21:09 An ELF Palindrome for AMD64

by Netspooky

The first Binary Golf Grand Prix was a challenge issued on Twitter to create a small binary that executed the same forwards as it did backwards. Included were certain rules, such as ensuring execution past the halfway point in the binary, and that scores would be based on the ratio of overall number of bytes executed to total bytes in the file.

The binary I chose to target was a 64-bit ELF binary, due to my familiarity with creating weird ELFs. I began investigating strategies for creating a palindromic binary in this format because there are quite a few sensitive areas that must remain intact for a binary to run at all.

Initial Efforts

I had already established a baseline of a barebones golf'd 64-bit ELF, and my previous attempts to produce the smallest 64-bit ELF yielded a binary that was 84 bytes in size. I chose this as my starting point.

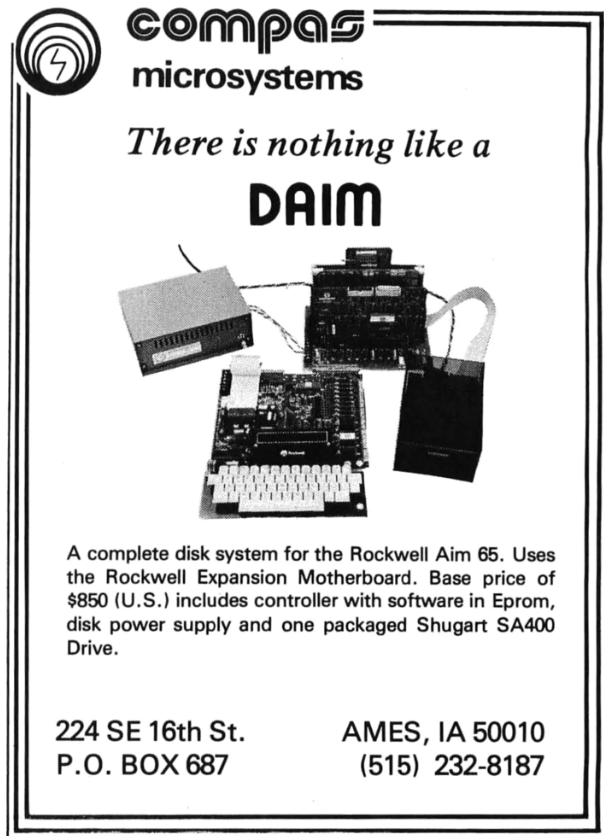
Since I used `nasm` to create ELF files, I began by first flipping the entire source code backwards after the end of my existing source code, and meticulously placing bytes in the correct order. After I finished, I used a Perl one-liner to flip the binary backwards, then executed both binaries and compared their hashes to validate my work.

The next stop was to create a payload that would be both valid, and easy to work with in both directions. My first idea was to use alphanumeric shellcode, as outlined in Phrack 57:15,[19] to have a series of single byte instructions that would also display a palindrome in the hex dump output. The issue with this approach is that alphanumeric shellcode is based on 32-bit x86, which wouldn't work to run on 64-bit Linux.

I also wanted my palindrome to be readable, and since palindromes tend to rely on the ambiguity of punctuation to work, my palindrome would have to use words that could be read if presented as a single string of alphanumeric characters. I decided to go with the phrase "`PULLUPIFIPULLUP`," because it was readable. Testing this in a disassembler showed that certain characters would not be valid machine code.

I tested all of the alphanumeric characters in a disassembler and realized that even fewer characters are usable than in 32-bit mode. This is due to prefix instructions taking the place of references to smaller registers, and certain encodings changing. These were the characters that were safest to use:

Op	Instruction	Char
50	`push rax`	P
51	`push rcx`	Q
52	`push rdx`	R
53	`push rbx`	S
54	`push rsp`	T
55	`push rbp`	U
56	`push rsi`	V
57	`push rdi`	W
58	`pop rax`	X
59	`pop rcx`	Y
5a	`pop rdx`	Z



[19] `unzip pocorgtfo21.pdf phrack5715.txt`

Luckily, there are vowel sounds that can be used to find some words and write my own palindrome. An online Scrabble word finder came in handy for this. After searching for words to use, I ended up with the phrase "`PUPPY SPY, PSY P. PUP`".

The nice thing about these particular instructions is that they are `push` and `pop` instructions, so you don't have to worry too much about messing up data that might be in these registers, and just have to track where values might end up if you use them at all.

Mirroring

The template 64-bit ELF source only executes seven bytes to perform the exit syscall:

```
0:  b0 3c          mov    al,0x3c
2:  48 31 ff       xor    rdi,rdi
5:  0f 05          syscall
```

What was particularly interesting was that when reversed, the bytes are actual usable instructions.

```
0:  05 0f ff 31 48    add    eax,0x4831ff0f
5:  3c b0             cmp    al,0xb0
```

This was a very lucky discovery, and I started thinking even more about interpreting instructions backwards. One of the challenges in something like this is that x86 has variable length instructions, and using bigger registers with smaller values gives a lot of null bytes to contend with. This means that carefully planning certain instructions of the basic operations I wanted to do was next on my list.

There is quite a lot of variance in both assemblers and disassemblers in generating and reading code, so ensuring that the source is assembled properly is of utmost importance. I ended up only using `nasm` and `ndisasm` to verify that instructions were what I wanted them to be.

Now that I had some ideas, I started padding out the remaining sections that might contain code with `nop`s, so that at the very least, I had some wiggle room when calculating things like jumps. Since the code began at offset `0x4` in the header, padding with five `nop`s filled the rest of the space to offset `0xF`.

Getting an idea of how to use jumps was the next thing to sort out. I figured that `jmp` instructions could be accounted for in one of two ways: either a pairing of `jmp`s that jump over each other, or a `jmp` that is interpreted as something else backward.

I wrote a small script to generate all of the possible opcode combinations for short jumps and what they disassemble to when interpreted backwards. Even though it's only two bytes, `EB` and the one byte `jmp` distance, there are a lot of incompatible instructions, such as references to `EBP` and other registers that aren't easily usable in x64.

```python
import sys
import subprocess

# python3 opiter.py opcode
# Will iter through one byte in front of the
# opcode you put in there.  It's hella
# bespoke, feel free to change heh

exp = sys.argv[1]

for i in range(0,255):
  opp = format(i, '02x')
  inf = '"'+opp+' '+exp+'"'
  print(opp+" "+exp+" |",end=" ")
  process = subprocess.run(
    ['/usr/bin/rasm2 -a x86 -b 64 -d '+inf],
    shell=True, check=True,
    stdout=subprocess.PIPE,
    universal_newlines=True)
  output = process.stdout
  if 'invalid' in output:
    print("--")
  else:
    print(output,end="")
```

This is the output from the `jmp` bruteforce table with invalid opcodes ignored:

```
00 eb   add bl,  ch      | 2c eb   sub    al,  0xeb
01 eb   add ebx, ebp     | 30 eb   xor    bl,  ch
02 eb   add ch,  bl      | 31 eb   xor    ebx, ebp
03 eb   add ebp, ebx     | 32 eb   xor    ch,  bl
04 eb   add al,  0xeb    | 33 eb   xor    ebp, ebx
08 eb   or  bl,  ch      | 34 eb   xor    al,  0xeb
09 eb   or  ebx, ebp     | 38 eb   cmp    bl,  ch
0a eb   or  ch,  bl      | 39 eb   cmp    ebx, ebp
0b eb   or  ebp, ebx     | 3a eb   cmp    ch,  bl
0c eb   or  al,  0xeb    | 3b eb   cmp    ebp, ebx
10 eb   adc bl,  ch      | 3c eb   cmp    al,  0xeb
11 eb   adc ebx, ebp     | 63 eb   movsxd rbp, ebx
12 eb   adc ch,  bl      | 6a eb   push 0xffffffffffffffeb
13 eb   adc ebp, ebx     | 70 eb   jo   0xffffffffffffffed
14 eb   adc al,  0xeb    | 71 eb   jno  0xffffffffffffffed
18 eb   sbb bl,  ch      | 72 eb   jb   0xffffffffffffffed
19 eb   sbb ebx, ebp     | 73 eb   jae  0xffffffffffffffed
1a eb   sbb ch,  bl      | 74 eb   je   0xffffffffffffffed
1b eb   sbb ebp, ebx     | 75 eb   jne  0xffffffffffffffed
1c eb   sbb al,  0xeb    | 76 eb   jbe  0xffffffffffffffed
20 eb   and bl,  ch      | 77 eb   ja   0xffffffffffffffed
21 eb   and ebx, ebp     | 78 eb   js   0xffffffffffffffed
22 eb   and ch,  bl      | 79 eb   jns  0xffffffffffffffed
23 eb   and ebp, ebx     | 7a eb   jp   0xffffffffffffffed
24 eb   and al,  0xeb    | 7b eb   jnp  0xffffffffffffffed
28 eb   sub bl,  ch      | 7c eb   jl   0xffffffffffffffed
29 eb   sub ebx, ebp     | 7d eb   jge  0xffffffffffffffed
2a eb   sub ch,  bl      | 7e eb   jle  0xffffffffffffffed
2b eb   sub ebp, ebx     | 7f eb   jg   0xffffffffffffffed
```

After generating all of these instructions, I realized that the distance between the code at `0xF` and the corresponding code on the other half of the binary was too great for a short jump. I moved on to the next phase, working out some sort of code to jump to within the main binary. There was another example of tiny code I had used in previous work, a stream covering approach to assembly code optimization called `i2ao`.[20] This code was simple and portable enough to reuse for this application. The code simply printed out a string and exited.

Now, we have a palindrome that works as both code and printable text, all of the possible short jumps, and some basic code to print the string, it was time to put it all together.

Putting it all together

Throughout this, you can refer to both the finished assembly code, and the diagram featuring the full labeled binary. If you are unfamiliar with the ELF format, check out Ange Albertini's Corkami ELF file explanations on Github!

The primary concern with all of this was to make sure that the registers we need are cleared prior to making a syscall, lest we segfault. In this case, there are two calls to make: `write` and `exit`.

The registers required for a `write` syscall are `RAX`, `RDX`, `RDI`, and `RSI`. Since the first instructions executed add values to `RAX`, an explicit `mov rax, 1` is needed, rather than any clever tricks to populate `RAX`. If we wanted to use something like `xor rax, rax; inc rax`, it would add an extra byte. Some other space saving measures are also used in the `write` syscall code, which you can refer to in the `i2ao` writeups or video.

The next step is to reference the string within the code that is immediately after the `write` syscall. There are a few ways of making references to the current offset, but none of them made much sense other than simply knowing where in the binary the string is, and moving that value into the `sil` register. This can be achieved by assembling your binary, and opening in a debugger before executing, to get the exact values needed.

After the `write` syscall code was sorted, it was time to start mirroring the entire executable section. Since the bounds of the headers have already been established, you can safely do this without messing up your binary. Jumps from the main code section back into the reverse header will be determined later.

The `write` syscall code doesn't really have too many instructions that you can safely execute backwards without entirely rewriting it. So instead of that, another approach is to simply jump over whatever wasn't executable. The alphanumeric machine code was placed before the `write` code, so that it could be used as a sled and have a known location. Since this is executable and won't interfere with the flow of the program, a `jmp` can be placed between that and the backwards code for the `write` syscall.

The size of the `write` code, along with the short `jmp`, produces `jmp 0x17`, which turns into `eb15` in machine code. This unfortunately doesn't translate to anything usable backwards. Referring to the `jmp` table, there is a usable instruction `sbb bl, ch`, that can be achieved by padding with three `nop`s to bring the opcode to `18eb` when backward, `eb18` forward. This would create a nice way to both jump over junk code, and still maintain executability in the code.

All of this `jmp` encoding was done mainly to prevent generating even more junk bytes to account for. Another solution would be to just encode a `jmp` instruction backwards, `02eb`, after the `jmp` to the `write` syscall label, which would do a small hop over the `jmp 0x17` that we can't execute backwards. This approach felt cleaner in the end.

Now all we have to do is just execute our string, clear `RAX`, and jump back into the headers. This operation just adds a small, five byte block that we have to account for when we jump out of the headers the first time and into the main code section.

A space saving trick used here was to completely overwrite the `p_align` section in the ELF's program header, saving 16 bytes in total (eight on each side) within the code section.

[20] `unzip pocorgtfo21.pdf i2ao.zip`

Final Optimizations

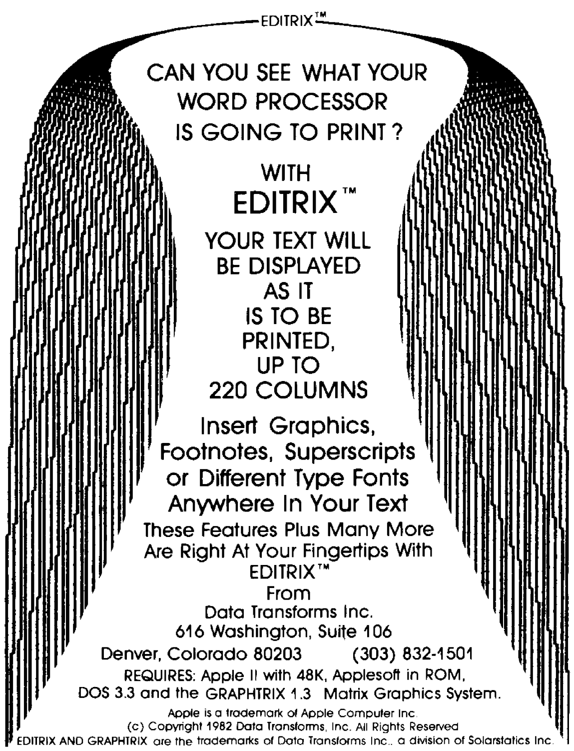
Due to the jump from the header to the main code area, there was junk code from `0x4` to `0x10`, where the binary begins and ends execution. So, a final step was tried to utilize all the space here.

Previous fuzzing of the various headers determined that there is writable space at both `0x3C` and `0x44` in the program header. They must be exactly the same or the binary will not execute. Each of these spots has four bytes of space to work with, which is perfect to do something simple like a short `jmp`.

A short `jmp` from the top of the ELF header at `0x0E` to `0x44`, produces some bytes that are usable backwards! This is `eb34`, which backwards, `34eb` decodes to `xor al, 0xeb`. Since it's only messing with `AL`, the lowest byte of `RAX`, this operation doesn't matter because the value is explicitly assigned afterwards. *Chef's kiss!*

This ensures that when we jump from the main code section, we will be able to use all of the bytes at the end of the binary before the `exit` syscall. Additionally, this increased the total number of executed bytes by four, bringing the grand total to 90 bytes executed out of 245 total.

The final code is shown on page 47.



Confirming Functionality

This was tested and built on Ubuntu 20.04 with kernel 5.4.0-42-generic. Here is a small script you can use to build and test the ASM file, and execution is shown on page 46.

```
#!/bin/bash

nasm -f bin ns.bggp.asm -o ns.bggp
chmod +x ns.bggp
echo "Executing initial binary..."
./ns.bggp
echo ""
xxd ns.bggp
echo ""
echo "Reversing..."
perl -0777pe '$_=reverse $_' ns.bggp > ns.R
chmod +x ns.R
echo "Executing binary in reverse..."
./ns.R
echo ""
xxd ns.R
echo ""
echo "Comparing hashes..."
sha1sum ns.bggp
sha1sum ns.R
```

Final Thoughts

I might've shrunk the `write` syscall code down even more to try and save 1 byte to produce a short `jmp` of `0xeb12->0x12eb (adc ch, bl)`. Since I was coding not just for size, but for percentage of bytes executed as well, it made more sense just to leave things as they were.

It will be exciting to do another challenge like this next time around, and hopefully expand on the competition as a whole. If you'd participate, or have any questions / comments, you can email me at `u@n0.lol` or talk to me on Twitter `@netspooky`.

A special thank you goes to everyone who competed in the first Binary Golf Grand Prix, 0xdade, ThugCrowd and Hermit. :}

```
$ ./build.sh
Executing initial binary...
PUPPYSPYPSYPPUP
00000000: 7f45 4c46 050f ff31 483c b090 9090 eb34  .ELF...1H<.....4
00000010: 0200 3e00 0100 0000 0400 0000 0100 0000  ..>.............
00000020: 1c00 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0100 0000 4000 3800 0100 0200 eb0b 0000  ....@.8.........
00000040: 0000 0000 eb0b 0000 0000 0000 3ceb c031  ............<..1
00000050: 4850 5550 5059 5350 5950 5359 5050 5550  HPUPPYSPYPSYPPUP
00000060: eb18 9090 9090 9005 0f95 b640 20e6 c148  ...........@ ..H
00000070: c689 0fb2 c789 0000 0001 b801 0000 0089  ................
00000080: c7b2 0f89 c648 c1e6 2040 b695 0f05 9090  .....H.. @......
00000090: 9090 9018 eb50 5550 5059 5350 5950 5359  .....PUPPYSPYPSY
000000a0: 5050 5550 4831 c0eb 3c00 0000 0000 000b  PPUPH1..<.......
000000b0: eb00 0000 0000 000b eb00 0200 0100 3800  ..............8.
000000c0: 4000 0000 0100 0000 0000 0000 0000 0000  @...............
000000d0: 0000 0000 1c00 0000 0100 0000 0400 0000  ................
000000e0: 0100 3e00 0234 eb90 9090 b03c 4831 ff0f  ..>..4.....<H1..
000000f0: 0546 4c45 7f                             .FLE.

Reversing...
Executing binary in reverse...
PUPPYSPYPSYPPUP
00000000: 7f45 4c46 050f ff31 483c b090 9090 eb34  .ELF...1H<.....4
00000010: 0200 3e00 0100 0000 0400 0000 0100 0000  ..>.............
00000020: 1c00 0000 0000 0000 0000 0000 0000 0000  ................
00000030: 0100 0000 4000 3800 0100 0200 eb0b 0000  ....@.8.........
00000040: 0000 0000 eb0b 0000 0000 0000 3ceb c031  ............<..1
00000050: 4850 5550 5059 5350 5950 5359 5050 5550  HPUPPYSPYPSYPPUP
00000060: eb18 9090 9090 9005 0f95 b640 20e6 c148  ...........@ ..H
00000070: c689 0fb2 c789 0000 0001 b801 0000 0089  ................
00000080: c7b2 0f89 c648 c1e6 2040 b695 0f05 9090  .....H.. @......
00000090: 9090 9018 eb50 5550 5059 5350 5950 5359  .....PUPPYSPYPSY
000000a0: 5050 5550 4831 c0eb 3c00 0000 0000 000b  PPUPH1..<.......
000000b0: eb00 0000 0000 000b eb00 0200 0100 3800  ..............8.
000000c0: 4000 0000 0100 0000 0000 0000 0000 0000  @...............
000000d0: 0000 0000 1c00 0000 0100 0000 0400 0000  ................
000000e0: 0100 3e00 0234 eb90 9090 b03c 4831 ff0f  ..>..4.....<H1..
000000f0: 0546 4c45 7f                             .FLE.

Comparing hashes...
c082d226c96b7251649c48526dd9766071fa5e59  ns.bggp
c082d226c96b7251649c48526dd9766071fa5e59  ns.bggp.R
```

Figure 11: Executing the palindrome backward and forward.

```
BITS 64
    org 0x100000000      ; Where to load this into memory
;-----------------------+------+--------------+-----------+--------------------------
; ELF Header struct     | OFFS | ELFHDR       | PHDR      | ASSEMBLY OUTPUT
;-----------------------+------+--------------+-----------+--------------------------
        db 0x7F, "ELF"  ; 0x00 | e_ident      |           | 7f 45 4c 46
_start:                 ;      |              |           |
    add eax,0x4831ff0f  ; 0x4  |              |           | 05 0f ff 31 48
    cmp al,0xb0         ; 0x9  |              |           | 3c b0
    nop                 ; 0xB  |              |           | 90
    nop                 ; 0xC  |              |           | 90
    nop                 ; 0xD  |              |           | 90
    jmp hjmp            ; 0xE  |              |           | eb 34
;-----------------------+------+--------------+-----------+--------------------------
; ELF Header struct ct. | OFFS | ELFHDR       | PHDR      | ASSEMBLY OUTPUT
;-----------------------+------+--------------+-----------+--------------------------
    dw 2                ; 0x10 | e_type       |           | 02 00
    dw 0x3e             ; 0x12 | e_machine    |           | 3e 00
    dd 1                ; 0x14 | e_version    |           | 01 00 00 00
    dd _start - $$      ; 0x18 | e_entry      |           | 04 00 00 00
;-----------------------+------+--------------+-----------+--------------------------
; Program Header Begin  | OFFS | ELFHDR       | PHDR      | ASSEMBLY OUTPUT
;-----------------------+------+--------------+-----------+--------------------------
phdr:                   ;      |              |           |
    dd 1                ; 0x1C |   ...        | p_type    | 01 00 00 00
    dd phdr - $$        ; 0x20 | e_phoff      | p_flags   | 1c 00 00 00
    dd 0                ; 0x24 |   ...        | p_offset  | 00 00 00 00
    dd 0                ; 0x28 | e_shoff      |   ...     | 00 00 00 00
    dq $$               ; 0x2C |   ...        | p_vaddr   | 00 00 00 00
                        ; 0x30 | e_flags      |   ...     | 01 00 00 00
    dw 0x40             ; 0x34 | e_shsize     | p_addr    | 40 00
    dw 0x38             ; 0x36 | e_phentsize  |   ...     | 38 00
    dw 1                ; 0x38 | e_phnum      |   ...     | 01 00
    dw 2                ; 0x3A | e_shentsize  |   ...     | 02 00
    ;dq 2               ; 0x3C | e_shnum      | p_filesz  | 02 00 00 00 00 00 00 00
    dw 0x0beb           ; eb 0b ; Overwrites e_shnum and p_filesz
    dw 0
    dd 0
hjmp:
    ;dq 2               ; 0x44 |              | p_memsz   | 02 00 00 00 00 00 00 00
    jmp sec0            ; eb 0b ; Overwrites p_memsz
    dw 0
    dd 0
    ;dq 2               ; 0x4C |              | p_align   | 02 00 00 00 00 00 00 00
;--- Outer bounds of executable portion
    cmp al, 0xeb                ; 3c eb ; Overwrites p_align
    db 0xc0                     ; c0
    db 0x31                     ; 31
    db 0x48                     ; 48
sec0:
    push    rax                 ; 50
    push    rbp                 ; 55
    push    rax                 ; 50
    push    rax                 ; 50
    pop     rcx                 ; 59
    push    rbx                 ; 53
    push    rax                 ; 50
    pop     rcx                 ; 59
    push    rax                 ; 50
    push    rbx                 ; 53
    pop     rcx                 ; 59
    push    rax                 ; 50
    push    rax                 ; 50
    push    rbp                 ; 55
```

```
    push   rax                   ; 50
    jmp sec1                     ; eb 18
    nop                          ; 90
    nop                          ; 90
    nop                          ; 90
    nop                          ; 90
    nop                          ; 90
    add eax,0x40b6950f           ; 05 0f 95 b6 40 ; Third byte is str offset
    and dh,ah                    ; 20 e6
    ror DWORD [rax-0x3a],0x89    ; c1 48 c6 89
    dd 0x89c7b20f                ; 0f b2 c7 89
    add BYTE [rax],al            ; 00 00
    add BYTE [rcx],al            ; 00 01
;--- split - the first byte is shared with the mov rax,1
sec1:
    mov rax, 1                   ; b8 01 00 00 00
    mov edi, eax                 ; 89 c7
    mov dl, 15                   ; b2 0f
    mov esi, eax                 ; 89 c6
    shl rsi, 0x20                ; 48 c1 e6 20
    mov sil, 0x95                ; 40 b6 95
    syscall                      ; 0f 05
    nop                          ; 90
    nop                          ; 90
    nop                          ; 90
    nop                          ; 90
    nop                          ; 90
    sbb bl, ch                   ; 18 eb
sec2:
    push   rax                   ; 50
    push   rbp                   ; 55
    push   rax                   ; 50
    push   rax                   ; 50
    pop    rcx                   ; 59
    push   rbx                   ; 53
    push   rax                   ; 50
    pop    rcx                   ; 59
    push   rax                   ; 50
    push   rbx                   ; 53
    pop    rcx                   ; 59
    push   rax                   ; 50
    push   rax                   ; 50
    push   rbp                   ; 55
    push   rax                   ; 50
    xor rax, rax                 ; 48 31 c0
    jmp rstart                   ; eb 3c
;--- Header Mirror       ; old offset |
    dd 0
    dw 0
    dw 0xeb0b             ; 0x44 |              | p_memsz  | 02 00 00 00 00 00 00 00
    dd 0                  ;
    dw 0                  ;
    dw 0xeb0b             ; 0x3C | e_shnum      | p_filesz | 02 00 00 00 00 00 00 00
    db 0                  ;
    db 2                  ; 0x3A | e_shentsize  |   ...    | 02 00
    db 0                  ;
    db 1                  ; 0x38 | e_phnum      |   ...    | 01 00
    db 0                  ;
    db 0x38               ; 0x36 | e_phentsize  |   ...    | 38 00
    db 0                  ;
    db 0x40               ; 0x34 | e_shsize     | p_addr   | 40 00
    dw 0                  ;
    db 0                  ;
    db 1                  ; 0x30 | e_flags      |   ...    | 01 00 00 00
    dd 0                  ; 0x2C |   ...        | p_vaddr  | 00 00 00 00
```

```
    dd 0            ; 0x28 | e_shoff     |   ...     | 00 00 00 00
    dd 0            ; 0x24 |   ...       | p_offset  | 00 00 00 00
    dw 0            ;
    db 0            ;
    db 0x1c         ; 0x20 | e_phoff     | p_flags   | 1c 00 00 00
    dw 0            ;
    db 0            ;
    db 1            ; 0x1C |   ...       | p_type    | 01 00 00 00
    dw 0            ;
    db 0            ;
    db 4            ; 0x18 | e_entry     |           | 04 00 00 00
    dw 0            ;
    db 0            ;
    db 1            ; 0x14 | e_version   |           | 01 00 00 00
    db 0            ;
    db 0x3e         ; 0x12 | e_machine   |           | 3e 00
    db 0            ;
    db 2            ; 0x10 | e_type      |           | 02 00
rstart:
    xor al, 0xeb    ; 34 EB ; Jmp in reverse
    nop             ; 90
    nop             ; 90
    nop             ; 90
    mov al, 0x3c    ; b0 3c
    xor rdi, rdi    ; 48 31 ff
    syscall         ; 0f 05
    db "F"
    db "L"
    db "E"
    db 0x7F         ; 0x00 | e_ident     |           | 7f 45 4c 46
```