ELF shared library injection forensics

By backtrace

Published: 2016-04-22 · Archived: 2026-04-06 15:17:27 UTC

At Backtrace we built and are continually building security and forensics features into our product that rely on understanding the structural nuances of ELF binary internals, and process memory infection techniques. This article outlines some of the core concepts that are being applied in our technology today.

For well over a decade attackers have been installing memory resident backdoors, rootkits, and parasites of various kinds into userland processes. The goal is to inject executable code into an existing process, to alter its functionality, while remaining stealth and keeping the attackers activity surreptitious. The most commonly used approach for inserting executable code into an existing process is a technique that has been termed *shared library injection*. This approach is flexible for a variety of reasons, and it can be very difficult for forensics investigators to detect whether the shared library is legitimate or not. The focus of this blog post is going to be centered around how to identify legitimate shared library objects vs. suspicious and potentially malicious ones. Gaining this knowledge is essential to move forward in identifying other related attacks such as PLT/GOT infections, which allow an attacker to hijack shared library functions and alter the flow of execution for their own gain.

The following is a great talk by Georg Wicherski, speaking at *Syscan 2013* about a real world example of an Advanced Nation State Actor who maintained stealth residence on a large European IT companies network, by using **ELF shared library injection** to modify the functionality behind the Apache web-server process, creating a stealth and nearly undetectable backdoor into the Linux servers on the network.

Building the process image

Linux and other modern UNIX flavor operating systems use the ELF binary format as the basis for building a process image, which generally consists of an executable file and shared libraries who have had their segments mapped into the address space. The internals to how this is accomplished vary by degree between operating systems, but typically the kernel is responsible for loading the programs individual segments, of which there is typically a segment for code and a segment for data. These are marked by program headers that are of a type called `PT_LOAD`. The kernel is also responsible for loading the program interpreter, also known as the dynamic linker (i.e. `/lib/x86_64/ld-linux.so`), which is marked by a single program header of type `PT_INTERP`. Using the readelf utility we can view the program headers of a simple dynamic-linked executable to demonstrate this.

```
$ readelf -l host
```

Elf file type is EXEC (Executable file)
Entry point 0x400440
There are 9 program headers, starting at offset 64

Program Headers:
Type Offset VirtAddr PhysAddr

FileSiz MemSiz Flags Align
PHDR 0x0000000000000040 0x0000000000400040 0x0000000000400040
0x00000000000001f8 0x00000000000001f8 R E 8
INTERP 0x0000000000000238 0x0000000000400238 0x0000000000400238
0x000000000000001c 0x000000000000001c R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD 0x0000000000000000 0x0000000000400000 0x0000000000400000
0x0000000000000764 0x0000000000000764 R E 200000
LOAD 0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
0x0000000000000230 0x0000000000000238 RW 200000
DYNAMIC 0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
0x00000000000001d0 0x00000000000001d0 RW 8

The above output is slightly truncated to show only the most relevant parts of the executable. The `INTERP` program header displays the path to the dynamic linker, and the two `LOAD` program headers mark the text and data segment. Also notice the `DYNAMIC` segment, which contains data stored as an array of structs. Note that the dynamic segment exists within the data segment range, so it gets loaded into memory along with the second `LOAD` segment.

The dynamic segment is an array of ElfN_Dyn structs

```
typedef struct {
Elf64_Sxword d_tag;
union {
Elf64_Xword d_val;
Elf64_Addr d_ptr;
} d_un;
} Elf64_Dyn;
extern Elf64_Dyn _DYNAMIC[];
```

The dynamic segment is parsed by the dynamic linker at runtime, as it contains all of the necessary information for the dynamic linker to begin finding shared library dependencies, and linking them into the process using a complex formula of *relocation* based *runtime patching*. The dynamic linker is itself a shared library object. It is responsible for hot patching a process's memory so that shared executable code executes within the current memory layout. These patches are applied with the help of *ELF relocation records*, which are stored in ELF sections of type `SHT_REL`, and `SHT_RELA`. More details on the internals of relocations can be found in the ELF specs, and also described with some easier to understand examples in the book Learning Linux binary analysis.

For the purpose of this article we are most interested in the way that the dynamic linker determines which shared library objects should be linked into a process image. The `ElfN_Dyn` struct has a member called `d_tag` which tells the dynamic linker which type of data is stored in the struct. Since there is an array of them each one may contain values relating to different things. See man elf(5) for a complete list of possible `d_tag` definitions. The dynamic linker looks for dynamic entries that hold a `d_tag` value of type `DT_NEEDED` to determine which shared objects are to be linked into the address space.

The dynamic segment viewed with readelf

`$ readelf -d host` Dynamic section at offset 0xe28 contains 24 entries:
Tag Type Name/Value
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
0x000000000000000c (INIT) 0x4003e0
0x000000000000000d (FINI) 0x4005f4
0x0000000000000019 (INIT_ARRAY) 0x600e10
0x000000000000001b (INIT_ARRAYSZ) 8 (bytes)
0x000000000000001a (FINI_ARRAY) 0x600e18
0x000000000000001c (FINI_ARRAYSZ) 8 (bytes)
0x000000006ffffef5 (GNU_HASH) 0x400298
0x0000000000000005 (STRTAB) 0x400318
0x0000000000000006 (SYMTAB) 0x4002b8
0x000000000000000a (STRSZ) 61 (bytes)
0x000000000000000b (SYMENT) 24 (bytes)
0x0000000000000015 (DEBUG) 0x0
0x0000000000000003 (PLTGOT) 0x601000
0x0000000000000002 (PLTRELSZ) 72 (bytes)
0x0000000000000014 (PLTREL) RELA
0x0000000000000017 (JMPREL) 0x400398
0x0000000000000007 (RELA) 0x400380
0x0000000000000008 (RELASZ) 24 (bytes)
0x0000000000000009 (RELAENT) 24 (bytes)
0x000000006ffffffe (VERNEED) 0x400360
0x000000006fffffff (VERNEEDNUM) 1
0x000000006ffffff0 (VERSYM) 0x400356
0x0000000000000000 (NULL) 0x0

Nearly all of the dynamic segment entries shown in the above output are relevant and necessary for the dynamic linker to accomplish what it needs on one level or another, and are explained in more depth in the ELF(5) manual pages. In particular we are interested in the `NEEDED` entries, because these allow us to see which shared libraries are needed by the dynamic linker, legitimately or not. This leads us to the first *infection point* that can be used by attackers, that we call a **DT_NEEDED infection**. This technique was originally published in the Cerberus ELF interface phrack article, by Mayhem. This is a *direct binary modification* technique that requires the attacker to modify the executable program on-disk, therefore being less stealth since this type of blatant modification can be picked up by a simple IDS program such as *tripwire*. Otherwise it can often be detected by the trained eye; typically an attacker overwrites the `DT_DEBUG` entry with an extra `DT_NEEDED` entry which points to a shared library path of the attackers choosing. This is easily spotted because the `DT_NEEDED` entries are expected to be contiguous. Now examine the following readelf output to see how this stands out like a sore thumb.

Using readelf to see how firefox has been infected

```
Dynamic section at offset 0x1da98 contains 33 entries:
Tag Type Name/Value
0x0000000000000001 (NEEDED) Shared library: [libpthread.so.0]
0x0000000000000001 (NEEDED) Shared library: [libdl.so.2]
0x0000000000000001 (NEEDED) Shared library: [libstdc++.so.6]
0x0000000000000001 (NEEDED) Shared library: [libm.so.6]
0x0000000000000001 (NEEDED) Shared library: [libgcc_s.so.1]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
0x0000000000000001 (NEEDED) Shared library: [ld-linux-x86-64.so.2]
0x000000000000000c (INIT) 0x41e0
0x000000000000000d (FINI) 0x169f0
0x0000000000000019 (INIT_ARRAY) 0x21d8c0
0x000000000000001b (INIT_ARRAYSZ) 24 (bytes)
0x000000000000001a (FINI_ARRAY) 0x21d8d8
0x000000000000001c (FINI_ARRAYSZ) 8 (bytes)
0x000000006ffffef5 (GNU_HASH) 0x2d0
0x0000000000000005 (STRTAB) 0x1cd0
0x0000000000000006 (SYMTAB) 0x788
0x000000000000000a (STRSZ) 5575 (bytes)
0x000000000000000b (SYMENT) 24 (bytes)
0x0000000000000015 (NEEDED) Shared library: [sneaky_rabbit.so.1]
0x0000000000000003 (PLTGOT) 0x21dce8
0x0000000000000002 (PLTRELSZ) 2016 (bytes)
0x0000000000000014 (PLTREL) RELA
0x0000000000000017 (JMPREL) 0x3a00
0x0000000000000007 (RELA) 0x35b0
0x0000000000000008 (RELASZ) 1104 (bytes)
0x0000000000000009 (RELAENT) 24 (bytes)
0x0000000000000018 (BIND_NOW)
0x000000006ffffffb (FLAGS_1) Flags: NOW
0x000000006ffffffe (VERNEED) 0x3460
0x000000006fffffff (VERNEEDNUM) 7
0x000000006ffffff0 (VERSYM) 0x3298
0x000000006ffffff9 (RELACOUNT) 36
0x0000000000000000 (NULL) 0x0
```

Even if the injected shared library wasn't named so silly, like `sneaky_rabbit.so.1`, it would still be very apparent. It is not as easy for an attacker to create a new entry in the dynamic segment as it is to overwrite an existing one. Therefore overwriting `DT_DEBUG` is perfect since it is only used for debugging purposes. The reason it is difficult for an attacker to add a new item to the dynamic segment is because the `.got.plt` section directly follows it which cannot be shifted forward without also adjusting every single `.rela.plt` entry, who's `r_offset` members reference the locations within the `.got.plt` section. We cannot rule out possible infection methods, but it is highly unlikely that attackers will reliably insert a phony `DT_NEEDED` entry that is contiguous

with the ones generated by the linker. In short, it is safe to say that we have established a generally reliable way to detect bogus `DT_NEEDED` entries by checking to see if one has been inserted over the `DT_DEBUG` entry.

Legitimate shared library linking

In order to detect suspicious shared objects in a process image we must first understand what a legitimate shared library object looks like. The dynamic linker loads shared libraries in three primary ways that can be considered as legitimate:

Valid DT_NEEDED entries

Assuming the `DT_NEEDED` entry is valid, which we can ascertain with some degree of confidence using the knowledge just described: If the `DT_NEEDED` entries are contiguous, then they are most likely legitimate.

Most attackers are interested in avoiding modification of the binary file on disk, but we would be remiss to not mention the **DT_NEEDED infection** as it is necessary to validate that the entries we are viewing are legitimate before using them as a baseline for detecting other infections that rely on *in-memory shared library injection* techniques. Also sometimes called *Reflective DLL injection*
techniques.

It is important to note that the `DT_NEEDED` entries must be transitively examined. For every `DT_NEEDED` entry there is a shared object file with with its own `DT_NEEDED` entries, so there is a dependency tree that begins with the parent object, and that must be followed through each child object. Duplicates will come up, especially with `libc.so` since so many different shared libraries rely on it, but the duplicates are simply ignored.

Examples of dependency resolution

```
$ readelf -d /bin/ls | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [libselinux.so.1]
0x0000000000000001 (NEEDED) Shared library: [libacl.so.1]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
$ readelf -d /lib/x86_64-linux-gnu/libselinux.so.1 | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [libpcre.so.3]
0x0000000000000001 (NEEDED) Shared library: [libdl.so.2]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
0x0000000000000001 (NEEDED) Shared library: [ld-linux-x86-64.so.2]
$ readelf -d /lib/x86_64-linux-gnu/libacl.so.1 | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [libattr.so.1]
0x0000000000000001 (NEEDED) Shared library: [libc.so.6]
$ readelf -d /lib/x86_64-linux-gnu/libc.so.6 | grep NEEDED
0x0000000000000001 (NEEDED) Shared library: [ld-linux-x86-64.so.2]
```

With the `readelf` command we can demonstrate how the resolution is actually accomplished, but to resolve all of them from the command line it really makes sense just to use the `/usr/bin/ldd` command that is available in Linux and FreeBSD. This will show them all without duplicates. Keep in mind too that `linux-vdso.so.1` doesn't

originate from a `DT_NEEDED` entry, it is mapped into `glibc` linked processes, by the Linux kernel, and it does not exist at all in FreeBSD.

```
$ /usr/bin/ldd /bin/ls
linux-vdso.so.1 => (0x00007fff7f1e9000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f699ec1b000)
libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f699ea12000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f699e647000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f699e3da000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f699e1d6000)
/lib64/ld-linux-x86-64.so.2 (0x000055a1ccba0000)
libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f699dfd0000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f699ddb2000)
```

Libraries that are preloaded with LD_PRELOAD

Although some userland rootkits rely on the `LD_PRELOAD` environment variable for linking the shared library code into the process image, it is a legitimate way to link a shared object and without some analysis being performed on a shared object that was preloaded, we will assume that the library is legitimate. There are some fairly surefire ways to quickly identify if a preloaded shared library is malicious, especially if it has symbol names that override common `libc.so` functions, such as `read`, `write`, `open`, `socket`, etc. This indicates that the shared library is trying to override or replace existing shared library functionality and is almost a sure sign of some type of code patching, but is beyond the scope of this article. Keep in mind that most serious attackers are not going to use `LD_PRELOAD` because it requires them restarting whatever process they want to infect, in order to get the dynamic linker to preload their shared object. Most attackers want to surreptitiously infect an existing process image without having to create a new process.

Libraries that are loaded with dlopen()

```
void * dlopen(const char *filename, int flag)
```

In Linux the `dlopen` function is apart of the `libdl` shared library, and it is considered the legitimate way to request the dynamic linker to dynamically load a shared library object at any given point in runtime. By examining a programs symbol table we can identify whether or not it is using the `dlopen` function and use some static analysis to see what string value is being passed to the function. This has to be done at runtime since not all strings will be stored in the `.rodata` and `.data` sections on-disk. It is possible that they were pulled from some external source and then stored in a stack buffer, a heap buffer, or even in the `.bss` section which marks the uninitialized gives global data found at the tail end of the data segment.

In FreeBSD the `dlopen` function exists only in `ld-elf.so`, but has a fake symbol in `libc.so`. The purpose (or side-effect) of this, is to waste an attackers time, who now believe that their shellcode should be invoking a function in `libc.so` that doesn't even exist. The fake `dlopen` symbol looks legitimate, and the executables have a `dlopen@plt` entry that presumably transfers control to the `dlopen` in `libc.so`, but in actuality control is being transferred to the `dlopen` function that exists in the dynamic linker. I just recently acquired this

knowledge from a colleague, and verified it by using GDB to inspect a program that calls `dlopen`. Upon inspection I verified that the global offset table entry for `dlopen` contained a value that points into the `ld-elf.so` address range, and not into the `libc.so` address range. A quick look at the disassembly also reveals that `dlopen` in `libc.so` does nothing more than pass an argument with the current time zone to `__rtld_error`, resulting in failure for any attackers who are attempting to trigger it.

Fake dlopen in FreeBSD's libc.so

```
0000000000123540 <dlopen>:
123540: 55 push %rbp
123541: 48 89 e5 mov %rsp,%rbp
123544: 48 8d 3d d5 c5 25 00 lea 0x25c5d5(%rip),%rdi # 37fb20 <tzname+0x10>
12354b: 31 c0 xor %eax,%eax
12354d: e8 42 44 f1 ff callq 37994 <_rtld_error@plt>
123552: 31 c0 xor %eax,%eax
123554: 5d pop %rbp
```

/proc/pid/maps and the final analysis

To find each shared library object within a process image, we may use the more naive approach of analyzing the first 64 `sizeof Elf64_Ehdr` bytes of file backed memory mappings. Lets not be short sighted though and only analyze file backed mappings when it is totally conceivable for an attacker to store the injected shared object into anonymous memory mappings, such as those created with the `mmap` flag `MAP_ANONYMOUS`. We should only observe the memory maps that have executable permissions, since the initial ELF file header exists in the text region. If the mapping is found to be an executable ELF object of type `ET_DYN` and it's within an anonymous memory mapping, it should be immediately flagged as suspicious. On other hand, if it is a file mapping, then we must put it through heuristics to determine if it was legitimately linked using one of the three primary linking concepts described above.

- Check if it has a corresponding legitimate `DT_NEEDED` entry through transitive dependency resolution
- Check if its path was set with the `LD_PRELOAD` environment variable, or if it is a dependency of a shared object that was set by `LD_PRELOAD`
- Check if its path was passed in a call to `dlopen`, or if it is a dependency of any shared objects that were linked by `dlopen`

If none of the above heuristics returned true, then the shared object should be flagged as suspicious. This leads us to the eventual question of "What other ways can a shared library be linked into a process?"

Shared library injection techniques

Although it is enough to know only the legitimate ways in which a shared library is linked in order to detect suspicious ones, it can give a forensics investigator some advantage in atleast knowing the other ways in which a shared library can be linked into an already existing process image by an attacker. This way, if the investigator is able to do some reverse engineering work, they will know exactly what to look for.

open/mmap based shellcode

Attackers can inject shellcode into a process using the `ptrace` system call. The shellcode uses the `open` and `mmap` system calls to map the shared library into the process address space. This may be mapped in either as a file backed mapping *(which is more obvious since it shows up in `/proc/pid/maps`)*, or it may store the code into an anonymous memory mapping. At this point the attacker must use some relatively complex code that is capable of using `ptrace` to apply all of the necessary relocations to the shared library just as the dynamic linker would to make it suitable for execution within the given address space.

VDSO manipulation

As discussed in the 2009 paper titled Modern day ELF Runtime infection via GOT poisoning, the attacker may not be allowed to inject shellcode into the address space directly with the `PTRACE_POKETEXT` request due to PaX restrictions which prevent even the `ptrace` system call from writing to read-only regions, due to the `access_process_vm` kernel function bailing out when *mprotect restrictions* are enabled. After discussions with a colleague and some experimentation, it quickly became evident that an attacker could hijack control of the code stubs mapped in from the `linux-vdso.so` object, allowing for *glibc syscall hijacking* by resetting the `%rax` register with the desired syscall number, and by setting the other registers with the needed arguments. Using some `ptrace` trickery, the user can set the instruction pointer to this VDSO code using the modified registers to redirect execution to the desired system calls, such as `open` and `mmap`. Once the shared library has been mapped into the process address space, and any relocations have been applied, the user can reset the instruction pointer back to the start of the VDSO stub with the original register values set and execute the syscall that was intended before control was hijacked.

__libc_dlopen_mode shellcode

This tends to be the most commonly used method for attacker driven shared library loading. The `__libc_dlopen_mode` function exists in the GNU C library, which is linked into virtually every process on a given Linux system. This means that an attacker can inject shellcode into a process that invokes `__libc_dlopen_mode` which in-turn will map the requested shared library into the process, and handle applying all of the relocations, thus removing the need for the attacker to understand all of the ELF relocation internals. This technique is a solid and stable way to inject shared libraries into a process, although it is not the most stealth since the shared library is mapped into the process as a file-backed mapping instead of an anonymous mapping. Nonetheless it seems to be the most popular technique *(Not that any of this knowledge is particularly well known or popular)*.

FreeBSD dlopen shellcode

In the FreeBSD operating system the `dlopen` function only exists in the dynamic linker, as mentioned previously. An attacker can simply use a command such as `readelf -s /libexec/ld-elf.so | grep dlopen` to acquire the offset and then calculate the final address at runtime, before injecting shellcode that invokes `dlopen` directly.

Legitimate dlopen calls vs. illegal (attacker) dlopen calls

The way to distinguish between an illegal `dlopen` call, and a legitimate `dlopen` call stands the same for both Linux and FreeBSD. A legitimate `dlopen` call will **always** be a `call` instruction into the PLT (procedure linkage table). This can easily be observed by disassemblers which track the ELF sections, and thus will see any calls that refer to the `.plt` section.

Example of legitimate dlopen cal

```
 0000000000400678 <main>:
400678: 55 push %rbp
400679: 48 89 e5 mov %rsp,%rbp
40067c: 48 83 ec 10 sub $0x10,%rsp
400680: be 00 00 00 00 mov $0x0,%esi
400685: bf 34 07 40 00 mov $0x400734,%edi
40068a: e8 b1 fe ff ff callq <dlopen@plt>
40068f: 48 89 45 f8 mov %rax,-0x8(%rbp)
```

Typically an attacker will be using `ptrace` to inject code somewhere into the process address space that temporarily overwrites some existing code in the text segment, and any calls to the `dlopen` function will typically be directly to the code for the `dlopen` function **and not into the PLT entry address**, of which we can easily establish the address range using the `readelf` tool, and verify.

```
$ readelf -S test | grep ".plt" -A1 | egrep -v 'got|rela|WA|AI'
[12] .plt PROGBITS 0000000000400500 00000500
0000000000000050 0000000000000010 AX 0 0 16
```

Looking at a legitimate `dlopen@plt` call does bring to mind the possibility of an attacker modifying the line of code right before the legitimate call to `dlopen@plt`, to set the `%rdi` register to point to a string containing a malicious shared library path. In Linux/FreeBSD this method would be somewhat rare because it would require that the process being infected already has a legitimate use of `dlopen` with a corresponding `dlopen@plt` entry. It is certainly difficult to isolate every single edge case of possible infection vectors, but its good to be aware of what's possible, even if unlikely.

Example of suspicious dlopen call

Assuming that the address of `__libc_dlopen_mode` in `libc.so` is `0x7f3e444a39f0`, then the shellcode will use some type of branch instruction to transfer execution to that location. Typically the branch instruction used will not be an immediate call instruction. It is a common practice for shellcode of this nature to use an *indirect call* or *indirect jump* because the instruction that moves the target address into a register can have the source operand bytes be all zeroes until the target address is learned, at which time the zeroes will be replaced with the target address. This allows a shellcode template to exist before the address is known. The shellcode will have to specify a string path name for the shared library it is loading and will likely use a `call; pop` trick to get the address of the string, which is another dead give away since the code generated by the compiler would be using either a hard coded address or IP-relative addressing to reference a string.

Shellcode before its patched with target address

```
 jmp B
A:
pop %rdi ; pop the address of string into first argument
mov $0x80000000, %rsi ; RTLD_DLOPEN flag
movabs $0x000000000000, %rcx ; will contain the address of __libc_dlopen_mode
call *%rcx
B:
call A ; will push the address of string onto the stack
.string "evil_lib.so"
```

The main thing to be aware of is that a suspicious call to `dlopen` will be one that **doesn't go through the PLT**.

An example of backtrace detecting an injected shared object

Saruman PIE injection

The Saruman tool is a prototype which allows the user to inject a PIE executable into an existing process address space using thread injection. That is to say that a user can inject an entire dynamic linked PIE executable into an existing process, and it will run concurrently alongside the existing process from a spawned thread. This is an anti-forensics technique that would typically be difficult to detect. Using the techniques that have been discussed throughout this blog post we are able to quickly identify the injected executable object.

- **NOTE**: Remember that PIE executables are `ET_DYN` (dynamic objects) just like shared libraries, so the techniques discussed will also find injected PIE executables since they are essentially the same thing as shared object files.*

In our example we will inject a remote backdoor called `./backdoor` into an existing process called `./host` which simply `printf's` a string `"I am a host"` in a loop. We will then demonstrate after injection that the remote backdoor is available for accepting login, and commands from the attacker.

Terminal 1 (Start the host)

```
 $ ./host
I am a host
I am a host
I am a host
I am a host
I am a host
```

Terminal 2 (Inject our PIE executable backdoor)

```
 # ./saruman `pidof host` ./backdoor
[+] Thread injection succeeded, tid: 3425
```

```
[+] Saruman successfully injected program: ./backdoor
[+] PT_DETACHED -> 3419
```

Terminal 3 (Observe backdoor)

Notice that there are two pid's for ./host, one for the original program and now another for the injected thread that is the backdoor program.

```
 $ ps auxw | grep './host'
elfmast+ 3419 100 0.0 6504 1332 pts/2 R+ 12:35 2:22 ./host
elfmast+ 3425 0.0 0.0 6504 1332 pts/2 S+ 12:35 0:00 ./host
```

Telnet to port 31337 to connect to the backdoor program that is running stealth within the apparent `./host` address space.

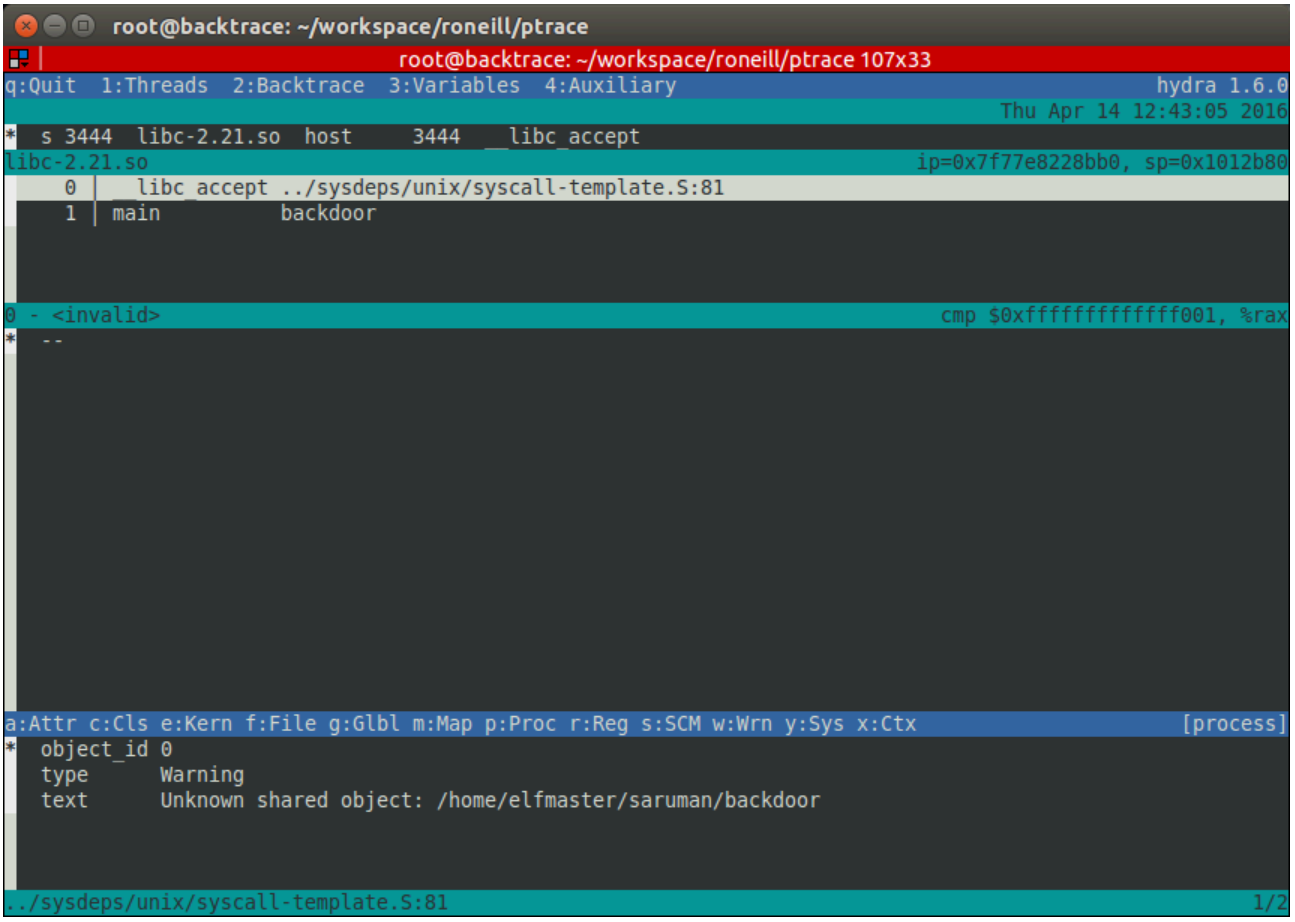
```
  $ telnet localhost 31337
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Password: password
```

Welcome to the Backdoor server! Type 'HELP' for a list of commands

command:~#

Using Backtrace technology to detect the infected process

```
 $ ptrace --module=security:enable,true `pidof host`
/home/elfmaster/host.34444.1460662985.btt
$ hydra host.3444.1460662985.btt
```



Notice in the bottom pane where it says “Unknown shared object”. It has identified the parasite code within the process. As a side note, you may see in the second pane, that the syscall `accept` is present since we had just logged into the backdoor with telnet.

Summary

In this blog post we covered the details of shared library injection. Stay tuned for future posts describing some of the other exploitation and malware techniques used in UNIX-flavor operating systems and how Backtrace detects them.

Edits

- 04/27/2016 – Updated the article to reflect the information given to me by Shawn Webb, that the `dlopen` function in `libc.so` is a fake, which aims to foil attackers who are trying to use it for shared library injection.
- 04/27/2016 – Added credit and a link to the classic 2003 phrack paper titled “Cerberus ELF interface” by Mayhem. This article covers some of the most innovative methods for ELF infection that have been explored to date.
