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, 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:

<table>
<thead>
<tr>
<th>Op</th>
<th>Instruction</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>push rax</td>
<td>P</td>
</tr>
<tr>
<td>51</td>
<td>push rcx</td>
<td>Q</td>
</tr>
<tr>
<td>52</td>
<td>push rdx</td>
<td>R</td>
</tr>
<tr>
<td>53</td>
<td>push rbx</td>
<td>S</td>
</tr>
<tr>
<td>54</td>
<td>push rsp</td>
<td>T</td>
</tr>
<tr>
<td>55</td>
<td>push rbp</td>
<td>U</td>
</tr>
<tr>
<td>56</td>
<td>push rsi</td>
<td>V</td>
</tr>
<tr>
<td>57</td>
<td>push rdi</td>
<td>W</td>
</tr>
<tr>
<td>58</td>
<td>pop rax</td>
<td>X</td>
</tr>
<tr>
<td>59</td>
<td>pop rcx</td>
<td>Y</td>
</tr>
<tr>
<td>5a</td>
<td>pop rdx</td>
<td>Z</td>
</tr>
</tbody>
</table>

---

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:

```
1 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 ,0 x4831ff0f
```

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 jmps 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.

```
import sys
import subprocess

# python3 o p i t e r . py opcode
# Will i t e r through one byte in f r o n t 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')
    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, bl
02 eb add ch, bl 31 eb xor ebx, ebp
03 eb add ebp, ebx 32 eb xor bl, bl
04 eb add al, 0xeb 33 eb xor ebp, bl
08 eb or bl, ch 34 eb xor b al, 0xeb
09 eb or ebx, ebp 38 eb cmp bl, bl
0a eb or ch, bl 39 eb cmp bl, ch, bl
0b eb or ebp, ebx 3a eb cmp bl, ch
0c eb or al, 0xeb 3b eb cmp ebp, ebp, bl
10 eb adc ebx, ebp 3c eb cmp al, bl, bl
11 eb adc ebx, ebp 63 eb movedx 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, bl 72 eb jbp 0xffffffffffffffed
19 eb sbb ebx, ebp 73 eb jae 0xffffffffffffffed
1a eb sbb ch, bl 74 eb jae 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 jnp 0xffffffffffffffed
24 eb and al, 0xeb 7b eb jnp 0xffffffffffffffed
28 eb sub bl, ch 7c eb j1 0xffffffffffffffed
29 eb sub ebx, ebp 7d eb jge 0xffffffffffffffed
2a eb sub bl, bl 7e eb jle 0xffffffffffffffed
2b eb sub ebp, ebx 7f eb jg 0xffffffffffffffed
```

This is the output from the jmp bruteforce table with invalid opcodes ignored:
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. 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 having 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 nops 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 -o777pe "$_=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. :}
Figure 11: Executing the palindrome backward and forward.
BITS 64

org 0x1000000000

; Where to load this into memory

ELF Header struct | OFFS | ELFHDR | PHDR | ASSEMBLY OUTPUT

    db 0x7F, "ELF" ; 0x00 | e_ident | 
    add eax, 0x4831ff0f ; 0x04 | e_type | 02 00
    cmp al, 0xb0 ; 0x09 | e_machine | 3e 00
    nop ; 0x0B | e_version | 01 00 00 00
    nop ; 0x0C | e_entry | 04 00 00 00

ELF Header struct ct. | OFFS | ELFHDR | PHDR | ASSEMBLY OUTPUT

    dd 2 ; 0x10 | . . . | p_type | 01 00 00 00
    dd phdr - $$ ; 0x20 | . . . | p_offset | 00 00 00 00
    dq $$ ; 0x2C | . . . | p_vaddr | 00 00 00 00
    dw 0x10 ; 0x34 | . . . | p_addr | 40 00
    dd 1 ; 0x38 ; 0x36 | . . . | p_offset | 38 00

Program Header Begin | OFFS | ELFHDR | PHDR | ASSEMBLY OUTPUT

    dq 2 ; 0x3C | e_shnum | 02 00 00 00 00 00 00 00
    jmp hjmp : eb 0b ; Overwrites e_shnum and p_filesz
    dq 2 ; 0x40 | . . . | p_memsz | 02 00 00 00 00 00 00 00
    jmp sec0 : eb 0b ; Overwrites p_memsz
    dd 0

Program Header Begin ; 0x44 | . . . | p_memsz | 02 00 00 00 00 00 00 00

; Outer bounds of executable portion

    cmp al, 0xeb ; 3c eb ; Overwrites p_align
    db 0x0eb ; 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
    push rax ; 50
    push rax ; 50
    push rbp ; 55
65       push rax         ; 50
66       jmp sec1       ; eb 18
67       nop           ; 90
69       nop           ; 90
70       nop           ; 90
71       nop           ; 90
72       add eax,0x40b6950f ; 05 0f 95 b6 40 ; Third byte is str offset
73 and dh,ah     ; 20 e6
74       ror DWORD [rax−0x3a],0x89 ; c1 48 c6 89
75       dd 0x89c7b20f ; 0f b2 c7 89
76       add BYTE [rax],al ; 00 00
77       add BYTE [rcx],al ; 00 01
78 ;—split—the first byte is shared with the mov rax,1
79 sec1:
80       mov rax, 1  ; b8 01 00 00 00
81       mov edi, eax ; 89 c7
82       mov dl, 15    ; b2 0f
83       shl rsi, eax ; 89 c6
84       mov sil, 0x20 ; 48 c1 e6 20
85       syscall      ; 0f 65
86       nop          ; 90
87       nop          ; 90
88       nop          ; 90
89       nop          ; 90
90       sbb bl, ch  ; 18 eb
91 sec2:
92       push rax     ; 50
93       push rbp     ; 55
94       push rax     ; 50
95       push rax     ; 50
96       pop rcx      ; 59
97       push rbx     ; 53
98       push rax     ; 50
99       pop rcx      ; 59
100      push rax     ; 50
101      push rax     ; 50
102      push rbx     ; 53
103      pop rcx      ; 59
104      push rax     ; 50
105      push rax     ; 50
106      push rax     ; 50
107      push rbp     ; 55
108      push rax     ; 50
109      xor rax, rax ; 48 31 c0
110      jmp rstart   ; eb 3c
111 ;— Header Mirror ; old offset |
112 dd 0
113 dd 0
114 dw 0
115 dd 0
116 dw 0
117 dw 0xexeb0b ; 0x44 | p_memsz | 02 00 00 00 00 00 00 00
118 dd 0
119 dw 0
120 dw 0xexeb0b ; 0x3c | e_shnum | p_filesz | 02 00 00 00 00 00 00 00
121 db 0
122 db 2 ; 0x3a | e_shentsize | ... | 02 00
123 db 1 ; 0x38 | e_phnum | ... | 01 00
124 db 0
125 db 0x38 ; 0x36 | e_phentsize | ... | 38 00
126 db 0
127 db 0x40 ; 0x34 | e_shsize | p_addr | 40 00
128 dw 0
129 db 0 ; 0x30 | e_flags | ... | 01 00 00 00
130 dd 0 ; 0x2c | ... | p_vaddr | 00 00 00 00

48
```
restart:  
xor al, 0xeb; 34 EB; Jmp in reverse
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
```