Running ELF executables from memory

guitmz.com/running-elf-from-memory

Guilherme Thomazi March 27, 2019

7 minute read Published: 27 Mar, 2019

> Executing ELF binary files from memory with memfd_create syscall

Something that always fascinated me was running code directly from memory. From Process Hollowing (aka RunPE) to `PTRACE` injection. I had some success playing around with it in `C` in the past, without using any of the previous mentioned methods, but unfortunately the code is lost somewhere in the forums of `VXHeavens` (sadly no longer online) but the code was buggy and worked only with Linux 32bit systems (I wish I knew about shm_open back then, which is sort of an alternative for the syscall we are using in this post, mainly targeting older systems where `memfd_create` is not available).

Overview and code

Recently, I have been trying to code in `assembly` a bit, I find it very interesting and I believe every developer should understand at least the basics of it. I chose FASM as my assembler because I think it is very simple, powerful and I like its concepts (like same source, same output). More information about its design can be found here. Anyway, I have written a small tool, `memrun`, that allows you to run ELF files from memory using the memfd_create syscall, which is available in Linux where kernel version is `>= 3.17`.

What happens with `memfd_create` is that it acts like `malloc` syscall but will return a file descriptor that references an anonymous file (which does not exists in the disk) and we can pass it to `execve` and execute it from memory. There are a couple in-depth articles about it around the internet already so I will not get too deep into it. A nice one by `magisterquis` can be found at his page

The assembly code might look too big but there are some things we need to take care in this case that we don't need to when writing in a HLL like `Go` (as you can see in its example below). Also it's nice if you want to use the code for an exploit, you can just adjust the assembly instructions to your needs. Both examples are for `x86_64` only:

```
format ELF64 executable 3

include "struct.inc"
include "utils.inc"

segment readable executable
entry start

start:
;-----------------------------------------------------------------------------
; parsing command line arguments
;-----------------------------------------------------------------------------
  pop   rcx                       ; arg count
  cmp   rcx, 3                    ; needs to be at least two for the self program
arg0 and target arg1
  jne   usage                     ; exit 1 if not

  add   rsp, 8                    ; skips arg0
  pop   rsi                       ; gets arg1

  mov   rdi, sourcePath
  push  rsi                       ; save rsi
  push  rdi
  call  strToVar

  pop   rsi                       ; restore rsi
  pop   rdi
  mov   rdi, targetProcessName
  pop   rsi                       ; gets arg2
  push  rdi
  call  strToVar
;-----------------------------------------------------------------------------
; opening source file for reading
;-----------------------------------------------------------------------------
  mov   rdi, sourcePath           ; loads sourcePath to rdi
  xor   rsi, rsi                  ; cleans rsi so open syscall doesnt try to use it
as argument
  mov   rdx, O_RDONLY             ; O_RDONLY
  mov   rax, SYS_OPEN             ; open
  syscall                         ; rax contains source fd (3)
  push  rax                       ; saving rax with source fd
;-----------------------------------------------------------------------------
; getting source file information to fstat struct
;-----------------------------------------------------------------------------
  mov   rdi, rax                  ; load rax (source fd = 3) to rdi
  lea   rsi, [fstat]              ; load fstat struct to rsi
  mov   rax, SYS_FSTAT            ; sys_fstat
  syscall                         ; fstat struct conntains file information
  mov   r12, qword[rsi + 48]      ; r12 contains file size in bytes (fstat.st_size)
;-----------------------------------------------------------------------------
; creating memory map for source file
;-----------------------------------------------------------------------------
  pop   rax                        ; restore rax containing source fd
  mov   r8, rax                    ; load r8 with source fd from rax
  mov   rax, SYS_MMAP              ; mmap number
```

```
  mov   rdi, 0                     ; operating system will choose mapping destination
  mov   rsi, r12                   ; load rsi with page size from fstat.st_size in
r12
  mov   rdx, 0x1                   ; new memory region will be marked read only
  mov   r10, 0x2                   ; pages will not be shared
  mov   r9, 0                      ; offset inside test.txt
  syscall                          ; now rax will point to mapped location
  push  rax                        ; saving rax with mmap address
;-------------------------------------------------------------------------------
; close source file
;-------------------------------------------------------------------------------
  mov   rdi, r8                   ; load rdi with source fd from r8
  mov   rax, SYS_CLOSE            ; close source fd
  syscall
;-------------------------------------------------------------------------------
; creating memory fd with empty name ("")
;-------------------------------------------------------------------------------
  lea   rdi, [bogusName]          ; empty string
  mov   rsi, MFD_CLOEXEC          ; memfd mode
  mov   rax, SYS_MEMFD_CREATE
  syscall                         ; memfd_create
  mov   rbx, rax                  ; memfd fd from rax to rbx
;-------------------------------------------------------------------------------
; writing memory map (source file) content to memory fd
;-------------------------------------------------------------------------------
  pop   rax                       ; restoring rax with mmap address
  mov   rdx, r12                  ; rdx contains fstat.st_size from r12
  mov   rsi, rax                  ; load rsi with mmap address
  mov   rdi, rbx                  ; load memfd fd from rbx into rdi
  mov   rax, SYS_WRITE            ; write buf to memfd fd
  syscall
;-------------------------------------------------------------------------------
; executing memory fd with targetProcessName
;-------------------------------------------------------------------------------
  xor   rdx, rdx
  lea   rsi, [argv]
  lea   rdi, [fdPath]
  mov   rax, SYS_EXECVE           ; execve the memfd fd in memory
  syscall
;-------------------------------------------------------------------------------
; exit normally if everything works as expected
;-------------------------------------------------------------------------------
  jmp   normal_exit
;-------------------------------------------------------------------------------
; initialized data
;-------------------------------------------------------------------------------
segment readable writable
fstat             STAT
usageMsg          db "Usage: memrun <path_to_elf_file> <process_name>", 0xA, 0
sourcePath        db 256 dup 0
targetProcessName db 256 dup 0
bogusName         db "", 0
fdPath            db "/proc/self/fd/3", 0
argv              dd targetProcessName
```

```go
package main

import (
	"fmt"
	"io/ioutil"
	"os"
	"syscall"
	"unsafe"
)

// the constant values below are valid for x86_64
const (
	mfdCloexec  = 0x0001
	memfdCreate = 319
)

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)
}

func main() {
	lenArgs := len(os.Args)
	if lenArgs < 3 || lenArgs > 3 {
		fmt.Println("Usage: memrun process_name elf_binary")
		os.Exit(1)
	}

	runFromMemory(os.Args[1], os.Args[2])
}
```

The full code for both versions can be found in this repo:
https://github.com/guitmz/memrun

See it in action

Allow me to show it in action. Let's start by creating a simple target file in `C`, named `target.c`. The file will try to open itself for reading and if it can't, it will print a message forever every 5 seconds. We will execute it from memory:

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

int main(int argc, char **argv)
{
  printf("My process ID : %d\n", getpid());

  FILE *myself = fopen(argv[0], "r");
  if (myself == NULL) {
        while(1) {
                printf("I can't find myself, I must be running from memory!\n");
                sleep(5);
        }
  } else {
        printf("I am just a regular boring file being executed from the disk...\n");
  }

  return 0;
}
```

Now we build `target.c` :

```
$ gcc target.c -o target
```

We should also build our `FASM` or `Go` tool, I will use the assembly one here:

```
$ fasm memrun.asm
flat assembler  version 1.73.04  (16384 kilobytes memory, x64)
4 passes, 1221 bytes.
```

Running the file normally gives us this:

```
$ ./target
My process ID : 4944
I am just a regular boring file being executed from the disk...
```

But using `memrun` to run it will be totally different:

```
$ ./memrun target MASTER_HACKER_PROCESS_NAME_1337
My process ID : 4945
I can't find myself, I must be running from memory!
I can't find myself, I must be running from memory!
```

Furthermore, if you look for its pid with `ps` utility, this is what you get:

```
$ ps -f 4945
UID        PID  PPID  C STIME TTY      STAT   TIME CMD
guitmz    4945  4842  0 15:31 pts/0    S+     0:00 MASTER_HACKER_PROCESS_NAME_1337
```

Finally, let's check the process directory:

```
$ ls -l /proc/4945/{cwd,exe}
lrwxrwxrwx 1 guitmz guitmz 0 Mar 27 15:38 /proc/4945/cwd ->
/home/guitmz/memrun/assembly
lrwxrwxrwx 1 guitmz guitmz 0 Mar 27 15:38 /proc/4945/exe -> /memfd: (deleted)
```

Note the `/memfd: (deleted)` part, no actual file in disk for this process :)

For those who know, this can be an interesting technique to run stealthy binaries in Linux, you can go even further by giving it a proper name (like a real Linux process) and detach it from the `tty` and change its `cwd` with some simple approches. Tip: `fork` is your friend :)

TMZ