2024
DUBLIN

2 - 4 October, 2024 / Dublin, Ireland

DETECTING SHARED OBJECT INJECTION

Daniel Jary
Independent researcher, UK

danieljary90@googlemail.com

ABSTRACT

This paper outlines the different methods of code injection on *Linux* and how they can be achieved. Part 1 details a few techniques related to shared object loading. Shared objected injection (which is the same as DLL injection in *Windows*) can be a clean and effective solution that suits the need of most attackers and runtime malware. Under normal circumstances a loaded shared object provides additional capability to a process by supplying exported functions. However, an attacker can modify, insert, or even execute their own shared object(s) in memory to perform malicious activities.

We will cover the following techniques that can be used to maliciously load shared objects:

- DT_NEEDED infections
- LD_PRELOAD injection to leverage user-level hooking.

Part 2 of the paper will instead focus on more dynamic methods that facilitate shared object injection into processes at runtime, such as:

- Direct use of *__libc_dlopen_mode()*
- Reflective shared object injection.

We also look at why sometimes methods designed to hide a shared object can actually make detection easier.

Unlike many excellent publications which focus on the orchestration of attack techniques, this paper is instead written to provide actionable detection methodologies that can be used not just to identify if a technique has taken place but also to enrich that detection with additional data to help an analyst:

- Determine if the technique has been used for legitimate/malicious purposes.
- Provide more context either to suggest what has been leveraged from that attack (e.g. hooked function calls) and/or provide a starting point to enable a deeper investigative dive (e.g. exact location and size of injected shared object in memory so an analyst can extract it and perform their own static analysis).

NOTE: Throughout this paper assume that the terms 'shared object', 'library' and 'module' can be used interchangeably.

PART 1

ELF SEGMENTS AND SECTIONS

When performing detection at scale the focus is on running processes rather than every single ELF binary on disk. This decision has been made for following reasons:

1. Scanning every binary on disk, although possible, would require a high resource demand on individual hosts, likely creating a noticeable performance impact for the daily user.
2. Some shared object injection techniques only occur in memory at runtime, thus would be missed by disk-only forensics.

On disk, an ELF binary is represented via a section view. Sections represent the smallest indivisible units that can be processed within an ELF file. Each section can be guaranteed to contain data that is only relevant to itself – for instance, the *.text* section contains only the executable instructions of a program; the *.rela.plt* section will contain the relocation sections for the procedure linkage table (PLT); and the *.interp* section will contain the path name of a program interpreter to use. Understanding sections and collecting their data is vital to developing detection strategies.

On disk, locating each section is a straightforward process. Within the executable header (which is guaranteed to start at the beginning of an ELF file) there will exist an offset to the section header table and a field specifying the number of table entries present. Each section header table entry will provide the name, type, address and size of an individual section. Using this in combination with a known structure of each section allows one to parse its data.

However, working with processes rather than static binaries introduces an additional layer of complexity. When a binary is loaded into memory it moves from a section view to a segment view. Segments are collections of sections that represent the smallest individual units that can be mapped into memory by *exec()* or by the runtime linker. Sections are grouped into segments based on their intended page permissions – for instance, the *.text* section needs to be in a readable and executable (RX) region of the process image, thus it is grouped together with other RX sections into a segment with RX page permissions. Each segment needs to be aligned with a page boundary since a page in memory cannot be assigned two different sets of permissions at any one time. And since each group of sections won't always add up to an exact multiple of pages, padding bytes must be introduced to make up any shortfalls. This increases the process image size when compared to the ELF binary on disk. A visual representation showing the translation between the section view and segment view can be seen in Figure 1.

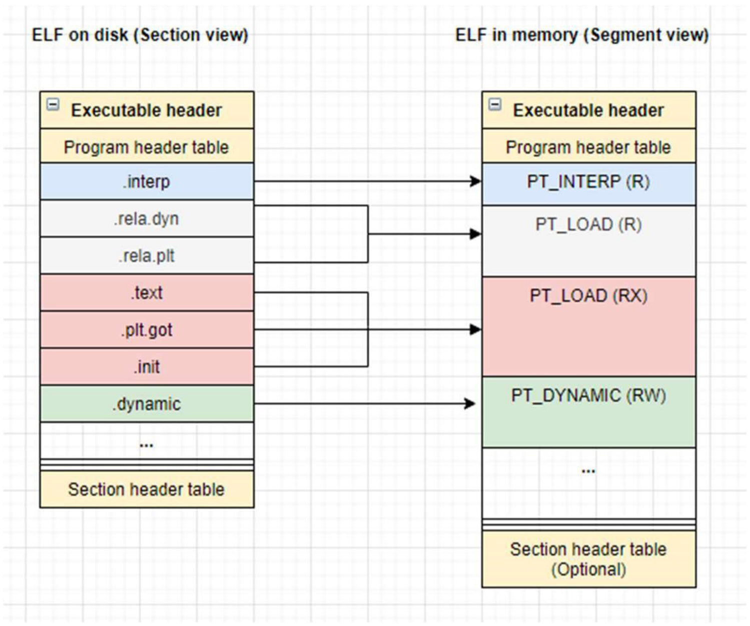
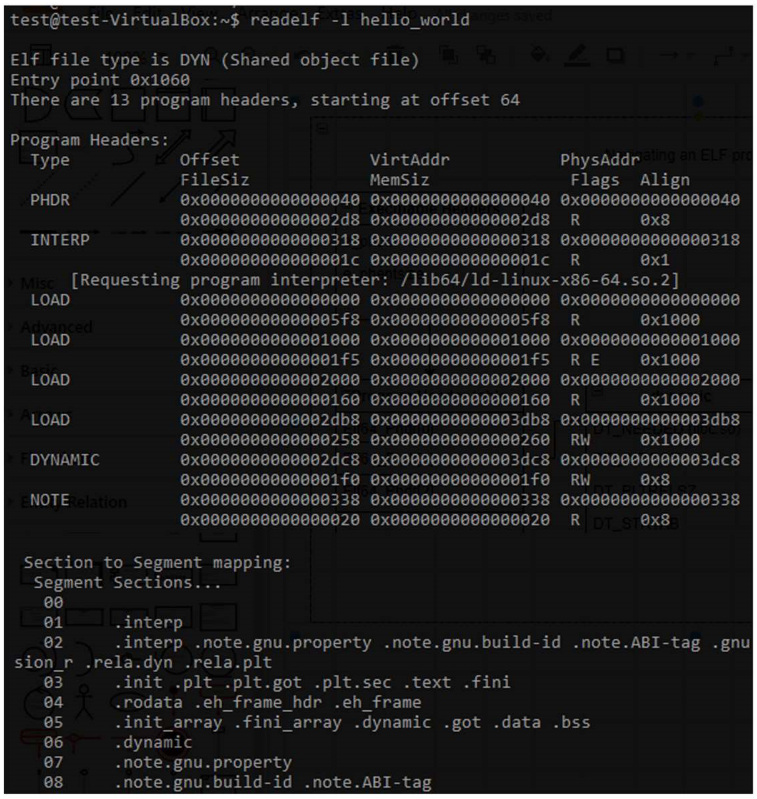


Figure 1: ELF section vs segment view.

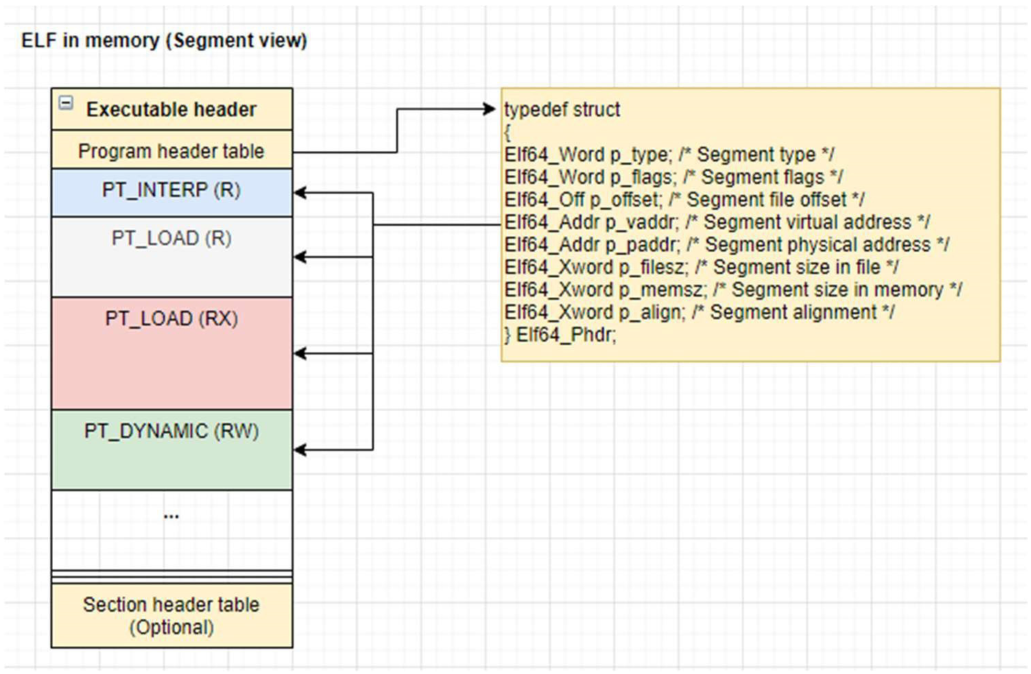
The equivalent view can also be portrayed using the *readelf* tool, which is part of *binutils* [1], showing the section to segment mappings, as shown in Figure 2.



```
test@test-VirtualBox:~$ readelf -l hello_world

Elf file type is DYN (Shared object file)
Entry point 0x1060
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000005f8 0x00000000000005f8  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x00000000000001f5 0x00000000000001f5  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x0000000000000160 0x0000000000000160  R      0x1000
  LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000258 0x0000000000000260  RW     0x1000
  DYNAMIC        0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000020 0x0000000000000020  R      0x8

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu
sion_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
```

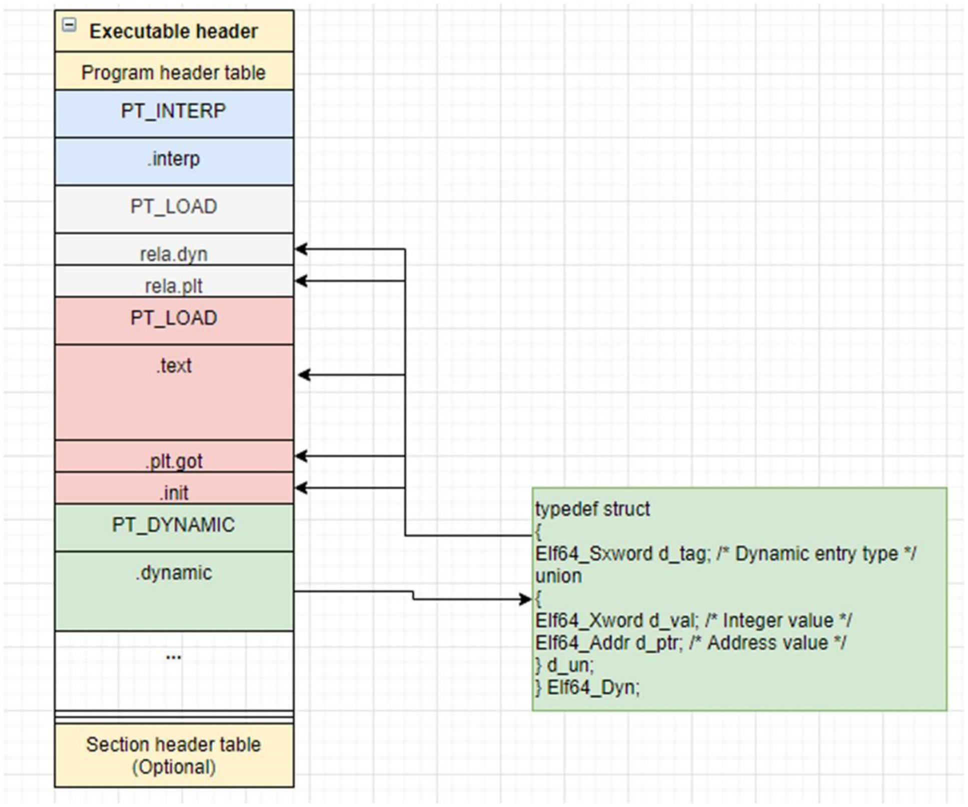
Figure 2: ELF section to segment mappings.

Another difference in memory is that the section header table is optional, and so can be deliberately removed by an attacker, thus it can't be relied upon to locate individual sections. Not knowing the starting location and size of each section makes it far more difficult to precure forensically valuable section data. One method that can be used to reconstruct some (but not all) of the sections is to use the program header table to enumerate through the different program headers and find the dynamic segment (Figure 3). Using this segment one can then start to rebuild individual sections, identifying their type, name, and sometimes their size (Figure 4). This process of recovering sections from memory is described in greater depth throughout the paper.



```
typedef struct
{
Elf64_Word p_type; /* Segment type */
Elf64_Word p_flags; /* Segment flags */
Elf64_Off p_offset; /* Segment file offset */
Elf64_Addr p_vaddr; /* Segment virtual address */
Elf64_Addr p_paddr; /* Segment physical address */
Elf64_Xword p_filesz; /* Segment size in file */
Elf64_Xword p_memsz; /* Segment size in memory */
Elf64_Xword p_align; /* Segment alignment */
} Elf64_Phdr;
```

Figure 3: Using the program header table entries to locate program segments.



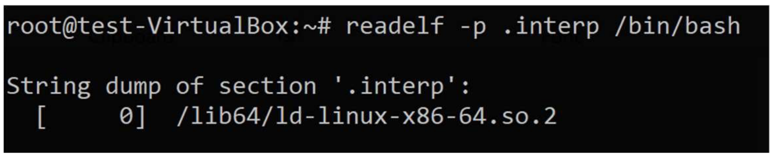
```
typedef struct
{
Elf64_Sxword d_tag; /* Dynamic entry type */
union
{
Elf64_Xword d_val; /* Integer value */
Elf64_Addr d_ptr; /* Address value */
} d_un;
} Elf64_Dyn;
```

Figure 4: Using the dynamic segment to locate sections within segments.

A more comprehensive breakdown of the ELF file format is excellently illustrated by *Intezer* in its 'Executable and Linkable Format 101' blog post series [2].

DT_NEEDED INFECTIONS

For dynamic linked binaries (any executable binary that isn't statically linked) the OS will execute the binary's interpreter, which is normally the dynamic linker *ld-linux.so.**. This can easily be identified since it will be stored in a binary's PT_INTERP segment. Using the *readelf* tool one can extract its path, as shown in Figure 5.

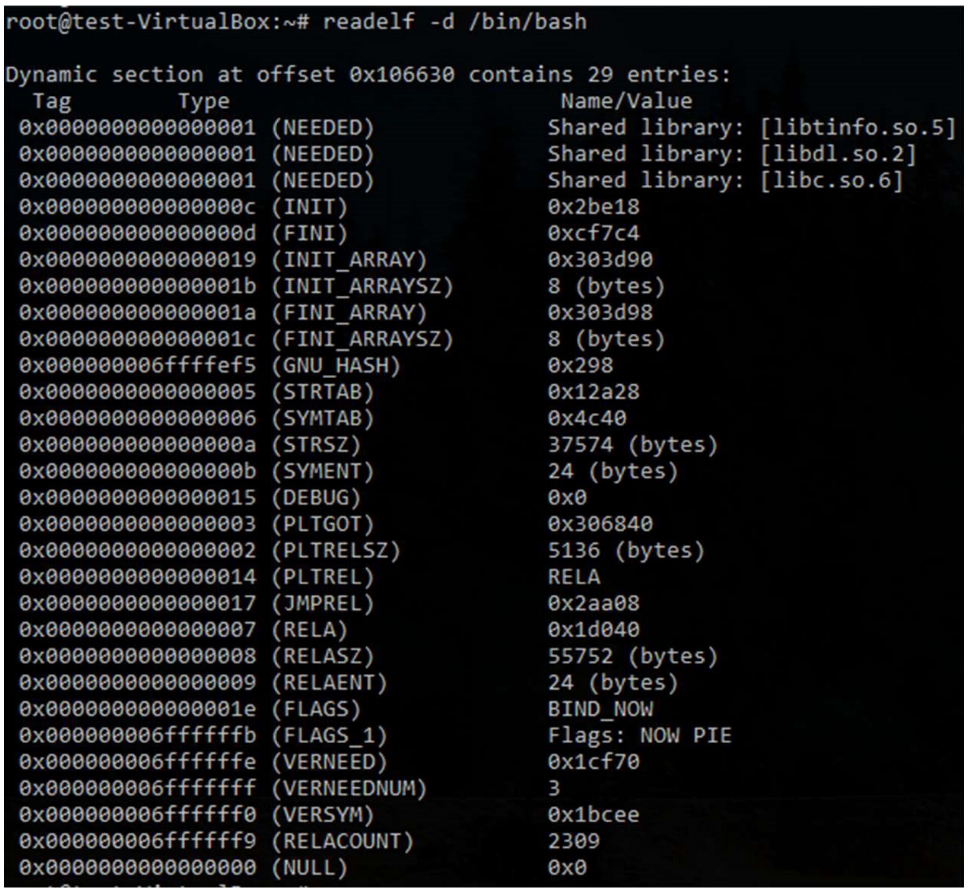


```
root@test-VirtualBox:~# readelf -p .interp /bin/bash

String dump of section '.interp':
  [     0]  /lib64/ld-linux-x86-64.so.2
```

Figure 5: Using readelf to find an ELF binary's dynamic linker.

Figure 6 shows the *.dynamic* section parsed by *readelf* (this is a 1:1 mapping of the dynamic segment). The dynamic linker uses this section to set up the environment when launching a process from the executed binary. Essentially, the *.dynamic* section is an array of ELFN_DYN structs that each contain information about a specific part of the binary that the dynamic linker needs to know about. Each ELFN_DYN struct will contain a tag that denotes what type of important information the remainder of the fields within the struct relate to. There are six tags of particular interest to the dynamic linker, they are:

- DT_NEEDED – this entry type indicates the name of a required dependency to load, i.e. a shared object file.
- DT_SYMTAB – this entry type holds the address of the dynamic symbol table.
- DT_FLAGS – this entry type holds flag values that indicate how to process loadable objects. For instance, a value of DT_BIND_NOW instructs the dynamic linker to process all relocations for DT_NEEDED entries prior to transferring control to the entry point of a program.
- DT_PLTGOT – this entry points to the Global Offset Table (GOT) on x86 architectures, which is an array of relocation addresses for symbols. During execution the runtime linker processes these relocations to determine the absolute addresses of the associated symbol values. (Essentially, the GOT is the ELF equivilent of the Import Address Table [IAT] in PE files).
- DT_RPATH & DT_RUNPATH – if present, these entries will point to a path string that determines which directory the dynamic linker should look to load libraries from.



```
root@test-VirtualBox:~# readelf -d /bin/bash

Dynamic section at offset 0x106630 contains 29 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libtinfo.so.5]
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x2be18
 0x000000000000000d (FINI)               0xcf7c4
 0x0000000000000019 (INIT_ARRAY)         0x303d90
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x303d98
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x298
 0x0000000000000005 (STRTAB)             0x12a28
 0x0000000000000006 (SYMTAB)             0x4c40
 0x000000000000000a (STRSZ)              37574 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x306840
 0x0000000000000002 (PLTRELSZ)           5136 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x2aa08
 0x0000000000000007 (RELA)               0x1d040
 0x0000000000000008 (RELASZ)             55752 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000000000001e (FLAGS)              BIND_NOW
 0x000000006ffffffb (FLAGS_1)            Flags: NOW PIE
 0x000000006ffffffe (VERNEED)            0x1cf70
 0x000000006fffffff (VERNEEDNUM)         3
 0x000000006ffffff0 (VERSYM)             0x1bcee
 0x000000006ffffff9 (RELACOUNT)          2309
 0x0000000000000000 (NULL)               0x0
```

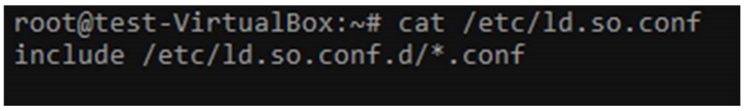
Figure 6: Dynamic section of /bin/bash parsed by readelf.

Figure 6 shows the dynamic section for the *bash* binary. Notice that the first three ELFN_DYN struct entries are of type DT_NEEDED, indicating to the dynamic linker */lib/ld-linux.so.** that the bash process depends on the *libtinfo.so.5*, *libdl.so.2* and *libc.so.6* shared objects. In addition, the DT_FLAGS entry is set to BIND_NOW, indicating that the aforementioned shared objects should be loaded into memory prior to handing control over to the entry point of the bash binary.

Notice that the DT_NEEDED entries are not accompanied by a path (this is standard compiler behaviour). Instead, the linker uses one of four ways to determine the full path of the shared object file; these appear in order of precedence:

1. First, the linker will look for the existence of a DT_RPATH entry in the *.dynamic* section, the value of which points to an offset in the DT_STRTAB of null-terminated library search paths (DT_STRTAB is the string table section for a binary).
2. Next, the linker will inspect the environmental variable LD_LIBRARY_PATH, which if set will hold a list of colon-separated directories to search within.
3. Thirdly, the linker will look for the existence of a DT_RUNPATH entry in the *.dynamic* section. This is essentially the same DT_RPATH except it is assigned a lower precedence.
4. Finally, if the previous methods have been unsuccessful the linker will look in the default directories as specified in */etc/ld.so.conf.d/*.conf.*

One notable exception to this search order is the LD_PRELOAD feature, which is discussed later within its own section. It is also worth noting that the location of the default directories can be modified using the *ldconfig* tool to point somewhere other than */etc/ld.so.conf.d/*.conf*, as seen in Figure 7. On some distributions the same can also be achieved by setting the LD_CONFIG environmental variable.



```
root@test-VirtualBox:~# cat /etc/ld.so.conf
include /etc/ld.so.conf.d/*.conf
```

Figure 7: ld.so.conf file.

Knowing all this, an attacker can go about injecting their own shared object references into existing binaries. All they need to do is add or modify a DT_NEEDED entry with the name of the shared object they wish to inject and then place it into a loadable path. Then, upon execution, the dynamic linker will perform the loading and resolution of symbols for them. The following sections detail two techniques that achieve this, and how they can be detected.

Method 1: Overwriting/appending new entries to the dynamic section

The first method involves either the modification of an existing ELFN_DYN entry or the addition of a new one to the end of the *.dynamic* section. This method is deemed simpler since it negates the need to perform relocations on existing ELFN_DYN entries. The only caveats being that there must be enough space at the end of the *.dynamic* section to tack-on a new DT_NEEDED entry *or* that the existing entry selected can be safely overwritten without causing the process to malfunction upon execution.

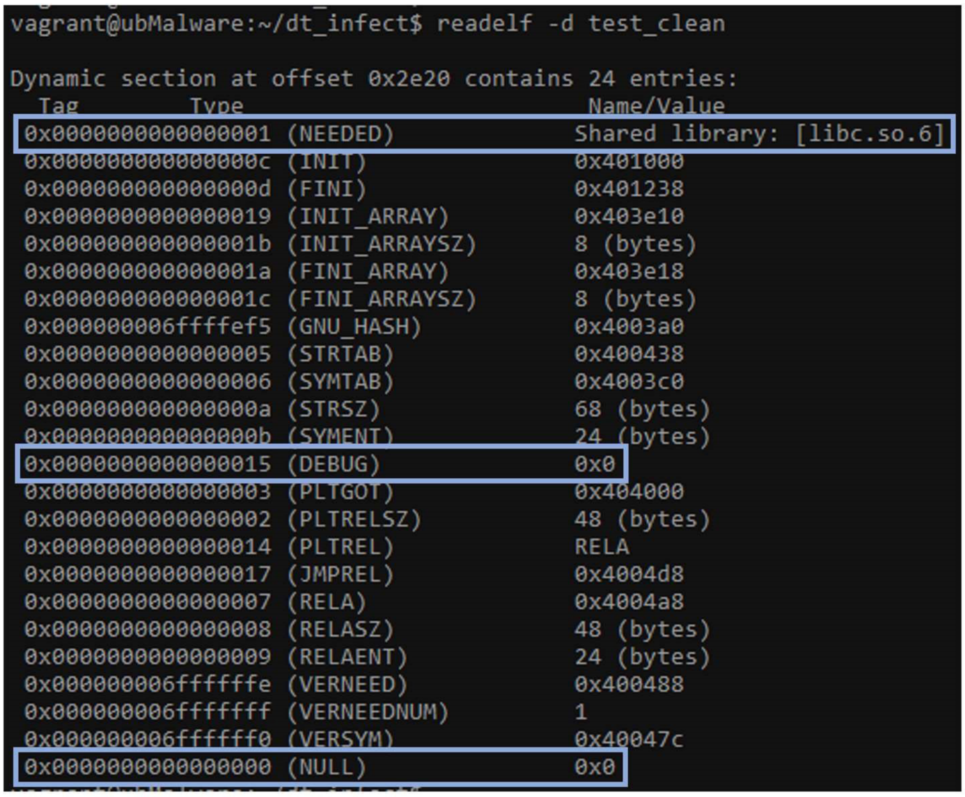
Two ELFN_DYN entry types that can be safely overwritten include:

- DT_NULL – marks the end of the dynamic array
- DT_DEBUG – used for debugging purposes.

The steps required are:

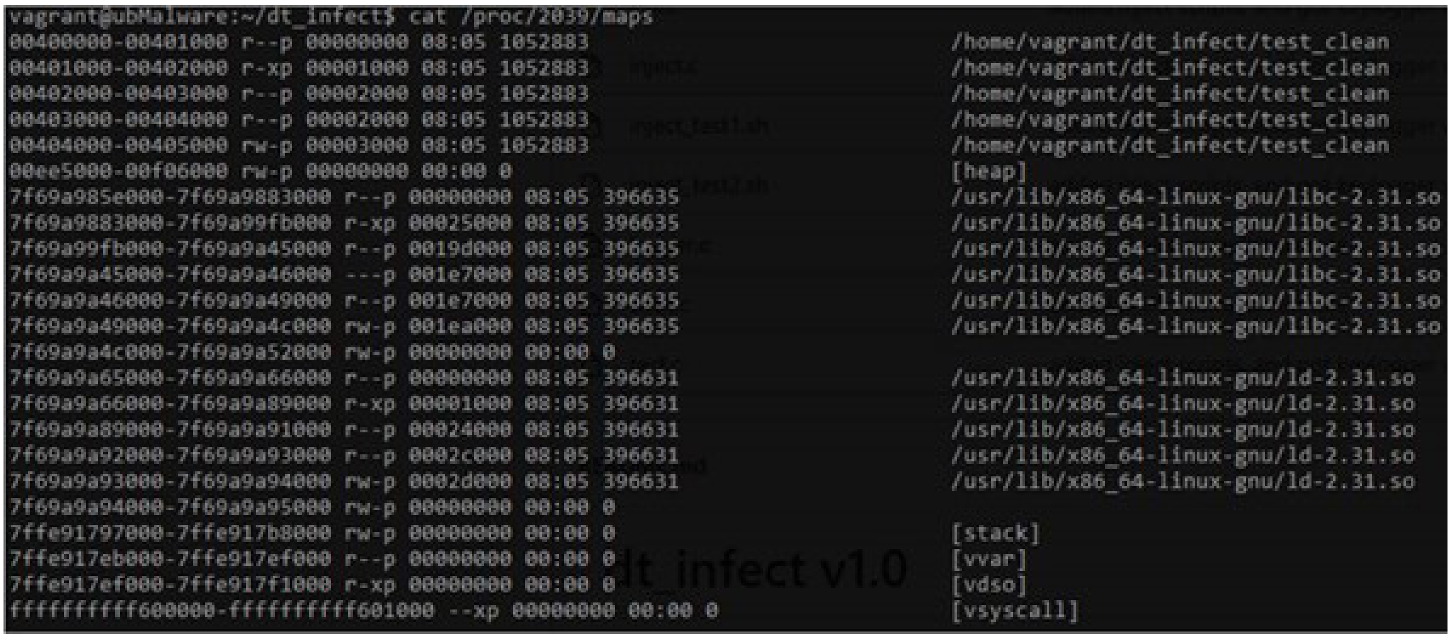
1. In the *.dynamic* section overwrite the DT_DEBUG/DT_NULL entry with a new DT_NEEDED entry for the module one wishes to inject. Alternatively, add a new DT_NEEDED entry to the end of the *.dynamic* section (if space permits).
2. Modify the dynamic string table to include the name of the new malicious library. (This requires a text padding infection unless there is already free space to append to.)
3. Place the shared object into the standard library path, e.g. */lib/x86_64-linux-gnu/*. (Optionally, configure a new LD_LIBRARY_PATH or LD_CONFIG to include the path to the malicious shared object.)
4. Fix up the GOT entries of the process with the function addresses of the malicious library, to achieve hooking of the existing loaded SO functions.

Figure 8 shows a clean dynamic section for the *test* binary (renamed to *test_clean*) prior to infection. Notice the existence of both DT_DEBUG and DT_NULL ELFN_DYN entries and that the only DT_NEEDED entry is listed first within the *.dynamic* section. Figure 9 represents the process mappings of *test_clean* once launched as a process. As expected, it contains mappings for the binary itself *test_clean*, libc *libc-2.31.so* and the dynamic linker *ld-2.31.so.*



```
vagrant@ubMalware:~/dt_infect$ readelf -d test_clean

Dynamic section at offset 0x2e20 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x401000
 0x000000000000000d (FINI)               0x401238
 0x0000000000000019 (INIT_ARRAY)         0x403e10
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x403e18
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x4003a0
 0x0000000000000005 (STRTAB)             0x400438
 0x0000000000000006 (SYMTAB)             0x4003c0
 0x000000000000000a (STRSZ)              68 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x404000
 0x0000000000000002 (PLTRELSZ)           48 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4004d8
 0x0000000000000007 (RELA)               0x4004a8
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400488
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x40047c
 0x0000000000000000 (NULL)               0x0
```

Figure 8: Clean dynamic section for test binary.



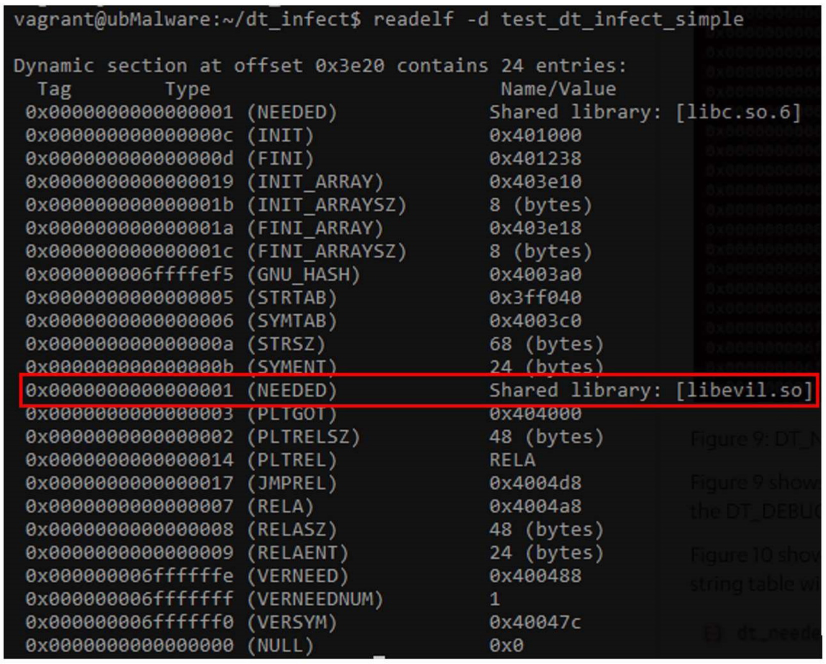
```
vagrant@ubMalware:~/dt_infect$ cat /proc/2039/maps
00400000-00401000 r--p 00000000 08:05 1052883                            /home/vagrant/dt_infect/test_clean
00401000-00402000 r-xp 00001000 08:05 1052883                            /home/vagrant/dt_infect/test_clean
00402000-00403000 r--p 00002000 08:05 1052883                            /home/vagrant/dt_infect/test_clean
00403000-00404000 r--p 00002000 08:05 1052883                            /home/vagrant/dt_infect/test_clean
00404000-00405000 rw-p 00003000 08:05 1052883                            /home/vagrant/dt_infect/test_clean
00ee5000-00f06000 rw-p 00000000 00:00 0                                  [heap]
7f69a985e000-7f69a9883000 r--p 00000000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f69a9883000-7f69a99fb000 r-xp 00025000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f69a99fb000-7f69a9a45000 r--p 0019d000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f69a9a45000-7f69a9a46000 ---p 001e7000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f69a9a46000-7f69a9a49000 r--p 001e7000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f69a9a49000-7f69a9a4c000 rw-p 001ea000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f69a9a4c000-7f69a9a52000 rw-p 00000000 00:00 0
7f69a9a65000-7f69a9a66000 r--p 00000000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f69a9a66000-7f69a9a89000 r-xp 00001000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f69a9a89000-7f69a9a91000 r--p 00024000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f69a9a92000-7f69a9a93000 r--p 0002c000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f69a9a93000-7f69a9a94000 rw-p 0002d000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f69a9a94000-7f69a9a95000 rw-p 00000000 00:00 0
7ffe91797000-7ffe917b8000 rw-p 00000000 00:00 0                          [stack]
7ffe917eb000-7ffe917ef000 r--p 00000000 00:00 0                          [vvar]
7ffe917ef000-7ffe917f1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Figure 9: Clean process mapping for test binary.

Figure 10 shows the *.dynamic* section of the same binary as before but this time it has been infected using the *dt_infect* [3] shared library injector and renamed as *test_dt_infect_simple*. Immediately one can spot two obvious changes:

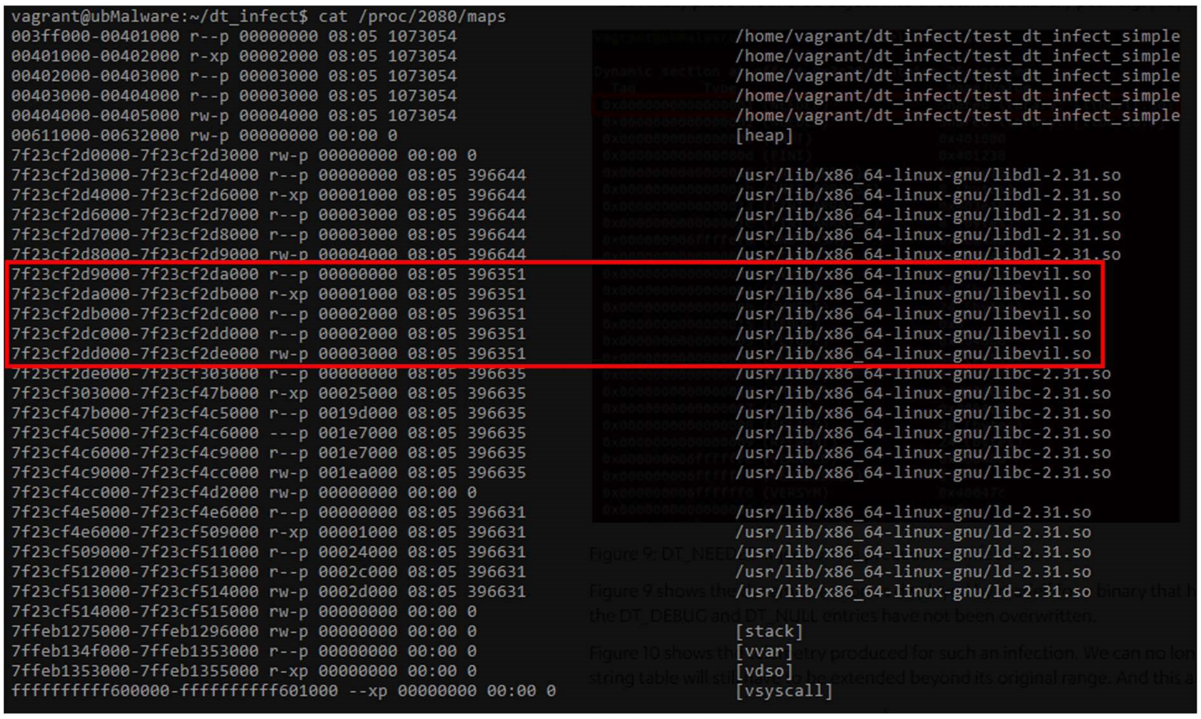
1. The disappearance of the DT_DEBUG entry.
2. An extra DT_NEEDED entry for *libevil.so* in its place.

Figure 11 shows that the process mappings have also changed, with *libevil.so* now loaded into the process address space.



```
vagrant@ubMalware:~/dt_infect$ readelf -d test_dt_infect_simple

Dynamic section at offset 0x3e20 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x401000
 0x000000000000000d (FINI)               0x401238
 0x0000000000000019 (INIT_ARRAY)         0x403e10
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x403e18
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x4003a0
 0x0000000000000005 (STRTAB)             0x3ff040
 0x0000000000000006 (SYMTAB)             0x4003c0
 0x000000000000000a (STRSZ)              68 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000001 (NEEDED)             Shared library: [libevil.so]
 0x0000000000000003 (PLTGOT)             0x404000
 0x0000000000000002 (PLTRELSZ)           48 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4004d8
 0x0000000000000007 (RELA)               0x4004a8
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400488
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x40047c
 0x0000000000000000 (NULL)               0x0
```

Figure 10: Infected dynamic section for test binary using a DT_NEEDED overwrite.



```
vagrant@ubMalware:~/dt_infect$ cat /proc/2080/maps
003ff000-00401000 r--p 00000000 08:05 1073054                            /home/vagrant/dt_infect/test_dt_infect_simple
00401000-00402000 r-xp 00002000 08:05 1073054                            /home/vagrant/dt_infect/test_dt_infect_simple
00402000-00403000 r--p 00003000 08:05 1073054                            /home/vagrant/dt_infect/test_dt_infect_simple
00403000-00404000 r--p 00003000 08:05 1073054                            /home/vagrant/dt_infect/test_dt_infect_simple
00404000-00405000 rw-p 00004000 08:05 1073054                            /home/vagrant/dt_infect/test_dt_infect_simple
00611000-00632000 rw-p 00000000 00:00 0                                  [heap]
7f23cf2d0000-7f23cf2d3000 rw-p 00000000 00:00 0
7f23cf2d3000-7f23cf2d4000 r--p 00000000 08:05 396644                     /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7f23cf2d4000-7f23cf2d6000 r-xp 00001000 08:05 396644                     /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7f23cf2d6000-7f23cf2d7000 r--p 00003000 08:05 396644                     /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7f23cf2d7000-7f23cf2d8000 r--p 00003000 08:05 396644                     /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7f23cf2d8000-7f23cf2d9000 rw-p 00004000 08:05 396644                     /usr/lib/x86_64-linux-gnu/libdl-2.31.so
7f23cf2d9000-7f23cf2da000 r--p 00000000 08:05 396351                     /usr/lib/x86_64-linux-gnu/libevil.so
7f23cf2da000-7f23cf2db000 r-xp 00001000 08:05 396351                     /usr/lib/x86_64-linux-gnu/libevil.so
7f23cf2db000-7f23cf2dc000 r--p 00002000 08:05 396351                     /usr/lib/x86_64-linux-gnu/libevil.so
7f23cf2dc000-7f23cf2dd000 r--p 00002000 08:05 396351                     /usr/lib/x86_64-linux-gnu/libevil.so
7f23cf2dd000-7f23cf2de000 rw-p 00003000 08:05 396351                     /usr/lib/x86_64-linux-gnu/libevil.so
7f23cf2de000-7f23cf303000 r--p 00000000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23cf303000-7f23cf47b000 r-xp 00025000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23cf47b000-7f23cf4c5000 r--p 0019d000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23cf4c5000-7f23cf4c6000 ---p 001e7000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23cf4c6000-7f23cf4c9000 r--p 001e7000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23cf4c9000-7f23cf4cc000 rw-p 001ea000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f23cf4cc000-7f23cf4d2000 rw-p 00000000 00:00 0
7f23cf4e5000-7f23cf4e6000 r--p 00000000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f23cf4e6000-7f23cf509000 r-xp 00001000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f23cf509000-7f23cf511000 r--p 00024000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f23cf512000-7f23cf513000 r--p 0002c000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f23cf513000-7f23cf514000 rw-p 0002d000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f23cf514000-7f23cf515000 rw-p 00000000 00:00 0
7ffeb1275000-7ffeb1296000 rw-p 00000000 00:00 0                          [stack]
7ffeb134f000-7ffeb1353000 r--p 00000000 00:00 0                          [vvar]
7ffeb1353000-7ffeb1355000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

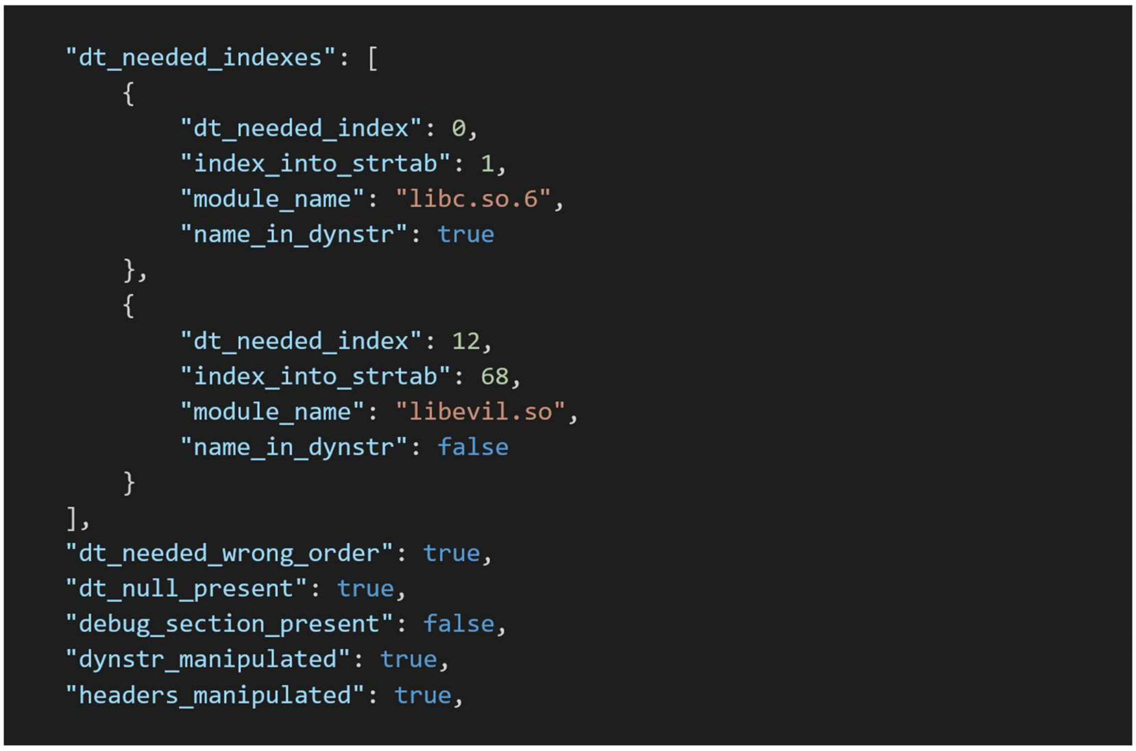
Figure 11: Infected process mappings for test binary.

Detection methods

One major drawback to using this method (from a stealth perspective) is that the object linker will never produce a binary that has its DT_NEEDED entries appearing in a non-sequential order within the dynamic section. Notice from Figure 10 that the shared library *libevil.so* does not appear immediately before/after the other DT_NEEDED entry *libc.so.6*, thus indicating it has been manually inserted after object linking.

Another drawback is that the dynamic string table (DT_SYMTAB) section (a null terminated array of library names) also needs to be appended to include the name of any additional shared object(s). This modification will introduce mix-ups between the true size and stated size (DT_STRSZ) in the *.dynamic* section. In many cases appending the name of the injected shared object is not possible without first making space (this is because compilers make efficient use of data so as not to introduce unnecessary padding into binaries). This can often only be achieved by relocating existing sections elsewhere in the process address space (such as with a reverse text padding infection), which in turn introduces more discrepancies between metadata fields and further irregularities when comparing the process image to that of a typical process.

Figure 12 shows the raw telemetry from the memory scanning tool *ELFieScanner* [4] when run against an infected process. DT_NEEDED entries within the dt_needed_indexes array should start with a dt_needed_index value of 0 and increment by one for each additional shared object. This rings true for the first DT_NEEDED entry, *libc.so.6*, however the second entry, *libevil.so*, appears at a dt_needed_index of 12 rather than 2 – indicating it was not linked using *Linux*'s object linker. In addition, the name reference to *libevil.so* appears outside of the dynamic string table, i.e. name_in_dynstr == false, thus it has been manually patched.



```
"dt_needed_indexes": [
    {
        "dt_needed_index": 0,
        "index_into_strtab": 1,
        "module_name": "libc.so.6",
        "name_in_dynstr": true
    },
    {
        "dt_needed_index": 12,
        "index_into_strtab": 68,
        "module_name": "libevil.so",
        "name_in_dynstr": false
    }
],
"dt_needed_wrong_order": true,
"dt_null_present": true,
"debug_section_present": false,
"dynstr_manipulated": true,
"headers_manipulated": true,
```

Figure 12: Telemetry from DT_NEEDED overwrite infection.

The dt_needed_index array provides enough evidence to confidently say the binary was manually patched with an additional shared library. However, the telemetry produces yet more evidence to suggest a DT_NEEDED infection has occurred. The additional indicators of use to an analyst include:

- debug_section_present == false. An attacker can overwrite the DT_DEBUG entry with a DT_NEEDED one. All GNU compiled binaries will have this section present, so the lack of a DT_DEBUG entry is highly suspicious and signifies manual manipulation of the *.dynamic* section.
- dt_needed_wrong_order == true. This is set to true when DT_NEEDED entries do not appear in sequential order, indicating a behaviour unlike that of the object loader which is responsible for generating the *.dynamic* section. The result of this will cause the dt_needed_indexes array to be populated in the telemetry.
- dynstr_manipulated == true. This indicates that a shared object name within one or more of the DT_NEEDED entries points to a region outside the initial confines of the dynamic string table, indicating it has been manually modified to include additional shared object names. To identify exactly which DT_NEEDED entries sit outside the dynamic string table refer to the dt_needed_index->name_in_dynstr field.
- headers_manipulated == true. Under normal circumstances the program headers will start immediately after the executable headers in memory. The *dt_infect* tool relocates the program headers, freeing up enough space to modify the *.dynamic* section and dynamic string table, which is detected as manipulation of the headers.
- dt_null_present. As with DT_DEBUG, the DT_NULL entry in the *.dynamic* section can also be overwritten with a DT_NEEDED entry. A DT_NULL entry should always be present, and its absence is a very strong indication of foul play. In this scenario DT_NULL has *not* been modified; however, one should always look to see if dt_null_present == false.

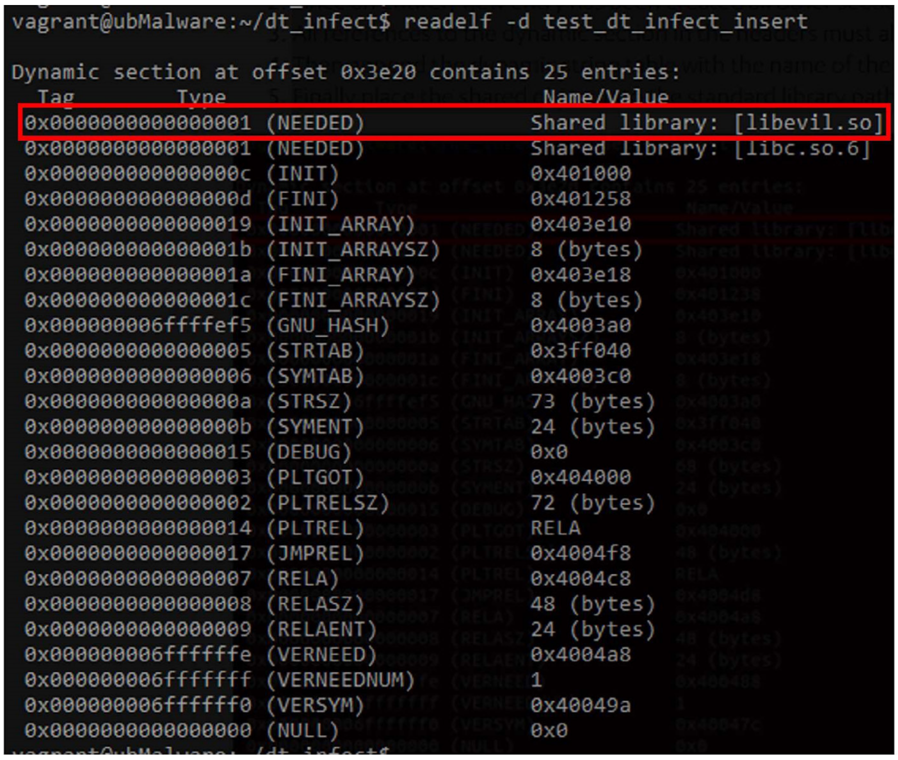
Method 2: Inserting a new entry

This method of infection is more complex, though in return it provides the added benefit of having the injected shared object loaded first and thus its functions will take precedence over any other DT_NEEDED entry, which also negates the need to manually fix up any GOT addresses.

The steps required are:

1. In the *.dynamic* section create a whole new DT_NEEDED entry for the shared object one wishes to inject, placing it as the first entry in the section. (This is to ensure its precedence and function resolution is higher than any other DT_NEEDED entry.)
2. Since an entirely new entry has been created, all other entries need to be shifted forward to allow space.
3. All references to the *.dynamic* section in the headers must also be updated to reflect these changes.
4. Then append the dynamic string table with the name of the malicious shared object. (This requires a text padding infection.)
5. Finally, place the shared object into the standard library path, e.g. */lib/x86_64-linux-gnu/*. (Optionally, configure a new LD_LIBRARY_PATH or LD_CONFIG to point towards the malicious library.)

Figure 13 shows a *.dynamic* section of the same clean *test* binary as before, but this time it has been infected using the insertion method with *dt_infect* [3] and then renamed to *test_dt_infect_insert*. Unlike the previous method this time the *libevil.so* DT_NEEDED entry has been inserted first, thereby maintaining sequential order of DT_NEEDED entries starting at entry 0. In addition, both the DT_DEBUG and DT_NULL entries remain present.



```
vagrant@ubMalware:~/dt_infect$ readelf -d test_dt_infect_insert

Dynamic section at offset 0x3e20 contains 25 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libevil.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x401000
 0x000000000000000d (FINI)               0x401258
 0x0000000000000019 (INIT_ARRAY)         0x403e10
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x403e18
 0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x4003a0
 0x0000000000000005 (STRTAB)             0x3ff040
 0x0000000000000006 (SYMTAB)             0x4003c0
 0x000000000000000a (STRSZ)              73 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x404000
 0x0000000000000002 (PLTRELSZ)           72 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4004f8
 0x0000000000000007 (RELA)               0x4004c8
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x4004a8
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x40049a
 0x0000000000000000 (NULL)               0x0
```

Figure 13: Infected dynamic section for the test binary using the DT_NEEDED insertion technique.

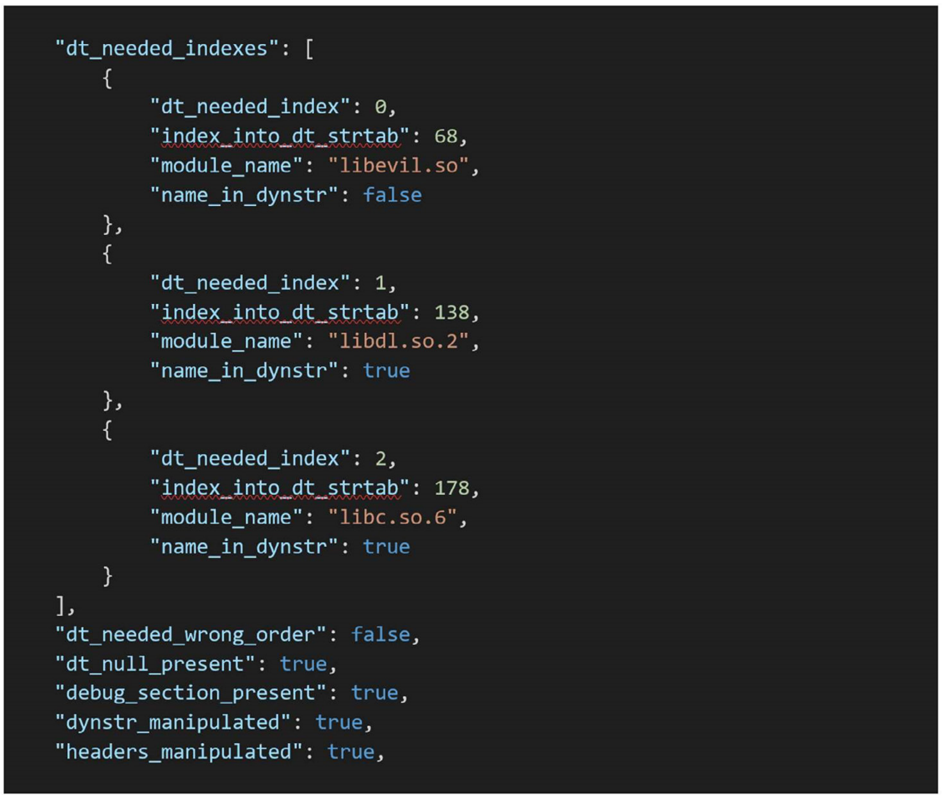
Detection methods

This method makes detection slightly trickier, since one can no longer rely on looking for a nonsequential DT_NEEDED order or a missing DT_NULL/DT_DEBUG entry in the *.dynamic* section. However, because a whole new module has been added, the dynamic string table will still have to be extended beyond its original range and headers still need be relocated to make space; all of which is still detectable.

Figure 14 shows the perspective of such an attack based on the raw telemetry of *ELFieScanner*. This time the key fields to monitor are:

- name_in_dynstr
- dynstr_manipulated
- headers_manipulated

Based on this one can deduce that *libevil.so* (module_name == libevil.so) has been injected into the process, since its string reference exists outside the original confines of the dynamic string table (name_in_dynstr == false). One can also see that the headers have been manipulated (headers manipulated == true), most likely to facilitate the rewriting of the dynamic string table at runtime (dynstr_manipulated == true) for the string *libevil.so*.



```
"dt_needed_indexes": [
    {
        "dt_needed_index": 0,
        "index_into_dt_strtab": 68,
        "module_name": "libevil.so",
        "name_in_dynstr": false
    },
    {
        "dt_needed_index": 1,
        "index_into_dt_strtab": 138,
        "module_name": "libdl.so.2",
        "name_in_dynstr": true
    },
    {
        "dt_needed_index": 2,
        "index_into_dt_strtab": 178,
        "module_name": "libc.so.6",
        "name_in_dynstr": true
    }
],
"dt_needed_wrong_order": false,
"dt_null_present": true,
"debug_section_present": true,
"dynstr_manipulated": true,
"headers_manipulated": true,
```

Figure 14: Telemetry from DT_NEEDED insertion infection.

TECHNIQUE LIMITATIONS FROM AN ATTACKING PERSPECTIVE

- This technique requires manual manipulation of a process binary prior to execution. Thus, it isn't applicable to currently running processes.
- This technique requires the malicious shared object to be present on disk (at least at process start time).

LD_PRELOAD INJECTION AND SYMBOL HOOKING

The LD_PRELOAD environmental variable is designed to be filled with a user-specified list of shared object files which will be loaded before all others when starting a process. The effect of this is that the LD_PRELOAD shared object functions will selectively override functions in other shared objects loaded via the traditional searching methods. Preloading shared objects is not exclusive to the LD_PRELOAD variable, it can also be achieved via:

- Invoking the '—preload' command-line options when invoking the dynamic linker directly.
- Modifying the */etc/ld.so.preload* file. This is a whitespace-separated list of ELF shared objects to be loaded before the program (this has a system-wide effect causing all newly launched processes to be affected).

The preload methods are handled in the following order:

1. LD_PRELOAD environmental variable.
2. '—preload' dynamic linker option. (Only used when invoking the dynamic linker directly.)
3. */etc/ld.so.preload* file.

Preloading can be used legitimately; it allows developers slightly more flexibility/portability with their code. This is because instead of having the rewrite their program to accommodate a new version of a function, they can instead ship the binary with the original shared object it was designed around, setting it to be preloaded in the process address space.

In addition, LD_PRELOAD can also be used legitimately to resolve debugging issues. For instance, some processes do not wish to be traced and will deploy anti-debug mechanisms to protect themselves. One such technique is to call the *ptrace()*

syscall with the PTRACE_TRACEME request argument; this instructs a process to try to trace itself (via its own parent process). If this fails it means it is currently being traced elsewhere; the process can then react accordingly to protect itself from being debugged. Employing this mechanism would hamper the GNU debugger's (*gdb*) effectiveness since it relies on *ptrace()* to attach to a process. Not being able to debug a process may significantly hamper an engineer's ability to conduct root cause analysis.

Luckily, preloading can be used to circumvent this protection. For instance, one could launch the target process with a LD_PRELOAD library containing a reimplemented version of the *ptrace()* syscall that ignores the PTRACE_TRACEME request, thus enabling *gdb* to attach to the target process and start a debugging session.

Unfortunately, attackers have also stumbled upon the use of LD_PRELOAD to orchestrate attacks. This is because LD_PRELOAD is essentially a search order hijacking technique. As mentioned previously, LD_PRELOAD shared objects take precedence over shared objects loaded using the default search methods.

This has two major benefits to an attacker:

1. When the dynamic linker looks to resolve an ELF binary's imports it will first search in the LD_PRELOAD environmental variable for shared objects. Symbols that match imports here will take precedence over any other versions from shared objects loaded via traditional means. For example, should the *execv()* symbol appear in a LD_PRELOAD library as well as *libc.so.6* only the LD_PRELOAD version will be resolved, essentially acting as a user-level hook. An attacker can therefore define their own version of *execv()* in their preloaded library and can rely on this be executed instead of the *libc.so.6* version.
2. LD_PRELOAD libraries are loaded first and mapped into memory by *dlopen()* and *mmap()* respectively. This means that any constructor code within the shared object is also executed before any other shared objects. Attackers can insert their malicious code into a constructor function without having to rely on it being called from within the *.text* segmentof the individual process.

Because of its ease of implementation the LD_PRELOAD injection method has been seen used in a wide array of malware samples and rootkits including:

- Azazel rootkit [5]
- BEURK rootkit [6]
- Jynx rootkit [7]
- Vlany rootkit [8]
- Umbreon rootkit [9]
- HiddenWasp [10]

LIMITATIONS

This technique is predicated on LD_PRELOAD/*ld.preload.conf* being in place prior to the target process running. It will not work on existing processes since they will already have their shared objects loaded in memory by the dynamic linker.

DETECTION METHODS

We have already seen examples of how the command line of a process can be monitored for the use of LD_PRELOAD, its environmental variables including LD_LIBRARY_PATH can be read for custom paths, and the */etc/ld.so.conf* and */etc/ld.so.conf.d/*conf* files can be checked for modification (as discussed in *AT&T*'s post on 'Hunting for Linux library injection with Osquery' [11]). In some cases collecting just the names and paths may be a sufficient starting point to then manually investigate an anomalous preloaded object.

However, as the number of hosts begin to scale and the number of benign preloading instances rise, it will become increasingly difficult to perform manual analysis on every shared object that sticks out to determine its legitimacy. One then starts to encounter questions such as: 'Should I pull every preloaded object?', 'What if there are multiple preloaded objects per process?' and 'What if the preloaded objects have been deleted from disk?' Is it then worth the analyst's time to create a core dump, extract the preloaded module(s), reverse them and discover their function all within the time-sensitive nature of an enterprise investigation?

Instead, a more robust detection approach is needed so that analysis can be performed at the telemetry level, limiting the need for manual intervention in order to asses a threat (in the majority of cases).

A proposed solution is to use *ELFieScanner*. It detects not only the presence of preloading across running processes but also every hooked function name, the path to the preloaded object, its location within that object and the original shared object it was hooked from. Using this, analysts can then asses the functionality (and in turn legitimacy) of preloaded libraries at scale without having to get their hands dirty performing manual binary analysis. The following steps detail how the tool goes about generating this data.

Step 1: Identifying all the imports used by a process

The diagram in Figure 15 is a heavily simplified version of ELF binary internals designed to demonstrate the pathway taken to curate a list of imported functions used by a process. The relevant steps are detailed below.

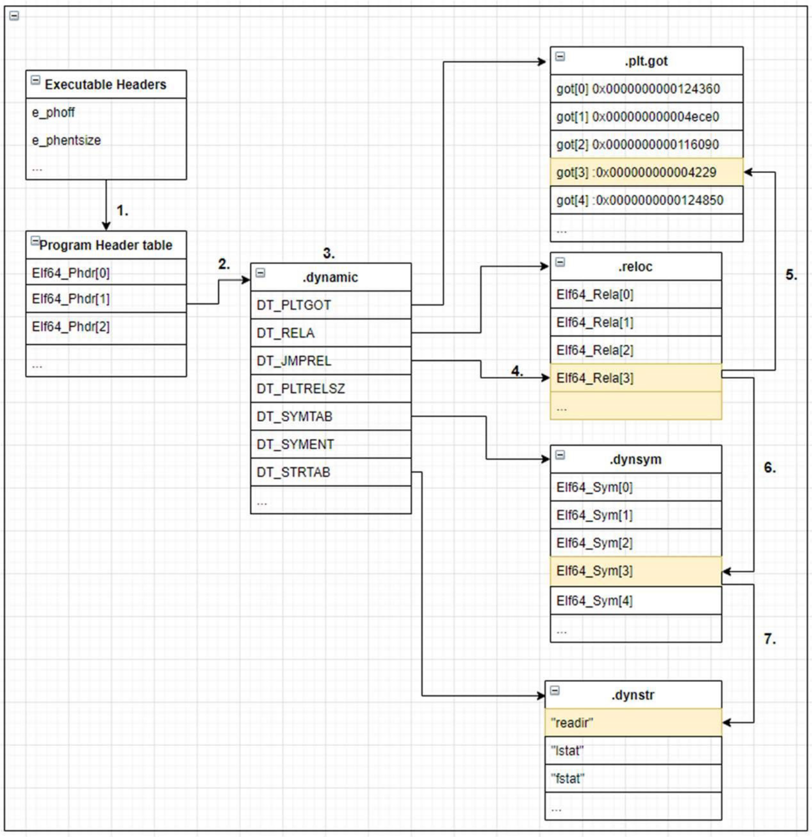


Figure 15: Obtaining import symbols from an ELF process.

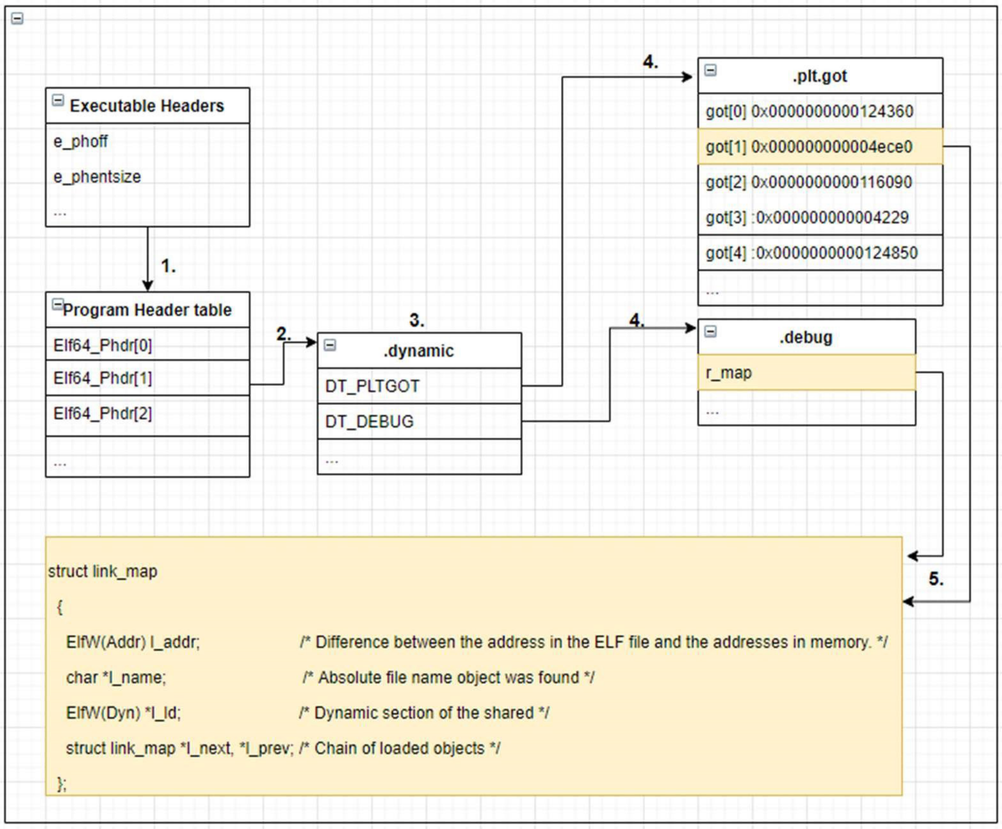
Sub-steps:

1. Inside the ELF executable header are the *e_phoff* and *e_phentsize* fields. They provide the offset and size of the program header table, respectively.
2. Using the *e_phoff* and *e_phentsize* fields, enumerate the program headers for an entry of type PT_DYNAMIC. This will point to the dynamic segment in memory.
3. The dynamic segment is a 1:1 mapping of the *.dynamic* section, which contains pointers to the following sections:
   - Global offset table (DT_PLTGOT)
   - Relocation table (DT_RELA)
   - Dynamic symbol table (DT_SYMTAB)
   - Dynamic string table (DT_STRTAB)
4. The DT_JMPREL entry in the *.dynamic* section points to the first entry in the relocation table responsible for imports. The DT_PLTRELSZ entry contains the total size in bytes of the relocation entries associated with the imported symbols. Together they can be used to identify every relocation table entry (Elf64_Rela) responsible for an imported symbol.
5. Each ELF64_Rela entry will contain:
   - The address of the GOT entry it is responsible for.
   - The offset of the dynamic symbol table entry it relates to (Elf64_Sym).
6. Within each ELF64_Sym entry exists an offset within the dynamic string table. Use this offset to associate an import entry with a function (symbol) name.
7. Using this technique one can successfully extract all the dynamically resolved imports (i.e. imported function names) used by a particular process.

Step 2: Establish a list of all the loaded libraries and their base addresses in the virtual address space

Later steps involve shared object symbol resolution, which is predicated on knowing the base addresses in memory of all shared objects. The most common way to establish the base address would be to read the */proc/<pid>/maps* file for the pid of the process. However, because this technique investigates LD_PRELOAD hooking it is entirely possible that an attacker may have hooked the *fread()* syscall in order to present falsified output to the user when reading the */proc/<pid>/maps* file, which could hide any preloaded libraries used by the attacker and thus provide one with an incomplete list of shared object base addresses.

As a result, the memory scanner uses a hooking-resistant mechanism to establish the correct base addresses for all the shared objects loaded into memory. Figure 16 and the following steps demonstrate how this is achieved.



```
struct link_map
 {
   ElfW(Addr) l_addr;                /* Difference between the address in the ELF file and the addresses in memory. */
   char *l_name;                     /* Absolute file name object was found */
   ElfW(Dyn) *l_ld;                  /* Dynamic section of the shared */
   struct link_map *l_next, *l_prev; /* Chain of loaded objects */
 };
```

Figure 16: Obtaining base addresses for loaded shared objects in memory.

Sub-steps:

1. Inside the ELF executable header are the *e_phoff* and *e_phentsize* fields. They provide the offset and size of the program header table, respectively.
2. One can then enumerate the program headers for an entry of PT_DYNAMIC, this will point to the dynamic segment in memory.
3. The dynamic segment is a 1:1 mapping of the *.dynamic* section, which contains pointers to the following sections:
    - Global offset table (DT_PLTGOT)
    - Debug section (DT_DEBUG)
4. Using one of two methods one can ascertain a pointer to a linked list (*link_map*) that contains shared object information:
    - Using DT_DEBUG – this will point to a *r_debug* struct that contains the field *r_map* which points to the *link_map* linked list.
    - Using the GOT – the GOT[1] entry will always point to the *link_map* linked list. This can be used as a backup incase DT_DEBUG has been overwritten or isn't present.
5. The *link_map* structure is created dynamically by the runtime linker and is a cyclical chain of structs in memory. Each struct will contain pointers to the previous and following structs using the *l_next* and *l_prev* fields respectively. This can be used to iterate through the linked list and extract the loaded base address, *l_addr*, and full path name, *l_name*, of each individual shared object in memory.

Using this method provides a more reliable and spoofing-resistant way to enumerate shared objects rather than just reading the */proc* file system. Techniques discussed in Part 2 will also reveal how this method can also be used to detect hidden shared objects through cross-referencing of data sources.

Step 3: Identify the names of the preloaded shared objects

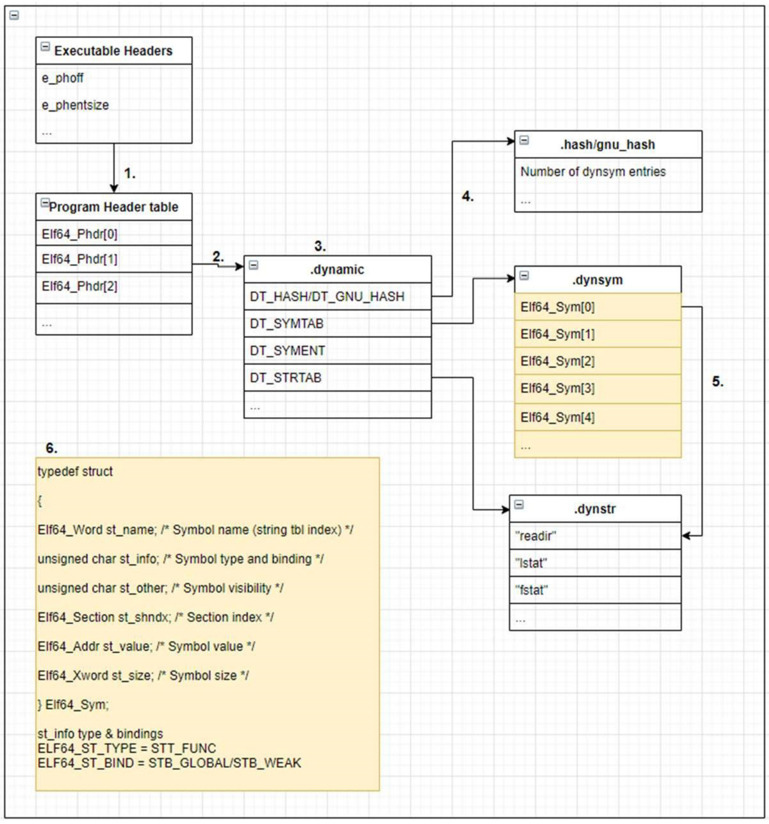
Using the following data sources one can curate a list of the shared objects that may have been successfully preloaded:

- Read the stack to identify the LD_PRELAOD environmental variable and whether this has been populated.
- Read the */etc/ld.so.preload* file to identify any additional preloaded libraries.

If preloaded libraries have been found, it doesn't necessarily mean they were successfully preloaded into the memory address space of the process. This can be validated against the *link_map* collected from the previous step.

Step 4: Resolve full symbol table from preloaded libraries

Before any potential hooks can be identified one must first generate a full list of every globally visible symbol from the preloaded shared object(s). The steps to achieve this are outlined in Figure 17.



```
typedef struct
{
Elf64_Word st_name; /* Symbol name (string tbl index) */
unsigned char st_info; /* Symbol type and binding */
unsigned char st_other; /* Symbol visibility */
Elf64_Section st_shndx; /* Section index */
Elf64_Addr st_value; /* Symbol value */
Elf64_Xword st_size; /* Symbol size */
} Elf64_Sym;

st_info type & bindings
ELF64_ST_TYPE = STT_FUNC
ELF64_ST_BIND = STB_GLOBAL/STB_WEAK
```

Figure 17: Resolving the full symbol table from shared objects.

Steps:

1. Inside the ELF executable header are the *e_phoff* and *e_phentsize* fields. They provide the offset and size of the program header table, respectively.
2. One can then enumerate the program headers for an entry of PT_DYNAMIC, which will point to the dynamic segment in memory.
3. The dynamic segment is a 1:1 mapping of the *.dynamic* section, which contains pointers to the following sections:
    - Hash table / GNU Hash table (DT_HASH / DT_GNU_HASH)
    - Dynamic symbol table (DT_SYMTAB)
    - Dynamic string table (DT_STRTAB)
4. For whatever reason, the dynamic section doesn't provide an entry for the total number of symbols. This is where hash tables come in handy. Either the *.hash* or *.gnu.hash* section will be present, but not both of them together. Hash tables are essentially a way to optimize symbol lookups performed by the dynamic linker. The standard hash table

.hash contains a field that equates to the number of hashes, thus symbol table entries. However, for *.gnu.hash* this isn't present and instead it needs to be dynamically rebuilt to establish the number of hashes, and in turn to determine the number of symbol table entries.

5. With the number of the symbol table entries calculated the names of the symbols can be resolved for each entry using their string table offset.
6. Only symbols of type STT_FUNC and Binding of STB_GLOBAL/STB_WEAK are collected, since these are associated with functions that can be exported from loaded shared objects to the main process executable.

Step 5: Compare imported symbols with preloaded visible symbols

Using the list of imported symbols from step 1, compare these against the list of global symbols from step 4. Any symbol matches will indicate that a function has been hooked by a preloaded library.

Step 6: Resolve full symbol table from non-preloaded libraries to provide context

Using the same methodology as in step 4, resolve the exported symbols from the shared objects that have been loaded without the use of any preload mechanisms and cross-reference these symbols against the hooked imports. This will provide greater context by identifying which original shared objects the symbols have been hooked from. This may aid further analysis, especially if the hook is of an inline type.

DETECTION IN RAW TELEMETRY

To demonstrate how this all comes together the BEURK rootkit has been used to infect *Ubuntu 18.04*. 'BEURK is an userland preload rootkit for GNU/*Linux*, heavily focused around anti-debugging and anti-detection' [6]. It is designed to evade detection by traditional forensic methods and uses hooking of specific functions to achieve this. Unlike some other rootkits BEURK deploys system-wide preloading by making use of the */etc/ld.so.preload* file. This means that every new process launched will be preloaded with the shared object(s) stated in *ld.so.preload*.

Figure 18 shows part of the raw detection telemetry produced by the *ELFieScanner* when scanning the infected machine.

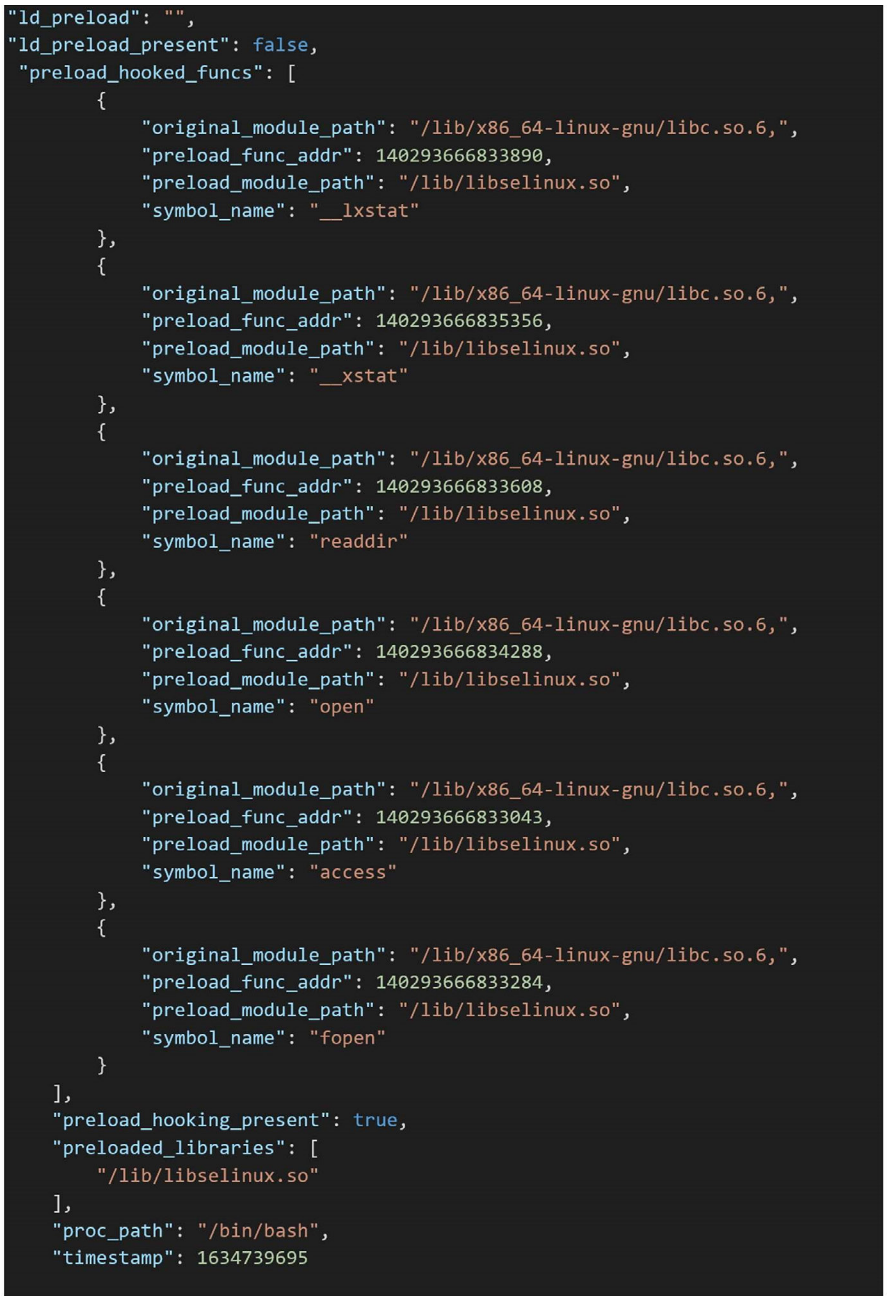
The following field descriptions can be used to help an analyst interpret the results:

- ld_preload_present – if this equals true it indicates that the LD_PRELOAD variable has been set. LD_PRELOAD can be used to preload one or more shared objects.
- ld_preload – this will contain the paths of any shared objects preloaded using the LD_PRELOAD environmental variable.
- preloaded_libraries – this contains a list of every successfully preloaded library from both */etc/ld.so.preload* and LD_PRELAOD.
- preload_hooking_present – if set to true then the preloaded libraries have successfully hooked one or more of the process's imports.
- preload_hooked_funcs – this is populated when preload_hooking_present is set to true. It contains an entry for each hook. Each entry contains the fields:
  - symbol_name – the name of the symbol that has been hooked.
  - preload_module_path – the preloaded shared object responsible for the hook.
  - preload_func_addr – the virtual address of the symbol in memory.
  - original_module_path – the shared object containing the original unhooked version of the symbol.

Based on the above descriptions and Figure 18 one can deduce that:

- The shared object */lib/libselinux.so* has been preloaded into the bash process (preloaded_libraries == /lib/libselinux.so) in a system-wide manner using */etc/ld.so.preload* since LD_PRELOAD has not been used (ld_preload == false).
- */lib/libselinux.so* has successfully hooked one or more of the /bin/bash process's imports (preload_hooking_present == true).
- The individual import hooks detected are for the functions *__lxstat()*, *__xstat()*, *readir()*, *open()*, *access()* and *fopen()* (preload_hooked_funcs->symbol_name), all of which originally existed in */lib/x86_64-linux-gnu/libc.so.6* (preload_ hooked_funcs->original_module_path).

Any additional analysis can then be performed by extracting the hook itself using the preload function address (preload_ hooked_funcs->preload_func_addr) to manually collect the bytes and determine what the hook does.



```
"ld_preload": "",
"ld_preload_present": false,
 "preload_hooked_funcs": [
        {
            "original_module_path": "/lib/x86_64-linux-gnu/libc.so.6,",
            "preload_func_addr": 140293666833890,
            "preload_module_path": "/lib/libselinux.so",
            "symbol_name": "__lxstat"
        },
        {
            "original_module_path": "/lib/x86_64-linux-gnu/libc.so.6,",
            "preload_func_addr": 140293666835356,
            "preload_module_path": "/lib/libselinux.so",
            "symbol_name": "__xstat"
        },
        {
            "original_module_path": "/lib/x86_64-linux-gnu/libc.so.6,",
            "preload_func_addr": 140293666833608,
            "preload_module_path": "/lib/libselinux.so",
            "symbol_name": "readdir"
        },
        {
            "original_module_path": "/lib/x86_64-linux-gnu/libc.so.6,",
            "preload_func_addr": 140293666834288,
            "preload_module_path": "/lib/libselinux.so",
            "symbol_name": "open"
        },
        {
            "original_module_path": "/lib/x86_64-linux-gnu/libc.so.6,",
            "preload_func_addr": 140293666833043,
            "preload_module_path": "/lib/libselinux.so",
            "symbol_name": "access"
        },
        {
            "original_module_path": "/lib/x86_64-linux-gnu/libc.so.6,",
            "preload_func_addr": 140293666833284,
            "preload_module_path": "/lib/libselinux.so",
            "symbol_name": "fopen"
        }
    ],
    "preload_hooking_present": true,
    "preloaded_libraries": [
        "/lib/libselinux.so"
    ],
    "proc_path": "/bin/bash",
    "timestamp": 1634739695
```

Figure 18: Telemetry from BEURK preloading infection and hooking.

PART 2

Unlike the static techniques we discussed in Part 1, which required intervention by an attacker prior to execution, this part will focus on more dynamic methods that facilitate shared object injection into processes at runtime, such as:

- Direct use of *__libc_dlopen_mode()*
- Reflective Shared object injection.

In addition to this we look at why sometimes methods designed to hide a shared object can actually make detection easier.

EXISTING DETECTION STRATEGIES

To help with detection of shared object injection existing methods seem to rely on marrying up events such as:

- Setting ptrace scope to 0 in Yama security module.
- Monitoring for *ptrace()* syscalls that attach/write/get registers of another process.

And then combining these with surrounding telemetry to determine if injection has taken place. However, at scale this becomes more of a blunt tool. This is because:

- Legitimate processes such as browsers, debuggers and AV will often employ the same methods, thus generating a higher ratio of false positives to true positives.
- The detection methods do not target shared object injection directly, but rather injection as a whole, which then requires further interpretation by a skilled analyst.

In this part of the paper we will look to demonstrate some slightly more granular detection methods that can be used alongside broader detections to assist in triaging alerts.

__LIBC_DLOPEN_MODE() INJECTION

A rather easy and simple way to inject a shared object into a process would be to debug a victim process, force it to call a library function and have that do the heavily lifting, negating the need to manually perform image loads and relocations. There are two functions that can be used to load a library into a process on *Linux*:

- *dlopen()* from LIBDL (the dynamic linker). This is the legitimate way to request a shared object to be loaded by the dynamic linker at runtime.
- *__libc_dlopen_mode()*, which is LIBC's implementation.

Once loaded, a malicious library allows an attacker to execute code immediately (using a constructor function) and/or hook existing functions used by a process. But first the victim process must call either *__libc_dlopen_mode* or *dlopen()*.

One solution is to write an injector that will:

1. Attach to a victim process with LIBC already loaded and suspend its execution.
2. Resolve the address of *__libc_dlopen_mode()*/*dlopen()*.
3. Modify the instruction pointer to point at *__libc_dlopen_mode()*/*dlopen()*.
4. Replace registers (x64) or stack values (x86) with the 'shared object path' and 'mode' arguments.
5. Resume execution.
6. (Optionally) Once the library is loaded into a process it can then be deleted from disk, residing only in memory.

Refer to David Kaplan's post 'Linux code injection paint-by-numbers' [12] for an excellent walkthrough of how this is achieved programmatically.

Alternatively, instead of writing a custom injector an attacker could instead leverage GDB (the GNU debugger) [13] to attach the target process and force it to call *__libc_dlopen_mode()*/*dlopen()*, as seen in the simple one-liner below:

```
echo 'print __libc_dlopen_mode("/tmp/sample_library.so", 2)' | gdb -p <PID>
```

A more verbose breakdown of this technique us is described in 'Process Injection with GDB' [14].

When it comes to deciding which function to forcibly call the answer is almost always going to be *__libc_dlopen_mode()*. This is because it is exposed by LIBC which is loaded by 99% of processes, meaning it can reliably be resolved at runtime. On the contrary, it is rare to have a process expose *dlopen()* since it requires the dynamic linker to have been explicitly linked (using the -ldl GCC/G++ flag) at compile time, which is something that is not common practice.

In the following detection we have chosen not to focus on *dlopen()*. This is because, in addition to it being far less portable (thereby less attractive to attackers), it is also the legitimate way to load shared objects, thus monitoring calls will generate large amounts of benign data which isn't ideal when working at scale.

Detections

For this attack type we have split the detections into two categories:

- Detection of victim processes
- Detection of attacking/injector processes.

Detection of victim processes

Since the attack technique is predicated on forcing a victim process to call *__libc_dlopen_mode()* from GLIBC, an obvious solution is to directly monitor library calls.

There are a number of avenues to achieve this, probably the most well-known is by using the *ltrace* [15] program that comes bundled with most *Linux* distributions. Ltrace uses *ptrace()* syscalls to attach to processes and intercept function calls at a user-mode level. Every time a library function is hit it pauses the process to read the function arguments. In doing so this introduces a significant performance overhead. Also, it relies on *ptrace()*, which is susceptible to anti-debugging mechanisms which an attacker can use to prevent it from attaching to a target process in the first place. For these reasons it has been overlooked in favour of a better solution: *Uprobes*.

Uprobes, introduced since *Linux 4.5*, are the user-space counterpart to *Kprobes* [16]. They are a kernel feature that allow the monitoring of user-space processes and functions by inserting a fast breakpoint at the target instruction which then passes execution to a Uprobes handler. Originally designed to generate efficient performance metrics, their use case can be hijacked to develop real-time security detections. Simplistically put, think of Uprobes and Kprobes as the *Linux* equivalent of *Microsoft*'s Event Tracing for Windows (ETW). For more information as to how Uprobes work refer to the kernel documentation [17].

Prior to collecting trace data, a Uprobe must first be defined. This requires the user to know:

1. The full path of the shared object containing the function they wish to target.
2. The offset of that function within the object.
3. What registers (x64) or stack values (x86) one wishes to collect at the probe point, i.e. the function parameters.

Only then can a probe be placed in the correct location. Failing to provide the correct details will result in erroneous output.

Most Uprobes are unlikely to work across different builds. This is because there are multiple versions of GBLIC that have all been compiled separately on each individual host. What this means is that the offset to *__libc_dlopen_mode()* is unlikely to remain constant. In addition, there may be different function prototypes across different versions of GLIBC.

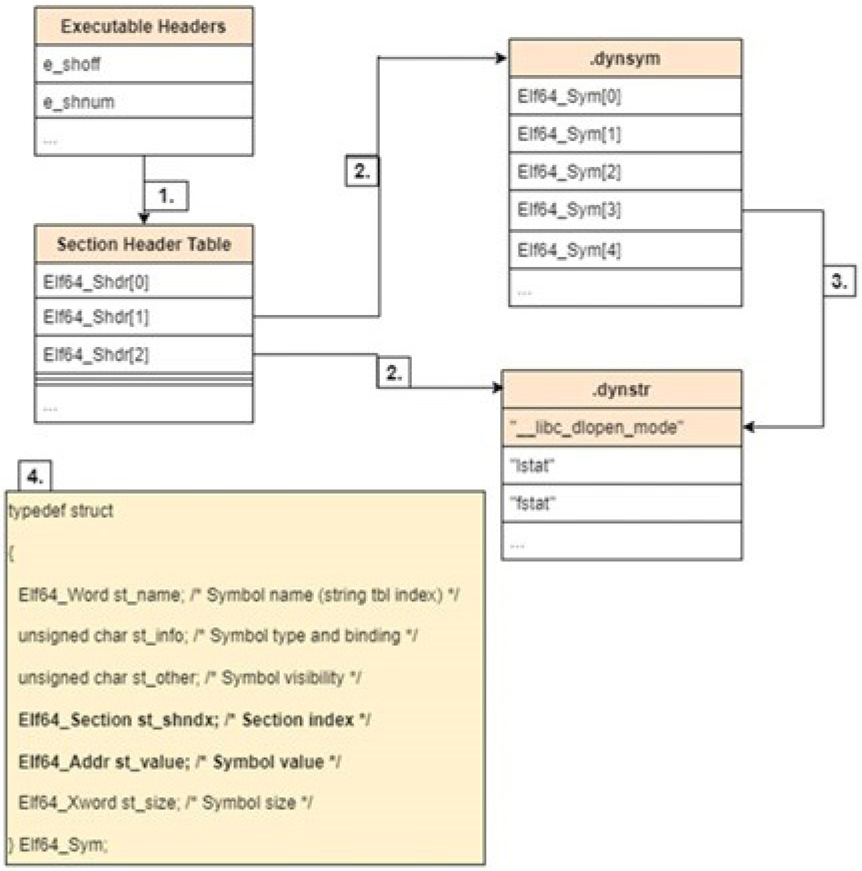
Calculating the offset to __libc_dlopen_mode

One way of calculating the offset of *__libc_dlopen_mode()* on each host is to the use 'nm' as part of the GNU Binutils:

```
nm -D /usr/lib/x86_64-linux-gnu/libc-2.32.so | grep __libc_dlopen_mode
```

In the absence of nm it can also be calculated programmatically. This is demonstrated in Figure 19 using the following steps:

1. Use the binary executable header to locate the section header table.
2. Use the section header table to locate the dynamic symbol table (.dynsym) and dynamic string table (.dynstr).
3. Enumerate each ELF64_Sym entry in .dynsym, collect the st_name field (which are offsets into .dynstr) and use this to calculate which symbol name corresponds with each entry.
4. Once the __libc_dlopen_mode symbol has been paired with an ELF64_Sym entry, read the st_value field to determine its file offset.



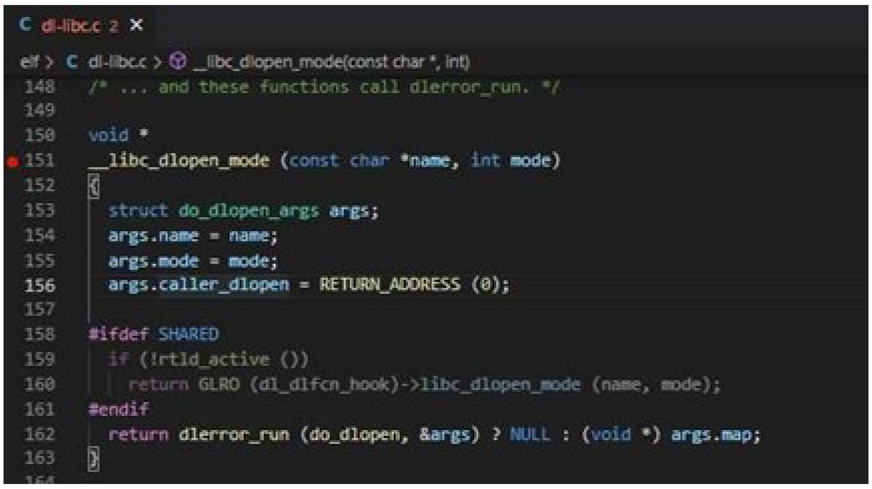
```
typedef struct
{
  Elf64_Word st_name; /* Symbol name (string tbl index) */
  unsigned char st_info; /* Symbol type and binding */
  unsigned char st_other; /* Symbol visibility */
  Elf64_Section st_shndx; /* Section index */
  Elf64_Addr st_value; /* Symbol value */
  Elf64_Xword st_size; /* Symbol size */
} Elf64_Sym;
```

Figure 19: Obtaining __libc_dlopen_mode from GLIBC using section headers.

Since this method uses the dynamic symbol table and dynamic string table to locate the correct offset it will work with both stripped and non-stripped binaries.

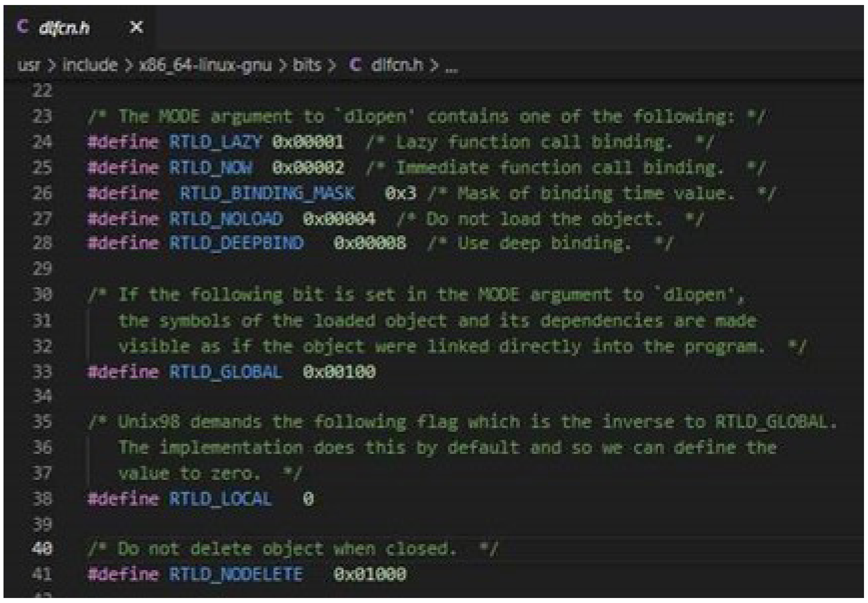
Determining the function parameters

Next, to establish the parameters supplied to *__libc_dlopen_mode()* and discover any variations between GLIBC versions look at the publicly available source code (Figure 20), which reveals that only two parameters are used: one for the path of the shared object to load (name) and the other specifying the loading method (mode). These are same parameters as *dlopen()*, and upon further investigation they use the same set of flags for the mode parameter (Figure 21).



```
C dl-libc.c 2 ×
elf > C dl-libc.c > __libc_dlopen_mode(const char *, int)
/* ... and these functions call dlerror_run. */

void *
__libc_dlopen_mode (const char *name, int mode)
{
  struct do_dlopen_args args;
  args.name = name;
  args.mode = mode;
  args.caller_dlopen = RETURN_ADDRESS (0);

#ifdef SHARED
  if (!rtld_active ())
    return GLRO (dl_dlfcn_hook)->libc_dlopen_mode (name, mode);
#endif
  return dlerror_run (do_dlopen, &args) ? NULL : (void *) args.map;
}
```

Figure 20: Function prototype for __libc_dlopen_mode.



```
C dlfcn.h ×
usr > include > x86_64-linux-gnu > bits > C dlfcn.h > ...

/* The MODE argument to `dlopen' contains one of the following: */
#define RTLD_LAZY 0x00001  /* Lazy function call binding.  */
#define RTLD_NOW  0x00002  /* Immediate function call binding.  */
#define  RTLD_BINDING_MASK   0x3 /* Mask of binding time value.  */
#define RTLD_NOLOAD  0x00004  /* Do not load the object.  */
#define RTLD_DEEPBIND   0x00008  /* Use deep binding.  */

/* If the following bit is set in the MODE argument to `dlopen',
   the symbols of the loaded object and its dependencies are made
   visible as if the object were linked directly into the program.  */
#define RTLD_GLOBAL  0x00100

/* Unix98 demands the following flag which is the inverse to RTLD_GLOBAL.
   The implementation does this by default and so we can define the
   value to zero.  */
#define RTLD_LOCAL   0

/* Do not delete object when closed.  */
#define RTLD_NODELETE   0x01000
```

Figure 21: Flags related to mode parameter.

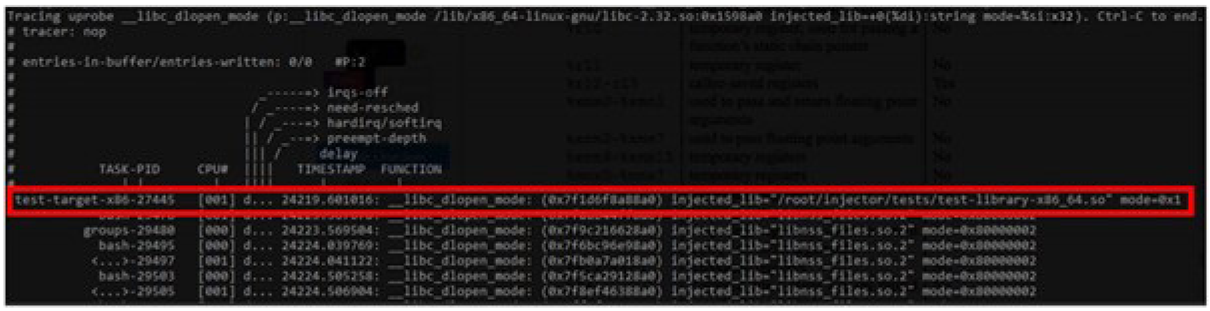
Searching all GLIBC releases since 2013 revealed no variations in the function prototype. As a result, for x86_64 the first two parameters will be stored in the rdi and rsi assembly registers.

Putting all this together we can now define a Uprobe. For the following example we have utilized Brendan Gregg's Uprobe tool [18], which is a wrapper around the F-Trace instrumentation [19]. Specifying the path to libc, offset to __libc_dlopen_mode, extracting the first parameter 'path', renaming this to 'injected lib' from the rdi register, and extracting the second parameter mode from the rsi register to a 32-bit hexadecimal format.

```
./uprobe -H 'p: /lib/x86_64-linux-gnu/libc2.32.so :0x1598a0 injected_lib=+0(%di):string
mode=%si:x32
```

Figure 22 shows the live tracing output when tested against kubo/injector [20]. This probe was run for a period of 24 hours on a *Ubuntu* host. The first event was the result of the successful injection of *test-library-x86_64.so* using the RTLD_

LAZY flag (0x1) into the *test-target-x86* process. A limited number of other events were produced, all of which supplied the RTLD_NOW flag and, more importantly, only specify the file name of the object to load rather than its full path.



```
Tracing uprobe __libc_dlopen_mode (p:__libc_dlopen_mode /lib/x86_64-linux-gnu/libc-2.32.so:0x1598a0 injected_lib=+0(%di):string mode=%si:x32). Ctrl-C to end.
# tracer: nop
#
# entries-in-buffer/entries-written: 0/0   #P:2
#
#                                _-----=> irqs-off
#                               / _----=> need-resched
#                              | / _---=> hardirq/softirq
#                              || / _--=> preempt-depth
#                              ||| /     delay
#           TASK-PID     CPU#  ||||    TIMESTAMP  FUNCTION
#              | |         |   ||||       |         |
test-target-x86-27445   [001] d... 24219.601016: __libc_dlopen_mode: (0x7f1d6f8a88a0) injected_lib="/root/injector/tests/test-library-x86_64.so" mode=0x1
         groups-29480   [000] d... 24223.569504: __libc_dlopen_mode: (0x7f9c216628a0) injected_lib="libnss_files.so.2" mode=0x80000002
           bash-29495   [000] d... 24224.039769: __libc_dlopen_mode: (0x7f6bc96e98a0) injected_lib="libnss_files.so.2" mode=0x80000002
          <...>-29497   [001] d... 24224.041122: __libc_dlopen_mode: (0x7fb0a7a018a0) injected_lib="libnss_files.so.2" mode=0x80000002
           bash-29503   [000] d... 24224.505258: __libc_dlopen_mode: (0x7f5ca29128a0) injected_lib="libnss_files.so.2" mode=0x80000002
          <...>-29505   [001] d... 24224.506904: __libc_dlopen_mode: (0x7f8ef46388a0) injected_lib="libnss_files.so.2" mode=0x80000002
```

Figure 22: __libc_dlopen_mode uprobe tracing output.

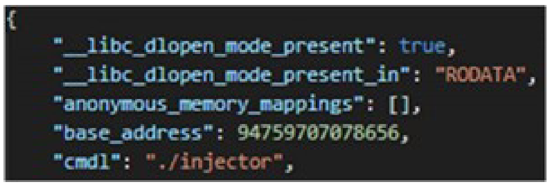
Uprobes provides the ability to include filters without introducing a significant performance hit. Introducing a filter for events specifying paths alongside objects resulted in hits only when testing attack tools such as ‘kubo/injection’ and ‘gaffe23/linux-inject’ [21]. So far, this serves as a high accuracy, low-cost detection strategy.

Detection of an attacking process

Detection of the same attack from an attacking process can be achieved in a number of ways:

1. Take the *__libc_dlopen_mode* Uprobe event (discussed previously) and work back, looking to see the most recent process that attached to it using *ptrace()*. This will be the injector process.
2. Another way is to look for the GDB method. Signaturing on command-line arguments supplied to gdb containing ‘__libc_dlopen_mode’.
3. Finally, say an implant is running on the host with this capability but it hasn’t yet been used, it may still be detectable in memory. This because the injector needs to:
    - Attach to a victim process to gain control over its execution flow.
    - Dynamically resolve the address of *__libc_dlopen_mode* using *__libc_dlsym()* from LIBC.
    - Restart the victim process at the function address, forcing it to execute *__libc_dlopen_mode*.

A consequence of this is that the injector process needs the ‘__libc_dopen_mode’ string to explicitly be defined in data to resolve it in memory using *dlsym()*. As such, it can be searched for in the .rodata section of the injector process. Figure 23 shows the *ELFieScanner* telemetry fields having caught the ‘kubo\injector’ tool via this method.



```
{
    "__libc_dlopen_mode_present": true,
    "__libc_dlopen_mode_present_in": "RODATA",
    "anonymous_memory_mappings": [],
    "base_address": 94759707078656,
    "cmdl": "./injector",
```

Figure 23:_libc_dlopen_mode string found in .rodata.

The downside of this mechanism is that it requires the injector process to be running at the time of the memory scan and the __libc_dlopen_mode string not to have been intentionally obfuscated by the malware author.

In addition to this, *ELFieScanner* also scans the GOT of a process for *__libc_dlopen_mode()* to see if this has been imported directly (which is highly unusual since *dlopen()* is the legitimate way to open a shared object).

REFLECTIVE SHARED OBJECT INJECTION

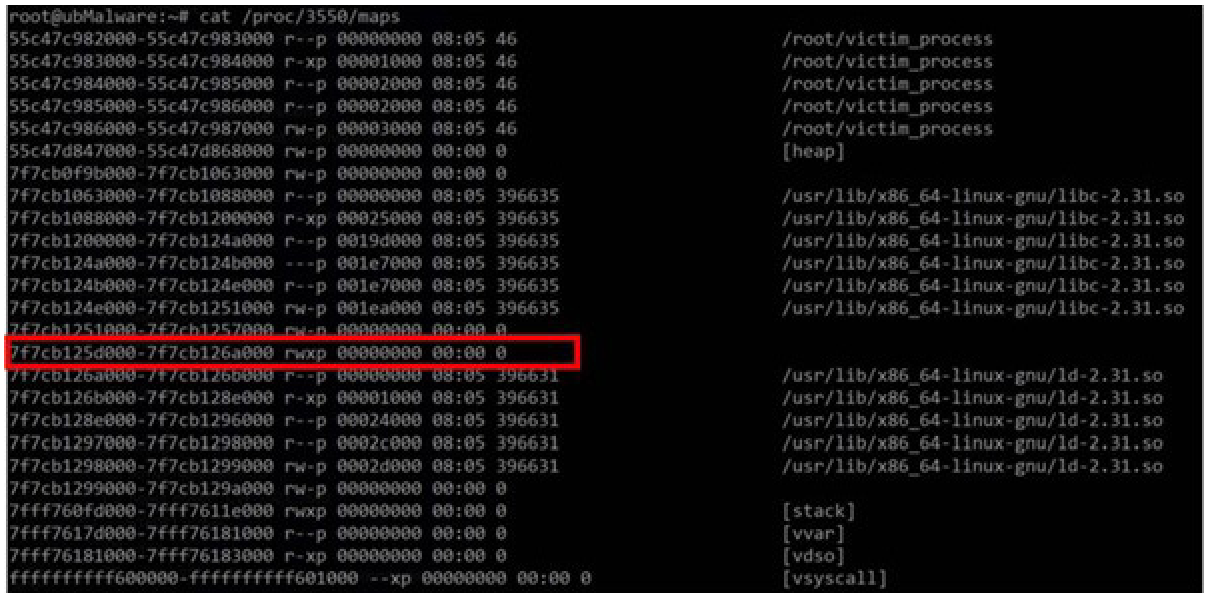
Reflective shared object injection is much like reflective dynamic library loading on *Windows*, whereby the shared object (DLL on *Windows*) is loaded directly from memory rather than disk to avoid rudimental detection mechanisms. In order to achieve this a custom loader performs the work of the traditional loader, such as mapping program segments into memory, loading required libraries, resolving functions and performing relocations.

TrustedSec’s blog post ‘Linux: How’s my memory’ [22] provides a great introduction to this topic alongside a practical demonstration with infosecguerilla’s ‘RefectiveSOinjection’ tool [23]. Many other tools, such as n1nj4sec’s ‘Pupy’ [24] post exploitation framework also make use of this technique in the same way. Briefly, it is achieved by:

1. Allocating a RWX anonymous memory region (defined as a memory mapping with no file or device backing it) by using mmap.

2. Mapping the shared object into this region.
3. Using a combination of the libc versions of *dlopen()*, *dlsym()*, *mprotect()* and *dlclose()* to resolve the new shared object's symbols and perform any relocations.

In reviewing the current literature, *Alien Labs*' 'Hunting for Linux library injection with Osquery' [11] describes how searching for anonymous memory regions with RWX permissions can reveal signs of RO injection. An example of what this may look like can be seen in Figure 24, whereby the RefectiveSOinjection tool has been used to reflectively load a library into the process named 'victim_process'.



```
root@ubMalware:~# cat /proc/3550/maps
55c47c982000-55c47c983000 r--p 00000000 08:05 46                         /root/victim_process
55c47c983000-55c47c984000 r-xp 00001000 08:05 46                         /root/victim_process
55c47c984000-55c47c985000 r--p 00002000 08:05 46                         /root/victim_process
55c47c985000-55c47c986000 r--p 00002000 08:05 46                         /root/victim_process
55c47c986000-55c47c987000 rw-p 00003000 08:05 46                         /root/victim_process
55c47d847000-55c47d868000 rw-p 00000000 00:00 0                          [heap]
7f7cb0f9b000-7f7cb1063000 rw-p 00000000 00:00 0
7f7cb1063000-7f7cb1088000 r--p 00000000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7cb1088000-7f7cb1200000 r-xp 00025000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7cb1200000-7f7cb124a000 r--p 0019d000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7cb124a000-7f7cb124b000 ---p 001e7000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7cb124b000-7f7cb124e000 r--p 001e7000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7cb124e000-7f7cb1251000 rw-p 001ea000 08:05 396635                     /usr/lib/x86_64-linux-gnu/libc-2.31.so
7f7cb1251000-7f7cb1257000 rw-p 00000000 00:00 0
7f7cb125d000-7f7cb126a000 rwxp 00000000 00:00 0
7f7cb126a000-7f7cb126b000 r--p 00000000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7cb126b000-7f7cb128e000 r-xp 00001000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7cb128e000-7f7cb1296000 r--p 00024000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7cb1297000-7f7cb1298000 r--p 0002c000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7cb1298000-7f7cb1299000 rw-p 0002d000 08:05 396631                     /usr/lib/x86_64-linux-gnu/ld-2.31.so
7f7cb1299000-7f7cb129a000 rw-p 00000000 00:00 0
7fff760fd000-7fff7611e000 rwxp 00000000 00:00 0                          [stack]
7fff7617d000-7fff76181000 r--p 00000000 00:00 0                          [vvar]
7fff76181000-7fff76183000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

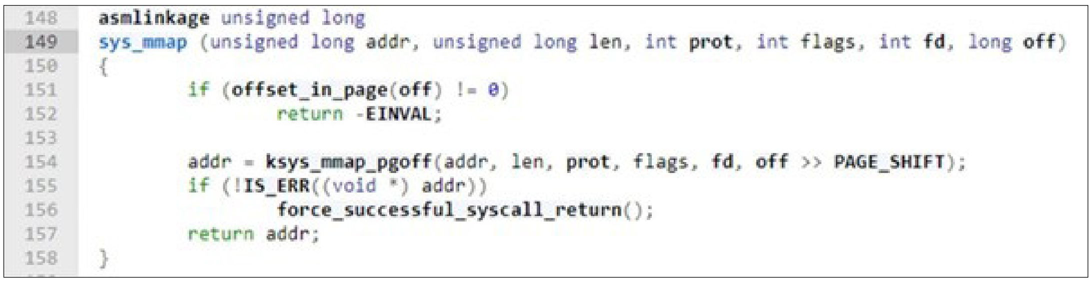
Figure 24: An RWX anonymous memory region created by the 'ReflectiveSOinjection' tool.

However, by questioning what variables an attacker has control over we can start to envisage ways to circumvent such a detection, for instance:

- If the process maps have been collected from */proc/<pid>/maps* we know (from Part 1) user-mode functions used to read this data can be hooked to supply false output. An attacker could use this technique hide their anonymous RWX memory region.
- In addition, process mappings only reflect the current permissions. There is nothing stopping an attacker initially creating an RWX mapping using *mmap()* and then subsequently removing the Writable permission using *mprotect()*.

So, to build upon this we could instead look to trigger on the initial allocation itself, rather than the current permission state of an anonymous memory region. To do this we can utilize Kprobes [16] to target the underlying kernel function for *mmap()*. This way, not only is the detection produced in real time, but it also unaffected by user-mode hooking since the probe is placed by the F-Trace subsystem on a kernel function.

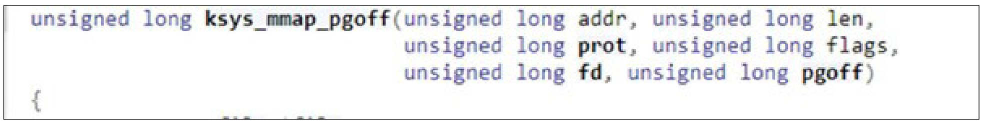
The /proc/kallsyms file is created when the kernel boots up. It contains a list of symbols exported by the kernel and the corresponding addresses. Unfortunately, *mmap()* isn't present. So we must first analyse the underlying kernel source to identify an alternative symbol to target. Figure 25 shows the routine for *mmap()* – notice it invokes the function 'ksys_mmap_pgoff'.



```c
asmlinkage unsigned long
sys_mmap (unsigned long addr, unsigned long len, int prot, int flags, int fd, long off)
{
        if (offset_in_page(off) != 0)
                return -EINVAL;

        addr = ksys_mmap_pgoff(addr, len, prot, flags, fd, off >> PAGE_SHIFT);
        if (!IS_ERR((void *) addr))
                force_successful_syscall_return();
        return addr;
}
```

Figure 25: Sys_mmap function in Linux kernel source [25].

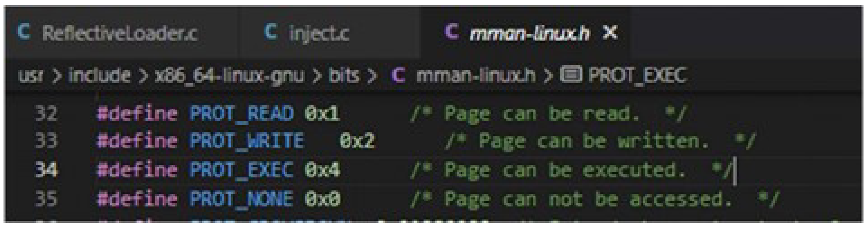
If we analyse the 'ksys_mmap_pgoff' function located in '/mm/mmap.c' (Figure 26) we can see it makes use of the same arguments as 'sys_mmap'. It is also exported by the kernel since it appears in the */proc/kallsyms* file. Using this we start to construct our Kprobe.



```
unsigned long ksys_mmap_pgoff(unsigned long addr, unsigned long len,
                              unsigned long prot, unsigned long flags,
                              unsigned long fd, unsigned long pgoff)
{
```

Figure 26: Kysys_mmap_pgoff function parameters [26].

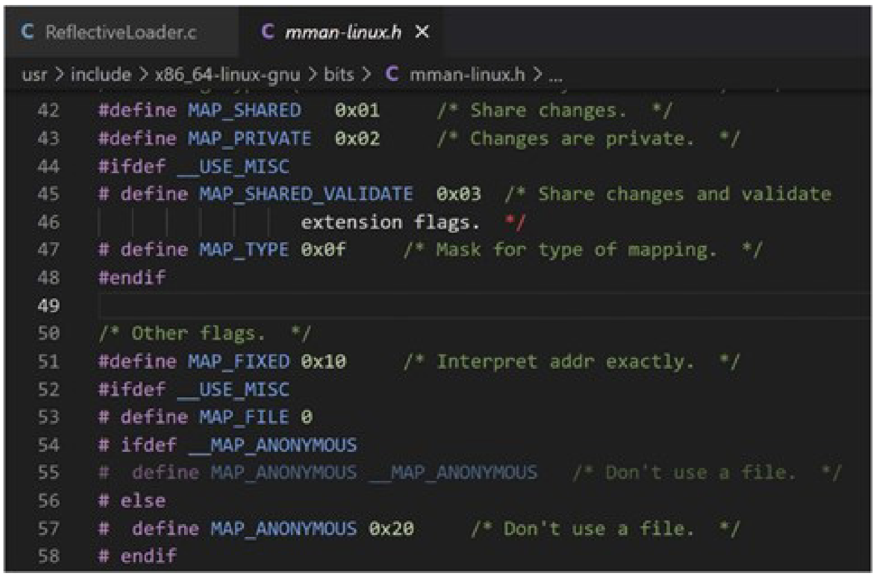
Knowing the function prototype identifies the registers (x64) or stack values (x86) that can be collected to extract the function parameters. In this case we are particularly interested in the ‘prot’ and ‘flags’ parameters that are used to set the permissions and memory type, respectively. Under x86_x64 ‘prot’ would be stored in the ‘dx’ register and ‘flags’ in the ‘cx’ register. Both parameters represent bit fields that are OR’d together to determine combined types. Figures 27 and 28 show the available flags.



ReflectiveLoader.c
inject.c
mman-linux.h
usr > include > x86_64-linux-gnu > bits > mman-linux.h > PROT_EXEC

```
#define PROT_READ 0x1      /* Page can be read.  */
#define PROT_WRITE   0x2      /* Page can be written.  */
#define PROT_EXEC 0x4      /* Page can be executed.  */
#define PROT_NONE 0x0      /* Page can not be accessed.  */
```

Figure 27: Memory protection flags.



ReflectiveLoader.c
mman-linux.h
usr > include > x86_64-linux-gnu > bits > mman-linux.h > ...

```
#define MAP_SHARED   0x01     /* Share changes.  */
#define MAP_PRIVATE  0x02     /* Changes are private.  */
#ifdef __USE_MISC
# define MAP_SHARED_VALIDATE  0x03  /* Share changes and validate
                   extension flags.  */
# define MAP_TYPE 0x0f     /* Mask for type of mapping.  */
#endif

/* Other flags.  */
#define MAP_FIXED 0x10     /* Interpret addr exactly.  */
#ifdef __USE_MISC
# define MAP_FILE 0
# ifdef __MAP_ANONYMOUS
#  define MAP_ANONYMOUS __MAP_ANONYMOUS   /* Don't use a file.  */
# else
#  define MAP_ANONYMOUS 0x20     /* Don't use a file.  */
# endif
```

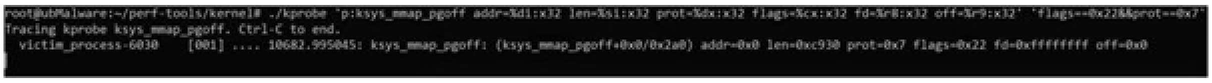
Figure 28: Memory type flags.

For the ‘prot’ parameter, performing a logical OR using the MAP_PRIVATE and MAP_ANONYMOUS flags outputs the value ‘0x22’. For the ‘flags’ parameter, PROT_READ, PROT_WRITE, PROT_EXEC outputs the value ‘0x7’. Using these values we can create a Kprobe with a filter that only outputs results for allocations in private anonymous memory with RWX permissions, as seen in Figure 29.

```
./kprobe 'p:ksys_mmap_pgoff addr=%di:x32 len=%si:x32 prot=%dx:x32 flags=%cx:x32 fd=%r8:x32
off=%r9:x32' 'flags==0x22&&prot==0x7'
```

Figure 29: Kprobe for ksys_mmap_pgoff.

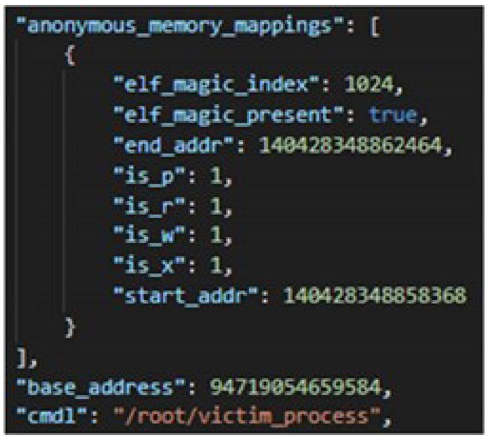
When tested against the ‘RefectiveSOinjection’ tool it successfully identifies the creation of the RWX private anonymous mapping in ‘victim_process’ (Figure 30).



```
root@ubMalware:~/perf-tools/kernel# ./kprobe 'p:ksys_mmap_pgoff addr=%di:x32 len=%si:x32 prot=%dx:x32 flags=%cx:x32 fd=%r8:x32 off=%r9:x32' 'flags==0x22&&prot==0x7'
Tracing kprobe ksys_mmap_pgoff. Ctrl-C to end.
  victim_process-6030    [001] .... 10682.995045: ksys_mmap_pgoff: (ksys_mmap_pgoff+0x0/0x2a0) addr=0x0 len=0xc930 prot=0x7 flags=0x22 fd=0xffffffff off=0x0
```

Figure 30: Kprobe telemetry for ksys_mmap_pgoff.

However, not all processes adhere to the same rules and may create legitimate instances of private anonymous memory regions with RWX permissions (python processes are a good example of this). Taking this a step further, we can then look inside the memory region. This is possible since the scanning is much more targeted at this stage and isn’t subject to the same performance overhead of full process memory scans. By looking for the existence of ELF magic bytes we are able to determine if a shared object has been injected. Figure 31 represents telemetry produced after running *ELFieScanner* against the 'victim_process'; it shows that ELF executable header magic bytes have been identified 1,024 bytes from the start of the anonymous RWX memory region.



```
"anonymous_memory_mappings": [
    {
        "elf_magic_index": 1024,
        "elf_magic_present": true,
        "end_addr": 140428348862464,
        "is_p": 1,
        "is_r": 1,
        "is_w": 1,
        "is_x": 1,
        "start_addr": 140428348858368
    }
],
"base_address": 94719054659584,
"cmdl": "/root/victim_process",
```

Figure 31: RWX memory region scan result.

HIDING SIGNS OF SO INJECTION

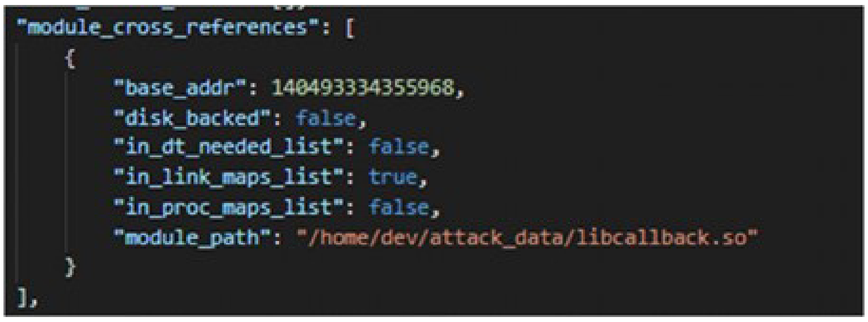
It is common for attackers to attempt to hide their presence on the host so as to not raise suspicion. However, this behaviour is usually a clear sign of malevolence and can form part of a good detection strategy. One way to spot if a shared object has been hidden is to correlate different enumeration methods. *ELFieScanner* uses three methods to detect loaded shared objects:

1. Reading the */proc/<pid>/maps* file for each process.
2. Enumerating the *link_map* linked list (see Part 1 for how this is achieved).
3. Collecting all the DT_NEEDED entries from the *.dynamic* section.

Both the linked *link_map* and */proc/<pid>/maps* specify the loaded base addresses and names of every module that has been loaded by the runtime linker; as a result, these should match. DT_NEEDED entries contain only the names of shared objects to load, not the base addresses; they should be present in both the *link_map* and */proc/<pid>/maps*, but unlike them they are non-exhaustive lists because each shared object may have its own dependencies that need to be loaded in addition to itself. To determine if a shared object has been hidden *ELFieScanner* looks for:

- Shared objects that only appear in either the *link_map* or *proc/<pid>/maps* but not both.
- DT_NEEDED entries that don't appear in either the *link_map* or *proc/<pid>/maps*.

Figure 32 shows the resultant telemetry produced when scanning a process that has attempted to hide one of its shared objects from the */proc/<pid>/maps* file system (in_proc_maps_list = false). In addition, notice the shared object is not backed by a file on disk (disk_backed = false). Attackers can often delete shared objects once loaded as another way to hide. Fortunately, from a detection standpoint this is uncommon to see and serves as a good indicator of foul play.



```
"module_cross_references": [
    {
        "base_addr": 140493334355968,
        "disk_backed": false,
        "in_dt_needed_list": false,
        "in_link_maps_list": true,
        "in_proc_maps_list": false,
        "module_path": "/home/dev/attack_data/libcallback.so"
    }
],
```

Figure 32: Libcallback.so hidden from two enumeration methods.

REFERENCES

[1] GNU Binutils. https://www.gnu.org/software/binutils/.

[2] Sanmillan, I. Executable and Linkable Format 101 - Part 1 Sections and Segments. Intezer. 2 January 2018. https://www.intezer.com/blog/research/executable-linkable-format-101-part1-sections-segments.

[3] elfmaster / dt_infect. https://github.com/elfmaster/dt_infect.

[4] Daniel Jary / ELFieScanner. https://github.com/JanielDary/ELFieScanner.

[5] chokepoint / azazel. https://github.com/chokepoint/azazel.

[6] unix-thrust / beurk. https://github.com/unix-thrust/beurk.

[7] chokepoint / Jynx2. https://github.com/chokepoint/Jynx2.

[8] mempodippy / vlany. https://github.com/mempodippy/vlany.

[9] NexusBots / Umbreon-Rootkit. https://github.com/NexusBots/Umbreon-Rootkit.

[10] Sanmillan, I. HiddenWasp Malware Stings Targeted Linux Systems. Intezer. 29 May 2019. https://www.intezer.com/blog/malware-analysis/hiddenwasp-malware-targeting-linux-systems/.

[11] Blasco, J. Hunting for Linux library injection with Osquery. LevelBlue. 20 June 2019. https://cybersecurity.att.com/blogs/labs-research/hunting-for-linux-library-injection-with-osquery.

[12] Kaplan, D. Linux code injection paint-by-numbers. LinkedIn. 11 January 2021. https://www.linkedin.com/pulse/linux-code-injection-paint-by-numbers-david-kaplan.

[13] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[14] Stuart. Process Injection with GDB. https://magisterquis.github.io/2018/03/11/process-injection-with-gdb.html.

[15] ltrace(1) — Linux manual page. https://man7.org/linux/man-pages/man1/ltrace.1.html.

[16] LWN.net. An introduction to KProbes. https://lwn.net/Articles/132196/#:~:text=KProbes%20is%20a%20debugging%20mechanism,specific%20event s%2C%20trace%20problems%20etc.

[17] Dronamraju, S. Uprobe-tracer: Uprobe-based Event Tracing. https://www.kernel.org/doc/html/latest/trace/uprobetracer.html.

[18] Gregg, B. Linux uprobe: User-Level Dynamic Tracing. Brendan Gregg's Blog. 28 June 2015. https://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html.

[19] LWN.net. Debugging the kernel using Ftrace – part 1. https://lwn.net/Articles/365835/.

[20] kubo / injector. https://github.com/kubo/injector.

[21] gaffe23 / linux-inject. https://github.com/gaffe23/linux-inject.

[22] Haubris, K. Linux: How's My Memory. TrustedSec. 18 September 2018. https://www.trustedsec.com/blog/linux-hows-my-memory/.

[23] infosecguerrilla / ReflectiveSOInjection. https://github.com/infosecguerrilla/ReflectiveSOInjection.

[24] n1nj4sec / pupy. https://github.com/n1nj4sec/pupy.

[25] https://elixir.bootlin.com/linux/v5.19.17/source/arch/ia64/kernel/sys_ia64.c#L149.

[26] https://elixir.bootlin.com/linux/v5.19.17/source/mm/mmap.c#L1595.