxiliary vector is set to the entry point of the Linux ELF interpreter and control jumps to this address. The Linux ELF interpreter then performs dynamic linking and passes control to the original binary.

The manner in which ELFuck loads the Linux ELF interpreter into memory seems to be faulty. This is reflected by an error in the run-time dynamic linker (aka, the Linux ELF interpreter). We've not been able to execute a simple packed dynamically-linked HelloWorld program:

```
$ ./hello_world_dynamic_packed
Inconsistency detected by ld.so: rtld.c: 1206:
dl_main: Assertion `GL(dl_rtld_map).l_libname->next ==
NULL' failed!
```

The author of ELFuck is aware of this[6]: `The name is not random, the way we're loading ELF binary into memory is not so clean, so things are just getting fucked up sometimes.` It is part of our future work to identify the cause of this inconsistency and fix it in the ELFuck loader.

1.4 Unpacking Tool

To unpack a packed ELF binary and dump the original, we emulate it until the unpacked binary is available in memory. We used the Qiling framework for emulation.

As mentioned earlier, `EI_CLASS` and `EI_DATA` fields in the packed binary are set to zero. This results in a parsing error in pyelftools which is internally used in the Qiling framework. The first step taken by the unpacking tool is fixing these fields in the ELF header. `EI_CLASS` is set to `ELFCLASS32` since ELFuck only operates on 32-bit ELF binaries. We assume that the original binary also targets little-endian systems, so `EI_DATA` is set to `ELFDATA2LSB`. Once the headers are fixed, we dump the corrected ELF binary to disk.

The unpacking tool leverages the Qiling framework for emulating the packed binary. The emulation requires two arguments: file path to the binary and root filesystem path of 32-bit Linux. The root filesystem is available in the Qiling framework GitHub repository.[20] We hooked all instructions and waited for the first `scasb` instruction to hit. This signaled the end of decompression and `EDI`

---

20 https://github.com/qilingframework/rootfs/tree/d8a9b0d6c52a3c5bc627c055d5f711dacbb1a1f6/x86_linux

+ 1 pointed to either the first PT_LOAD segment of the unpacked binary, or the ELF interpreter string that was copied to the end of the ELF loader by ELFuck's packing algorithm. If it points to the ELF interpreter string, we jump over those bytes until we reach a PT_LOAD segment that starts with the ELF magic, \x7fELF. The memory address of this PT_LOAD segment marks the base address of the unpacked binary and emulation is terminated.

The next step is to read the bytes of the memory region containing the unpacked binary. The first argument to read() is the base address of the unpacked binary, say B, which we previously determined. The second argument is the number of bytes to read. To ensure that all data belonging to PT_LOAD segments are read, we needed the highest address, say H, at which such data exists. We could then subtract the base address, B of the unpacked binary from it to get the number of bytes to read. In the packed binary, p_memsz attribute of the only entry in the program header table is calculated as the difference of H and the base address of the packed binary in memory. Since this is the first and only PT_LOAD segment, its p_vaddr attribute value is equal to the base address of the packed binary. Thus, we can calculate H as the summation of p_memsz and p_vaddr. Moreover, the number of bytes to read is equal to the difference of H and B, where H is the highest memory address where data of the unpacked binary can exist, and B is the base address of the unpacked binary

We also extract the number of entries, i.e., e_phnum, in the program header table of the original binary. The ESI register points to the auxiliary vector which was placed after the compressed data by ELFuck's packing routine. This vector contains the value of e_phnum. This value is required for the tool to traverse the program header table and correct the file offsets of each entry, in a process called file unmapping. For each PT_LOAD entry in the program header table of the unpacked binary we subtract the segment's virtual address, i.e., p_vaddr from the base address of the unpacked binary and assign it to the p_offset field of the said entry. If the binary contains a PT_INTERP segment, we subtract the length of the interpreter string from p_offset as well. Remember that the interpreter string existed before the first PT_LOAD segment in the decompressed region, thus pushing ahead offsets by the length of the interpreter string. This unmapping process corrects that offset.

The unmapped ELF binary is dumped to disk and ready for analysis. It is important to note that there is no section header table in the unpacked ELF binary. It was lost in the packing process. Any tool that relies on the section header table to find sections containing information helpful to debugging will fail to find it. For example, IDA will not be able to determine function names if it cannot find .symtab or .strtab sections.

The Python-based unpacking tool is available in the supplementary material. It was tested with `Python v3.6.9` and ELFuck at commit `5e60852b1fc2f1b5eb5d8834152eeffd0f8b3597`. An example is shown below:

```
$ python3 deob.py -f hello_world_dynamic_packed_poly -
-fs ~/qiling/examples/rootfs/x86_linux
[+] Profile: Default
[+] Map GDT at 0x30000 with GDT_LIMIT=4096
[+] Write to 0x30018 for new entry
b'\x00\xf0\x00\x00\x00\xfeO\x00'
[+] Write to 0x30028 for new entry
b'\x00\xf0\x00\x00\x00\x96O\x00'
[+] Mapped 0x8046000-0x804b000
[+] mem_start : 0x8046000
[+] mem_end   : 0x804b000
[+] mmap_address is : 0x774bf000
```

```
$ chmod +x
hello_world_dynamic_packed_poly_fixed_unpacked
```

```
$ ./hello_world_dynamic_packed_poly_fixed_unpacked
Hello World!
```

It is good that even though a packed dynamically-linked ELF binary may not execute (as we noted earlier), we can still unpack it and retrieve the original binary. The disassembly below shows the difference between a packed ELFuck binary and an unpacked ELF binary:

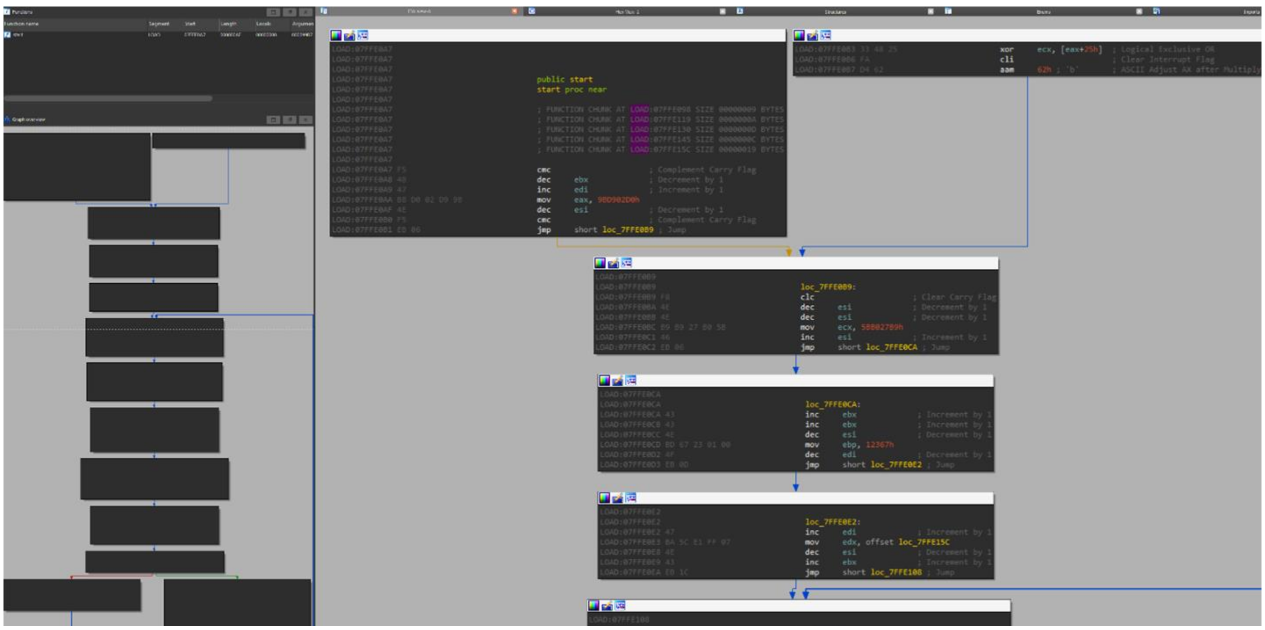

Figure 4: ELFuck-packed Hello World ELF Binary

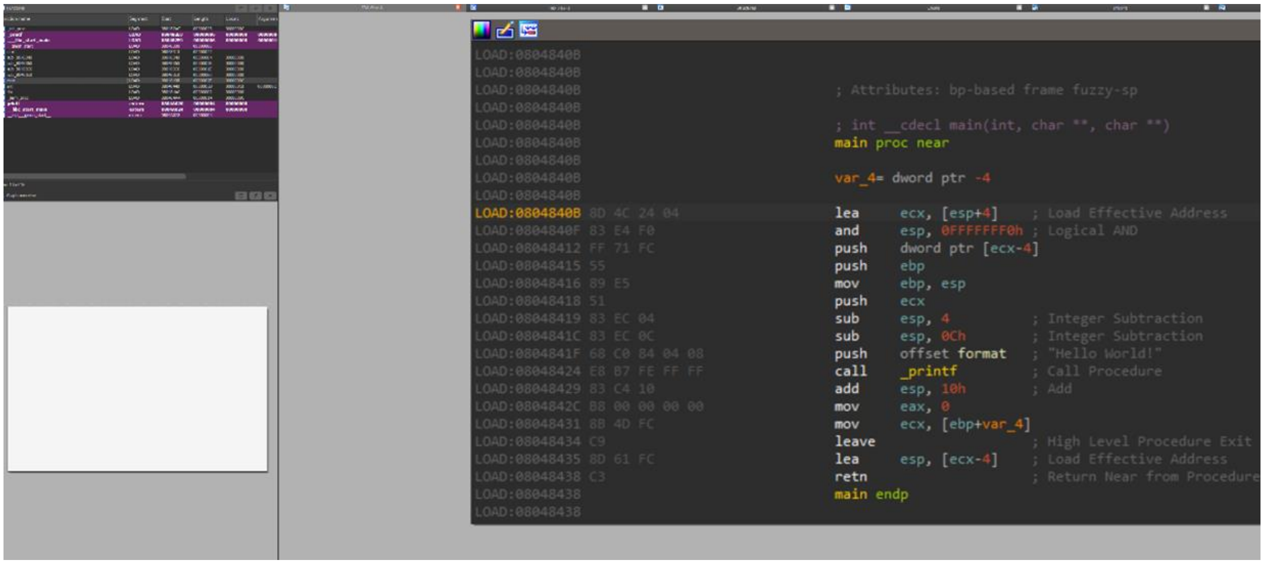

Figure 5: Hello World ELF Binary after Unpacking

1.5 Conclusion

ELFuck is a packer for 32-bit ELF binaries. While we were not able to successfully execute a packed dynamically-linked ELF binary, we could execute a packed statically-linked ELF binary. It is a relatively old software, but it can still be used to pack the latest malware and evade hash or pattern-based detection systems. Blue teams can use our Python-based unpacking script in their detection systems to unpack ELF binaries packed with ELFuck. This will enable them to be more effective with their detection content.