21:10 BootNoodle: A Palindromic Bootloader for BGGP

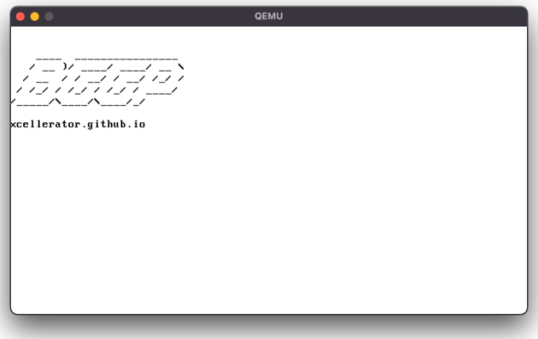
by Harvey Phillips

Recently `@netspooky` announced the first annual Binary Golf Grand Prix on Twitter. The objective was to create a binary of any sort that is the same forwards as it is byte-reversed, but with an emphasis on creating as small a binary as possible, hence *golfing*.

This was one of those challenges where I thought that I had no chance of creating a qualifying submission, where it might be better to just wait for the results and admire the work of others. However, it wasn't long until I found myself thinking about how it would even be possible to create such a binary. Clearly, executables that are just pure x86 instructions (like COM files) wouldn't count; otherwise, I could've just submitted `0x90` and been done!

I decided that if I was going to attempt something like this, I'd have to first settle on a file format. In the end, I think I took the easy option and chose to create an x86 bootloader palindrome. The main reasons for this were that bootloaders are essentially formatless: the only requirement for a valid bootloader is that bytes at offset 511 and 512 are `0x55` and `0xAA` respectively. The rest can be just raw x86 instructions.

That brings us to the second reason: technically (as far as I am aware) the absolute minimum size of an x86 bootloader is 512 bytes. This felt like a bit of a double-edged sword, just enough space to do something interesting, but still fairly limited. Especially since it has to read the same backward!



[21] `http://xcellerator.github.io`
[22] `http://www.ctyme.com/intr/int.htm`

Workflow

I knew that the first thing I had to get right was generating a palindromic file, whether or not anything really executed. The bootloader itself was going to be written in NASM, so I could then just use `dd` to snip off the first 256 bytes, reverse it with a bit of Perl from StackOverflow and `cat` the two halves together. I stuck all this into a shell script and started to get to work.

Once this is complete, we can run the bootloader with `qemu-system-x86_64 bootnoodle.bin`.

```
nasm -f bin -o bootnoodle.bin bootnoodle.asm
dd if=bootnoodle.bin of=tmp.bin \
   bs=1 count=256
rm bootnoodle.bin
perl -0777pe '$_=reverse $_'    \
   tmp.bin >tmp2.bin
cat tmp.bin tmp2.bin >bootnoodle.bin
rm tmp*
```

x86 Bootloaders

Creating an x86 bootloader is a surprising easy task. After fighting with a few different ideas for what to do, I settled on printing a nice bit of "BGGP" ASCII art and a link to my blog, where this write-up first appeared.[21]

You might wonder how on Earth you might fit a printing function into just 256 bytes. It turns out that a huge amount of functionality, from printing characters to graphics primitives, are built into the BIOS. In our specific case, we'll be targeting the SeaBIOS that ships with QEMU. We call these built-in functions by selecting the byte we place into the `AH` register before invoking a particular interrupt.

For example, we can call the Teletype Output routine to print a character by placing `0x0E` into `AH`, the ASCII character we want to print into `AL` and calling interrupt `0x10`. There are couple of extra options to this function, like a page number and foreground colour, that we can put in to the `BX` register as well. In general, this is the flow of a bootloader; load registers, interrupt, load registers, interrupt, etc. We can find an exhaustive list of all these dif-

ferent routines and which registers are used at the infinitely useful x86 Interrupt Table.[22]

The rough flow of execution of my bootloader is as follows:

- Clear the screen with the "scroll up window" routine (`Int 10/AH=06h`).
- Set the cursor to the position we want to start printing from with the "set cursor position" routine (`Int 10/AH=02h`).
- Load the memory address of our null-terminated string into the `SI` register.
- Call the string printing routine.
- Halt

The reason we have to clear the screen is that otherwise we'd have fragments of information about the BIOS cluttering the screen up. In QEMU's case, you get the SeaBIOS copyright string stuck at the top of the screen, so for the sake of a few extra bytes, its nicer to clear that out. Also, one of the extra options we get with "scroll up window" is that we can change the background/foreground colour by setting `BH`. I opted for `0x03`, which keeps the background black but makes the foreground cyan.

The only thing left as far as the actual programming goes is the `printString` function. The BIOS provides us with only a character printing function, so we have to handle the looping logic and checking for a null byte ourselves. This is all pretty standard stuff if you've done x86 assembly programming before.

```
printString:
  pusha
  .loop:
    lodsb
    test al, al
    jz .end
    call printChar
    call delay
  jmp .loop
  .end:
    popa
    ret
```

First, we push all the registers onto the stack with `pusha`. Entering the loop, we use `lodsb` which loads a single byte from the `SI` register into `AL`, and increments `SI`. We check for a NULL byte with `test al, al`, and if found, jump to `.end` where we restore the saved registers with `popa` and return. If we don't have a NULL byte in `AL`, then we call the `printChar` and `delay` routines. These are less interesting and very similar to the `clearScreen` and `setCursor` routines: set some registers, interrupt.

The last thing worth mentioning is why we have the call to `delay`, which uses the "wait" routine.[23] The reason is simple: introducing a 20ms delay between printing each character results in a poor man's animation effect!

There is still one thing that we're forgetting. Earlier, I mentioned that a bootloader, while being exactly 512 bytes long, *must* finish with bytes `0x55 0xAA`. Because we're creating a palindrome, this means that our binary must *start* with `0xAA 0x55`. Execution starts at offset 0, so we cannot avoid executing these two bytes as the first instructions.

```
aa ;  stosb es:di
55 ;  push bp
```

The `stosb` instruction is similar to `lodsb`; it stores whatever is in `AL` in `DI`, ignoring segment registers as they aren't really relevant to this discussion. We don't care about this because, (1) we aren't using `DI` and (2) we're about to load `AL` with the address of our string, so whatever happens to be loaded in `AL` beforehand is irrelevant to us. (It's probably just a null byte, but that might vary from BIOS to BIOS.)

Clearly, `push bp` also doesn't matter to us. In theory, we should clean up the stack later by `pop`ing `bp`, but seeing as we're halting into infinite loop with the `jmp $` instruction after printing our string, it really doesn't matter either.

So, thankfully we don't need to worry about these two extra instructions. Merely starting the source file with `db 0xaa, 0x55` before going straight into the `_start` entrypoint is enough to get us out of trouble.

[23] `AH=86h, CX:DX` = interval in microseconds.

[24] `https://n0.lol/bggp`

Palindrome Time

So far, we've only used up `0xf5` bytes of the `0x1ff` available to us. This means that when we run our build script, we end up with a 512-byte binary that's reflected about the 256-byte boundary, with a patch of 20 NULL bytes positioned neatly in the middle. While we technically have a palindrome, Netspooky is way ahead of us, as can be seen by the stipulation in the rules on the contest page.[24]

> An easy solution would be to just have the binary end, and append the binary backwards at the end of the original file. Because of this, in order to qualify for entry, your binary must at a minimum execute $>$ 50% of the bytes in your binary, and must execute past the halfway mark in your binary as well.

So far, we just about meet the 50%+ byte execution mark thanks to the `0xAA 0x55` bytes at the very beginning. Unfortunately, we don't yet execute past the halfway mark, so we've got to do some thinking.

We've still got to do something a little more interesting than just producing a bootloader in under 256 bytes and flipping it back on itself. There's not a huge amount we can do about the data part of the binary (which makes up about 63% of all the bytes) unless the text itself is symmetric, which it isn't. Besides, the rule above specifically mentions that *execution* has to pass the 50% mark. That leaves us to look at what can be done with the code.

My idea was to purposefully reverse portions of the code in the upper half, so that they are re-reversed in the lower half. This means that I also need to fix the `call` offsets manually because NASM won't be able to calculate them for me.

I used Ghidra as a disassembler, but you might want to use `objdump` as a slimmer alternative.[25] Ghidra makes it easy to compare the NASM source with the disassembled bytes. Because my routines are all quite short, I just wrote in the bytes next to the functions that I wanted to reverse. These were chosen alternately so the execution jumps around as much as possible, and lives up to it's *noodly* name. For example, `clearScreen` looks like this.

```
clearScreen:
    pusha                   ;  60
    mov ah, 0x06            ;  b4 06
    xor al, al              ;  30 c0
    mov bh, 0x03            ;  b7 03
    xor cx, cx              ;  31 c9
    mov dx, 0x184f          ;  ba 4f 18
    int 0x10                ;  cd 10
    popa                    ;  61
    ret                     ;  c3
```

We could have worked this out without Ghidra and just used a hex editor, but hey, Ghidra is faster and takes out the guesswork. We can replace this with the raw bytes, but in reverse order:

```
clearScreen db 0xc3, 0x61, 0x10, 0xcd, 0x18,
               0x4f, 0xba, 0xc9, 0x31, 0x03,
               0xb7, 0xc0, 0x30, 0x06, 0xb4,
               0x60
```

But we also have to take a look at the `_start` entrypoint where we call `clearScreen` by name. This clearly will no longer work once we comment out `clearScreen` in favour of the reversed bytes above. You can try running the build script but NASM will exit out with a load of errors.

The solution here is that we need to replace `call clearScreen` with a raw short call. As explained on Felix Cloutier's x86 instruction reference, a short (or "near") call is a call to a memory address *relative to the next instruction*, where a `ret` is expected to be encountered eventually.[26] This means that we can replace the line `call clearScreen` with a simple `db 0xe8, 0x00, 0x00`. This won't work yet because we haven't specified an offset, but it will let us assemble the binary and look at some bytes.

After building, we get a binary that we can disassemble again. Even after picking "x86 Real Mode" from the list of languages in Ghidra, we're left without very much. Clicking on the first byte `0xaa` and pressing `D` kicks off the disassembly.

After the two bogus instructions caused by the reversed `0x55 0xaa` signature, we immediately get two `call`s. These are to `clearScreen` and `setCursor` which appear at the top of the source file! In particular, note that the first `call` is to `0x00 0x00`; this is what we need to change.

[25] `objdump -D -b binary -mi386 -M intel,addr16 bootnoodle.bin`
[26] `https://www.felixcloutier.com/x86/call`

In order to know which offset to set this to, we need the address of the instruction *after* this `call`, which is the `call` to `setCursor` at `0x5`, and the address of the re-reversed `clearScreen` routine. Scrolling through the disassembly, once we cross the half-way point, Ghidra doesn't know what it's looking at. Keep going, and eventually, towards the end, you'll find another bit of disassembled code. Checking it carefully, you'll see that it matches perfectly with the disassembly of `clearScreen` above! The address of this routine is `0x1e0`. Subtracting `0x5` from this gives `0x1db`, the relative offset that we need to set our hand-made `call` instruction to!

Going back to the source file, we change `db 0xe8, 0x00, 0x00` to `db 0xe8, 0xdb, 0x01`. (Remember that x86 is little-endian!) Rebuilding gives us a working bootloader.

I repeated this trick for the `printChar` routine using the exact same steps as for `clearScreen`: reverse the bytes by hand, replace any `call`s to `printChar` with `e8 00 00`, fire up Ghidra to calculate the correct relative memory address, and fix the `call` by hand again.

For fun I reversed the data in the binary, too. This meant that I had to fix the line in `_start` that loads the address of the data into `SI`. This was done by reversing the data and rebuilding. (It builds fine because the code is unaffected; running it will just print the string backwards.) Then, using a hex editor, I looked for the start of the re-reversed string, and found it at `0x10a`.

The instruction for `mov`ing into `SI` is `0xbe` followed by a memory location or register, as described back in the x86 Instruction Reference.[27] However, there is one caveat and it involves the very first line of the source file. Here, I've put `org 0x7c00`, which tells NASM that we are expecting our bootloader to be loaded to memory address `0x7c00` before being executed.

The reason for this seemingly arcane choice of load address is that 1024 bytes after `0x7c00` is `0x8000`, which is where the kernel is normally loaded. The usual purpose of a bootloader is to simply load the kernel from a hard disk (or other storage device) into memory address `0x8000` and then `jmp` to it. Seeing as a bootloader has to be 512 bytes in size, it makes sense to always load it in the memory region immediately prior to where the kernel will be copied to.

For a nicely commented example of loading the kernel into memory and passing execution to it, check out my ThugBoot project.[28] If you've been following this article, then you shouldn't have any issue reading the NASM source there. For a more in-depth read, `@0xax`'s incredible book Linux Insides, which goes into infinitely more detail.[29]

Anyway, all this means for us is that we have to add `0x7c00` to the address within the file of the re-reversed string we want to print, so we end up with a final address of `0x7d0a`, and our manual `mov` instruction becomes `db 0xbe, 0x0a, 0x7d`.

And with that, the palindromic bootloader is done! Source code is available by github, attached to this PDF and on page 54.[30]

I'd like to thank the good folks at ThugCrowd for being so encouraging and inspirational. It was there that I first discovered my interest in exploring x86 bootloaders that lead to the ThugBoot project. I'd also like to thank Netspooky in particular for starting this competition and I highly recommend taking part next year!

[27] `https://www.felixcloutier.com/x86/movsx:movsxd`
[28] `https://github.com/xcellerator/thugboot`
[29] `https://0xax.gitbooks.io/linux-insides/`
[30] `git clone https://github.com/xcellerator/bootnoodle || unzip pocorgtfo21.pdf`

```
; BootNoodle
; A Palindromic Bootloader for the
; Binary Golf Grand Prix
; github.com/xcellerator/bootnoodle

org 0x7C00
bits 16

db 0xaa, 0x55

_start:
  db 0xe8, 0xdb, 0x1    ;  call clearScreen
  call setCursor
  db 0xbe, 0x0a, 0x7d   ;  mov si, msg
  call printString
  jmp $

;clearScreen:
;  pusha                ;  60
;  mov ah, 0x06         ;  b4 06
;  xor al, al           ;  30 c0
;  mov bh, 0x03         ;  b7 03
;  xor cx, cx           ;  31 c9
;  mov dx, 0x184F       ;  ba 4f 18
;  int 0x10             ;  cd 10
;  popa                 ;  61
;  ret                  ;  c3

clearScreen db 0xc3, 0x61, 0x10, 0xcd, 0x18,
               0x4f, 0xba, 0xc9, 0x31, 0x03,
               0xb7, 0xc0, 0x30, 0x06, 0xb4,
               0x60

setCursor:
  pusha
  mov ah, 0x02
  mov bh, 0x00
  mov dh, 2
  mov dl, 0
  int 0x10
  popa
  ret

;printChar:
;  mov ah, 0x0e      ;  b4 0e
;  mov bh, 0x0       ;  b7 00
;  mov bl, 0x0       ;  b3 00
;  int 0x10          ;  cd 10
;  ret               ;  c3

printChar db 0xc3, 0x10, 0xcd, 0x00, 0xb3,
             0x00, 0xb7, 0x0e, 0xb4

printString:
  pusha
  .loop:
    lodsb
    test al, al
    jz .end
    ;call printChar
    db 0xe8, 0x8b, 0x01   ; call printChar
    call delay
  jmp .loop
  .end:
    popa
    ret

delay:
  pusha
  mov ah, 0x86
  mov al, 0
  mov cx, 0
  mov dx, 20
  int 0x15
  popa
  ret

msg db 0x0, 'oi.buhtig.rotarellecx', 0xa,
  0xd, 0xa, 0xd, '/_/____\/____\/_____/',
  0xa, 0xd,  '/____ / /_/ / /_/ / /_/ / ',
  0xa, 0xd, '/ /_/ /__ / /__ / /  __ /  ',
  0xa, 0xd, '\ __ /____ /____ /) __ /   ',
  0xa, 0xd,  '  ________________  ____

; MBR Signature
  times 510-($-$$) db 0
  db 0x55
  db 0xaa
```