

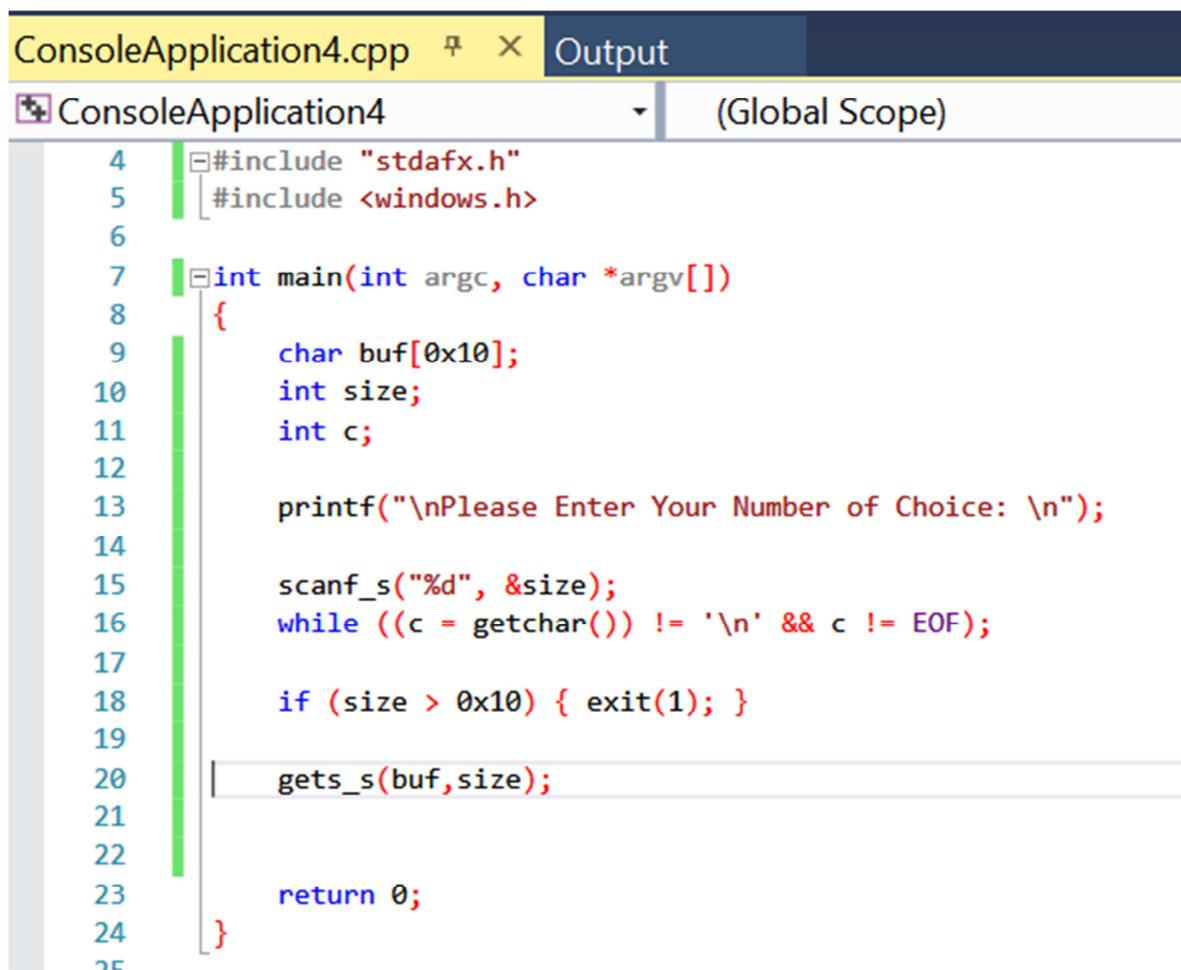
# REVERSING WITH IDA PRO FROM SCRATCH

## PART 21

In part 20, we left an exercise to analyze if it was vulnerable or not and in the list of CracksLatinos, there were many opinions, I did not answer and let them discuss the subject leaving the solution of the exercise and analysis to be answered in this part.

We had the source code that could help us, the **idb** and the executable to be able to reverse in IDA and debug if necessary.

The source code is this.



A screenshot of a code editor window titled "ConsoleApplication4.cpp". The window has tabs for "ConsoleApplication4" and "Output". The code is written in C++ and defines a main function. It includes stdafx.h and windows.h, declares variables buf[0x10], size, and c, and prints a prompt to the console. It then reads input from the user using scanf\_s and gets\_s, checks if the size is greater than 0x10, and returns 0 at the end.

```
4 #include "stdafx.h"
5 #include <windows.h>
6
7 int main(int argc, char *argv[])
8 {
9     char buf[0x10];
10    int size;
11    int c;
12
13    printf("\nPlease Enter Your Number of Choice: \n");
14
15    scanf_s("%d", &size);
16    while ((c = getchar()) != '\n' && c != EOF);
17
18    if (size > 0x10) { exit(1); }
19
20    gets_s(buf, size);
21
22
23    return 0;
24 }
```

We will analyze it statically in IDA first.

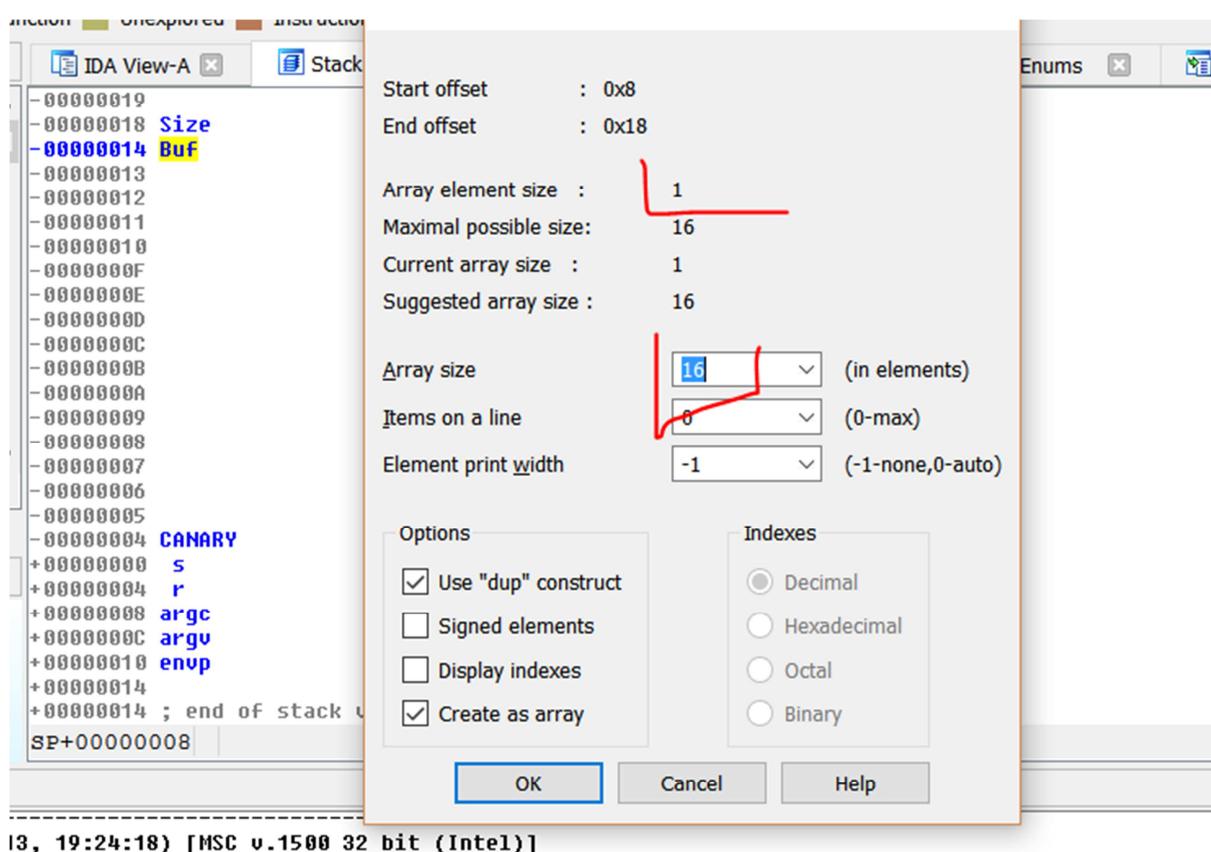
```

00401090 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401090 main proc near
00401090
00401090     Size= dword ptr -18h
00401090     Buf= byte ptr -14h
00401090     CANARY= dword ptr -4
00401090     argc= dword ptr 8
00401090     argv= dword ptr 0Ch
00401090     envp= dword ptr 10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     sub     esp, 18h
00401096     mov     eax, _security_cookie
0040109B     xor     eax, ebp
0040109D     mov     [ebp+CANARY], eax
004010A0     push    esi
004010A1     push    offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \\"...
004010A6     call    printf
004010AB     lea     eax, [ebp+Size]
004010AE     push    eax
004010AF     push    offset aD      ; "%d"
004010B4     call    scanf_s
004010B9     mov     esi, ds:_imp_getchar
004010BF     add     esp, 0Ch

```

There, we have the CANARY that we rename it as always.

We will see the Buf length in the static representation of the stack.



The length is 16 by 1 which is the length of each element or 16 decimal bytes.

```
IDA View-A Stack of Main Hex View-1
-00000019 db ? ; undefined
-00000018 Size dd ?
-00000014 Buf db 16 dup(?)
-00000004 CANARY dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ?
+00000010 envp dd ?
+00000014 ; end of stack variables
```

So if we could write more than 16 bytes in the BUFFER, it would be vulnerable.

DO NOT confuse common bugs or crashes with vulnerabilities, not everything that crashes a program is a vulnerability if I do.

XOR ECX, ECX

DIV ECX

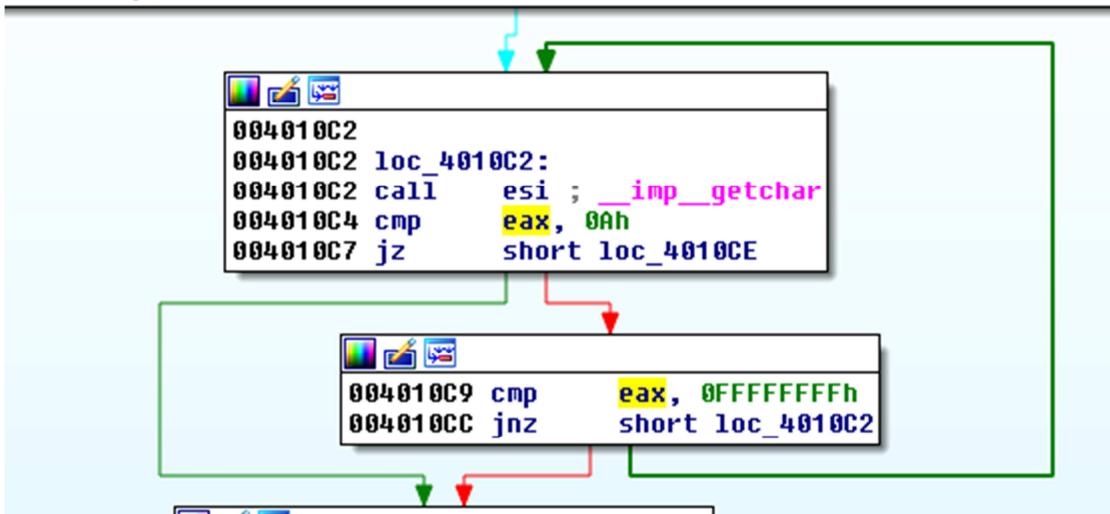
It is a bug that will zero to ECX and divide by zero and will cause an exception that if not handled will be a crash.

That is a simple crash, there are several types of vulnerabilities for now we are only seeing the simpler BUFFER OVERFLOW.

If there is an overflow of a BUFFER being able to write outside of where the space reserved for the same the program will be VULNERABLE because a BUFFER OVERFLOW could occur, later we will see if besides being VULNERABLE it is EXPLOITABLE or not, it can be VULNERABLE and for example NO EXPLOITABLE by some program or system mitigation such as CANARY for example that would prevent its exploitation, but that is a subject for following parts.

So the idea is to analyze if that BUFFER of 16 bytes decimal can be overflowed, as in this case, just below the Buf is the CANARY if we get to overwrite it when copying in the buffer, it is obvious that there is a BUFFER OVERFLOW.

```
v      esi, ds:_imp_getchar  
d      esp, 0Ch
```



That code corresponds to this line of the source code.

```
While ((c = getchar ())!= '\n' && c!= EOF);
```

It is a line that is used after the scan to read the 0xA of the standard input that is the line break, so it does not bother in a subsequent reading of it, because if it remains and does not filter, in the next call to read characters of the keyboard, it will not work.

We see that it loops until it finds the 0xA corresponding to the LINE JUMP or LF.

Non-printable ASCII control characters:

Ascii code 00 = NULL (null character)

Ascii code 01 = SOH (Header start)

Ascii code 02 = STX (Start of text)

Code ascii 03 = ETX (End of text, heart stick English poker cards)

Code ascii 04 = EOT (End of transmission, stick diamonds decks of poker)

Code ascii 05 = ENQ (Consult, stick clubs English poker cards)

Ascii code 06 = ACK (Recognition, stick cards poker cards)

Ascii code 07 = BEL (Ring)

Ascii code 08 = BS (Backspace)

Ascii code 09 = HT (horizontal Tab)

Ascii code 10 = LF (New line - line break)

Ascii code 11 = VT (Vertical Tab)

Ascii code 12 = FF (New page - page break)  
Ascii code 13 = CR (ENTER - carriage return)

I think that happens if I am not mistaken because in WINDOWS when pressing ENTER which is 13 decimal or 0x0d, you cut the input of characters, but always that 0xa in the standard input that annoys in the following input by keyboard, since the Line break also cancels the entry.

Here we see a python script that would be functional to test this exercise.

```
from subprocess import *
import time
p      = Popen([r'VULNERABLE_o_NO.exe','f'],stdout=PIPE,stdin=PIPE,
stderr=STDOUT)

print "ATACHEA EL DEBUGGER Y APRETA ENTER\n"
raw_input()

primera="10\n"
p.stdin.write(primera)

time.sleep(0.5)

segunda="AAAA\n"
p.stdin.write(segunda)

testresult = p.communicate()[0]
time.sleep(0.5)
print(testresult)
print primera
print segunda
```

We see that it is a script that uses subprocess to start the process and then...

```
p      = Popen([r'VULNERABLE_o_NO.exe','f'],stdout=PIPE,stdin=PIPE,
stderr=STDOUT)
```

It redirects the standard input and output so that we can send you characters as if we had typed.

```
primera="10\n"
p.stdin.write(primera)
```

```
time.sleep(0.5)
```

```
segunda="AAAAA\n"
p.stdin.write(segunda)
```

We see that there are two inputs. First, the size that asks me, I enter 10 to try and then the data I enter with gets\_s with the size that we passed before, I can type less than 10.

In addition, I added a raw\_input () so that once the process is started the python script will stop until ENTER is pressed, which allows me to attach the IDA to the process that will start and wait for input by stdin.

Let's test if the script works as we think.

Let's run it.

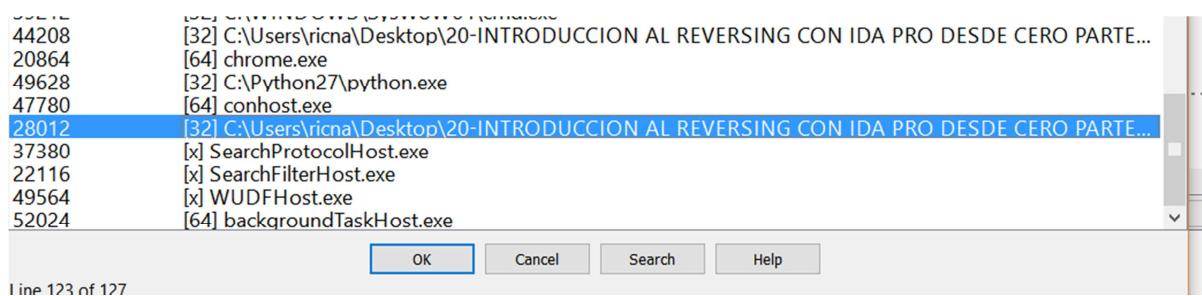
```
pepe.py x script.py x
10     time.sleep(0.5)
11
12     segunda="AAAAA\n"
13     p.stdin.write(segunda)
14
15     testresult = p.communicate()[0]
16     time.sleep(0.5)
17     print(testresult)
18     print primera
19     print segunda
20
```

Run: script pepe

C:\Python27\python.exe C:/Users/ricna/PycharmProjects/untitled/pepe.py  
ATACHEA EL DEBUGGER Y APRETA ENTER

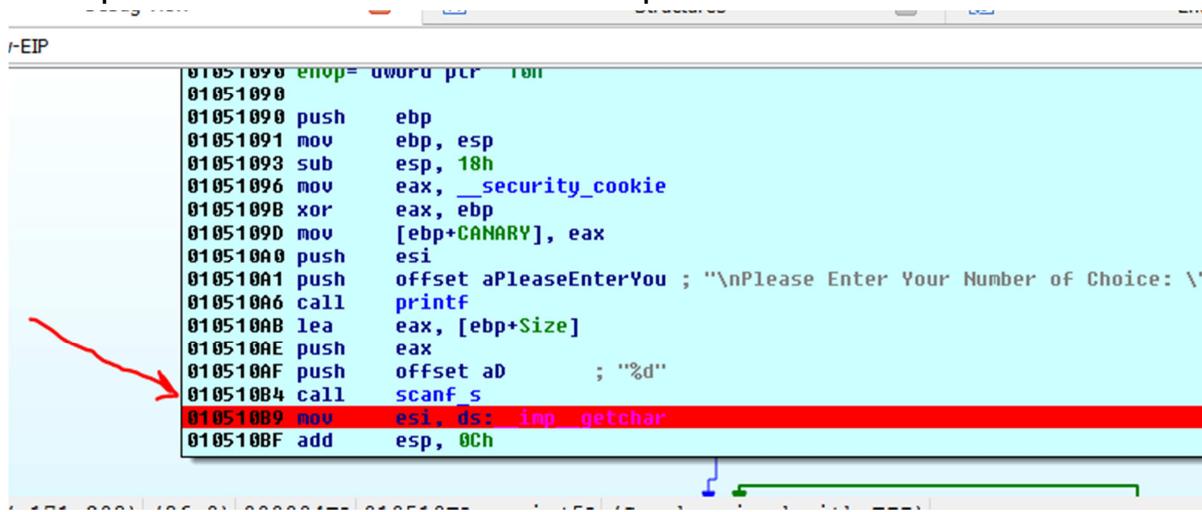
It is stopped there waiting for us to press ENTER giving us the possibility of attaching it in IDA.

In it, I open the VULNERABLE\_o\_NO.exe to analyze it in the LOADER without running it in the debugger and then, I choose LOCAL WINDOWS DEBUGGER and go to DEBUGGER-ATTACH TO PROCESS.



There, you cannot see the name of the executable because the name of the folder is very long, but that's it, I press OK and when it stops, I press F9, so that it is RUNNING.

Then before pressing ENTER in python, I'll set a BREAKPOINT after the first keyboard input since I was waiting there for the process, and I'll only be able to stop after it comes back from that input.



There, I set the BREAKPOINT and then in Python I press ENTER.

We see it stops at the breakpoint.

new-EIP

```

01051096 mov    eax, __security_cookie
01051098 xor    eax, ebp
0105109D mov    [ebp+CANARY], eax
010510A0 push   esi
010510A1 push   offset aPleaseEnterYou ; "\nPlease Enter Your Number of Choice: \\"...
010510A6 call    printf
010510AB lea    eax, [ebp+Size]
010510AE push   eax
010510AF push   offset aD      ; "%d"
010510B4 call    scanf_s
010510B9 mov    esi, ds: _imp_getchar
010510BF add    esp, 0Ch

```

If you pass the mouse over the variable size that is the one that we enter the value in the scanf\_s.

Debug View      Structures

```

01051096 mov    eax, __security_cookie
01051098 xor    eax, ebp
0105109D mov    [ebp+CANARY], eax
010510A0 push   esi
010510A1 push   offset aPleaseEnterYou ; "\nPlease Enter Your Number of Cho
010510A6 call    printf
010510AB lea    eax, [ebp+Size] ←
010510AE push   eax
010510AF push   offset aD      [ebp+Size]=[Stack[00008E90]:0115F7B4]
010510B4 call    scanf_s      db 0Ah
010510B9 mov    esi, ds: _imp ← db 0
010510BF add    esp, 0Ch      db 0
                                db 0
                                db 0
                                dh 0F3h + n

```

There, we see that in the variable Size input the value 10 decimal (0xA) that I entered through the script.

dd esp, 0Ch

```

010510C2 loc_10510C2:
010510C2 call    esi ; _imp_getchar
010510C4 cmp    eax, 0Ah
010510C7 jz     short loc_10510CE

010510C9 cmp    eax, 0FFFFFFFh
010510CC jnz   short loc_10510C2

```

0004C2 010510C2: main:loc\_10510C2 (Synchronized with EIP)

If I get to getchar, the reality is that I did not pass any 0xA characters in the script.

```
primera="10\n"
p.stdin.write(primera)
time.sleep(0.5)

segunda="AAAA\n"
p.stdin.write(segunda)
```

The “ \n” is the ENTER 0x0d

But if I trace the getchar with F8 to skip the CALL and not enter it.



We see that there was a character 0xA that I did not pass, which when read it removed it from the stdin and cleared it for the next input by keyboard.

We see that it compares my size 0xA with the maximum 0x10 and as it is smaller, it continues.

```
010510CE
010510CE loc_10510CE:
010510CE mov     eax, [ebp+Size]
010510D1 pop    esi
010510D2 cmp    eax, 10h
010510D5 jle    short loc_10510DF

1           ; Code
ds:_imp_exit

010510DF
010510DF loc_10510DF:           ; Size
010510DF push   eax
010510E0 lea    eax, [ebp+Buf]
010510E3 push   eax             ; Buf
010510E4 call   ds:  imp_gets_s

00004D5| 010510D5: main+45 (Synchronized with EIP)
```

When I trace the gets\_s with F8.

```
Code
010510DF
010510DF loc_10510DF:           ; Size
010510DF push   eax
010510E0 lea    eax, [ebp+Buf]
010510E3 push   eax             ; Buf
010510E4 call   ds:  imp_gets_s
010510EA mov    ecx, [ebp+CANARY]
010510ED add    esp, 8
010510F0 xor    ecx, ebp
010510F5 ---
```

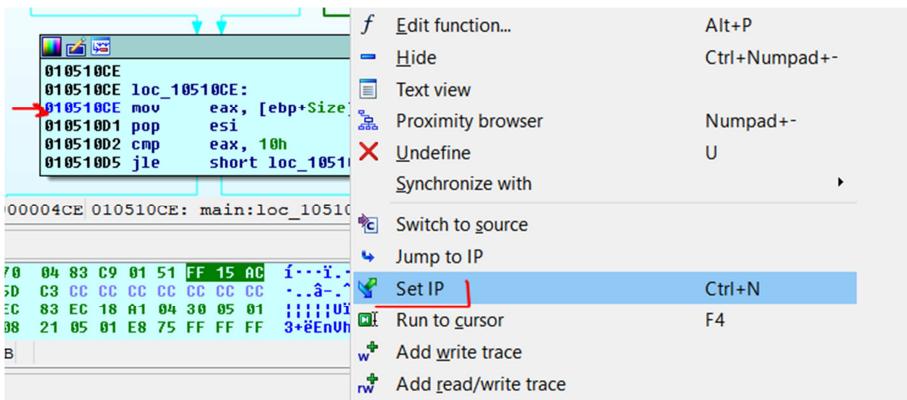
If I pass the mouse over the Buf.

I see that the following A's that I passed in the script entered so that it works and allows me to test and debug what is going on.

If I run the script again and when I get to getchar I skip it by changing the EIP so it does not read the 0xA.

The screenshot shows the assembly view of the main function in Immunity Debugger. The assembly code is as follows:

```
010510C2
010510C2 loc_10510C2:
010510C2 call    esi ; imp_getchar
010510C4 cmp     eax, 0Ah
010510C7 jz      short loc_10510CE
```



We change EIP there so it does not filter the 0xa to see what happens.

When I trace the gets\_s I see that now I do not enter anything.

The screenshot shows assembly code in a debugger. The code is as follows:

```
010510DF loc_10510DF:      ; Size
010510DF push    eax
010510E0 lea     eax, [ebp+Buf]
010510E3 push    eax      ; Buf
010510E4 call   ds:_imp_gets - [ebp+Buf]=[Stack[00006A60]:00CFFAF8]
010510EA mov     ecx, [ebp+CANAL] db 0
010510ED add    esp, 8      db 0A4h ; 
010510F0 xor    ecx, ebp    db 23h ; #
010510F2 xor    eax, eax    db 75h ; u
010510F4 call   __security_check_icall_nop db 0
010510F9 mov     esp, ebp    db 0D0h ; -
010510FB pop    ebp       db 0BFh ; +
010510FC ret
```

The registers column on the right shows:

EAX	EBX	ECX	EDX	ESI	EDI	EBP	ESP	EIP	FL
-----	-----	-----	-----	-----	-----	-----	-----	-----	----

That means that the 0xA character that is left in the stdin must be filtered after certain entries so that it does not affect if there are subsequent entries, which is why the line of code is justified.

While ((c = getchar ()) != '\n' && c != EOF);

Also now that we have the script we can analyze the crash that occurs in gets\_s when you type the maximum size to see if something else happens or it is just a crash.

```
primera="16\n"
p.stdin.write(primer)
time.sleep(0.5)

segunda="A" *16 + "\n"
p.stdin.write(segunda)
```

I'll see what happens in this case.

With that information, I get there.

```
010510DF
010510DF loc_10510DF:          ; Size
010510DF push    eax
010510E0 lea     eax, [ebp+Buf] ; Buf
010510E3 push    eax           ; Buf
010510E4 call    ds: imp gets_s
010510EA mov     ecx, [ebp+CANARY]
010510ED add    esp, 8
010510F0 xor    ecx, ebp
010510F2 xor    eax, eax
010510F4 call    security_check_cookie
```

When I pass the LEA, I write the address where the buffer starts, in my case, 0x0115FDD8.

If I double-click on CANARY and press D several times until it is a dword (dd), I can also note the address, in my case, 0x115FDE8 and the CANARY value, in this case, will be 965FA1F4.

```
Stack[0000C658]:0115FDE4 db 0FBh ; v
Stack[0000C658]:0115FDE5 db 0A4h ; ~
Stack[0000C658]:0115FDE6 db 23h ; #
Stack[0000C658]:0115FDE7 db 75h ; u
Stack[0000C658]:0115FDE8 dd 965FA1F4h
Stack[0000C658]:0115FDEC db 34h ; 4
Stack[0000C658]:0115FDED db 0FEh ; !
Stack[0000C658]:0115FDEE db 15h
Stack[0000C658]:0115FDEF db 1
Stack[0000C658]:0115FDF0 db 0C5h ; +
```

I press F9.



There, it is the exception that the API produces, I accept OK.

```
ucrtbase.dll:75287FAF db 6Ah ; j
ucrtbase.dll:75287FB0 db 5
ucrtbase.dll:75287FB1 db 59h ; v
ucrtbase.dll:75287FB2 ; -----
ucrtbase.dll:75287FB2 int 29h ; Win8: RtlFailFast(ecx)
ucrtbase.dll:75287FB2 ;
ucrtbase.dll:75287FB4 db 51h ; Q
ucrtbase.dll:75287FB5 db 0BEh ; +
ucrtbase.dll:75287FB6 db 17h
ucrtbase.dll:75287FB7 dh 4
```

There it is. Let's go with G and put the address of the buffer and then the canary to see what happened.

We see that the canary is intact.

DA View-EIP

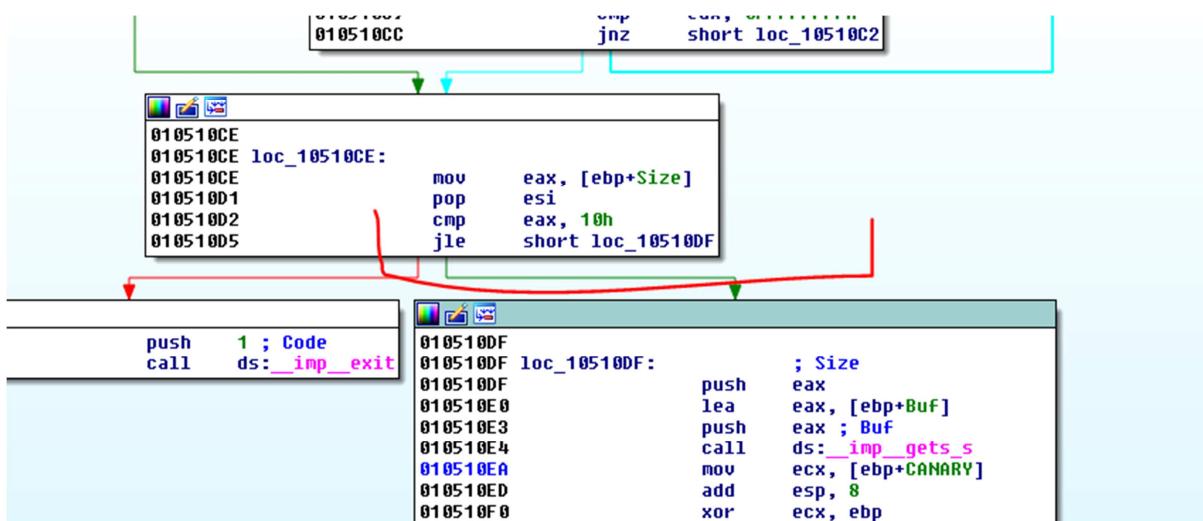
```
Stack[0000C658]:0115FDDE db 41h ; A
Stack[0000C658]:0115FDDF db 41h ; A
Stack[0000C658]:0115FDE0 db 41h ; A
Stack[0000C658]:0115FDE1 db 41h ; A
Stack[0000C658]:0115FDE2 db 41h ; A
Stack[0000C658]:0115FDE3 db 41h ; A
Stack[0000C658]:0115FDE4 db 41h ; A
Stack[0000C658]:0115FDE5 db 41h ; A
Stack[0000C658]:0115FDE6 db 41h ; A
Stack[0000C658]:0115FDE7 db 41h ; A
Stack[0000C658]:0115FDE8 dd 965FA1F4h
Stack[0000C658]:0115FDEC db 34h ; 4
```

And the buffer was filled. So, typing the maximum does not produce overflow, what is true is that the buffer was full and there was no final zero of the string, which could cause problems if the program continues, but in this case the exception is not managed and the program is closed. So, here it's just a crash. (I also think it puts a zero at the beginning of the buffer to override the string)

If the program handles the exception and continues, it should discard the data from the buffer because if it took it and used it as a string, having no end zero, it could append data after the stack and cause problems, but when it put the zero at the beginning also cancels it.

Well, knowing that there is no overflow here, let's go back to static analysis.

Let's focus on this part.



We had said that the JL or JLE jump considers the sign or that EAX could be negative, if for example it was 0xFFFFFFFF would be -1 and would be less than 0x10.

It means that if it passed as size -1, it would be possible for the comparison to pass. Let's see.

```
primera="-1\n"
```

```
p.stdin.write(primera)
```

```
time.sleep(0.5)
```

```
segunda="A" *0x2000 + "\n"
```

```
p.stdin.write(segunda)
```

Let's try the script with these values (size -1)

When you stop at the breakpoint.

```
0105109B xor    eax, ebp
0105109D mov    [ebp+CANARY], eax
010510A0 push   esi
010510A1 push   offset aPleaseEnterYou ; "\nPleasE Ent
010510A6 call   printf
010510AB lea    eax, [ebp+Size]
010510AE push   eax
010510AF push   offset aD      ; "%d"
010510B4 call   scanf_s
010510B9 mov    esi, ds:_imp_getchar
010510BF add    esp, 0Ch
```

Let's see the value of the size.

```
01051096 mov    eax, __security_cookie
0105109B xor    eax, ebp
0105109D mov    [ebp+CANARY], eax
010510A0 push   esi
010510A1 push   offset aPleaseEnterYou ; "\nPleasE Enter Your Number of Choice
010510A6 call   printf
010510AB lea    eax, [ebp+Size] ←
010510AE push   eax
010510AF push   offset aD      [ebp+Size]=[Stack[ 0000B290]:00CFF910]
010510B4 call   scanf_s
010510B9 mov    esi, ds:_imp
010510BF add    esp, 0Ch
db 0FFh
db 0FFh
db 0FFh
db 0E3h ; p
db 0A4h ; n
db 23h : #
```

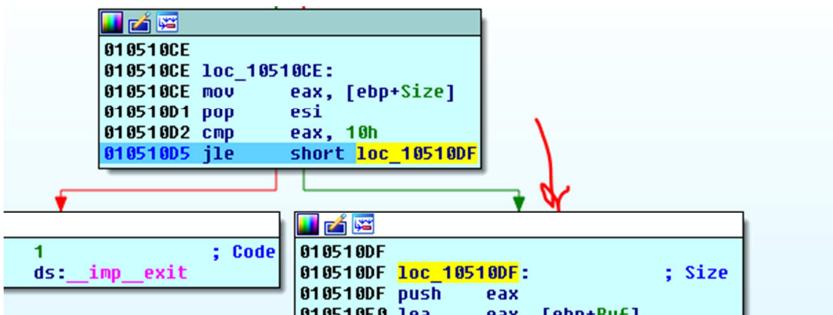
We see that it is worth 0xFFFFFFFF if I double click on size and press D to group it until it is a dword.

```

Stack[0000B290]:00CFF90E db 2Dh ; -
Stack[0000B290]:00CFF90F db 75h ; u
Stack[0000B290]:00CFF910 dd 0xFFFFFFFFh
Stack[0000B290]:00CFF914 db 0E3h ; p
Stack[0000B290]:00CFF915 db 0A4h ; n
Stack[0000B290]:00CFF916 db 23h ; #
Stack[0000B290]:00CFF917 db 75h ; u
Stack[0000B290]:00CFF918 db 0E3h ; p

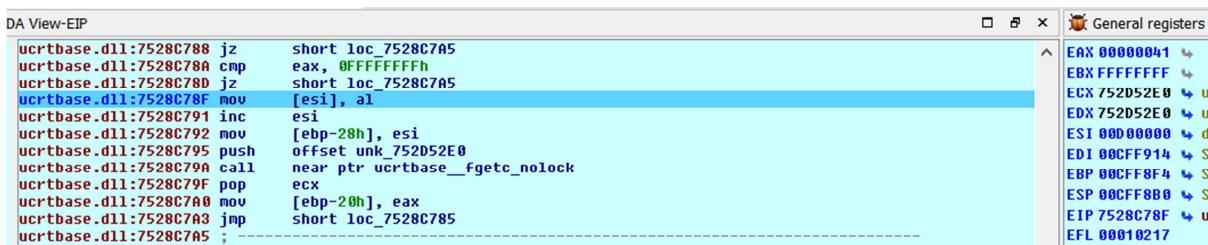
```

Let's continue tracing until the comparison.



We see that as 0xFFFFFFFF is -1, when considering the sign it does not filter and will continue to read that size. The reality is that in gets\_s as well as a memcpy and any API that copies or enters bytes the sizes are interpreted as unsigned, because there are no negative sizes, as they cannot be entered or copied -1 bytes, that is impossible, it Interprets it as positive 0xFFFFFFFF which we see will produce an overflow.

I go back to see where the buffer is this time at 0x00CFF914.



If I go where the buffer starts.

```
Stack[0000B290]:00CFF90D db 0FFh
Stack[0000B290]:00CFF90E db 0FFh
Stack[0000B290]:00CFF90F db 0FFh
Stack[0000B290]:00CFF910 dd 0FFFFFFFh
Stack[0000B290]:00CFF914 db 41h ; A
Stack[0000B290]:00CFF915 db 41h ; A
Stack[0000B290]:00CFF916 db 41h ; A
Stack[0000B290]:00CFF917 db 41h ; A
Stack[0000B290]:00CFF918 db 41h ; A
Stack[0000B290]:00CFF919 db 41h ; A
Stack[0000B290]:00CFF91A db 41h ; A
Stack[0000B290]:00CFF91B db 41h ; A
Stack[0000B290]:00CFF91C db 41h ; A
Stack[0000B290]:00CFF91D db 41h ; A
Stack[0000B290]:00CFF91E db 41h ; A
Stack[0000B290]:00CFF91F db 41h ; A
Stack[0000B290]:00CFF920 db 41h ; A
Stack[0000B290]:00CFF921 db 41h ; A
```

UNKNOWN 00CFF914: Stack[0000B290]:00CFF914 (Synchronized)

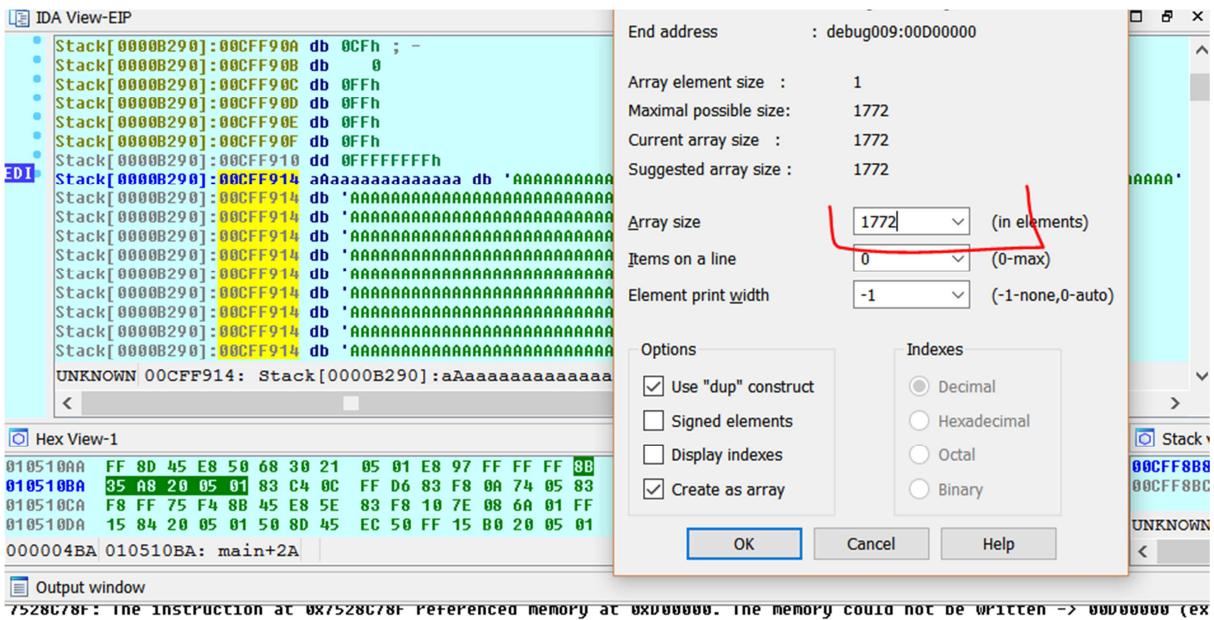
### lex View-1

**10AA** FF 8D 45 E8 50 68 30 21 05 01 E8 97 FF FF FF 8B .EFPh0?  
**10BA** 35 A8 29 95 01 83 C4 9C FF D6 83 F8 0A 74 05 83 5;...â...

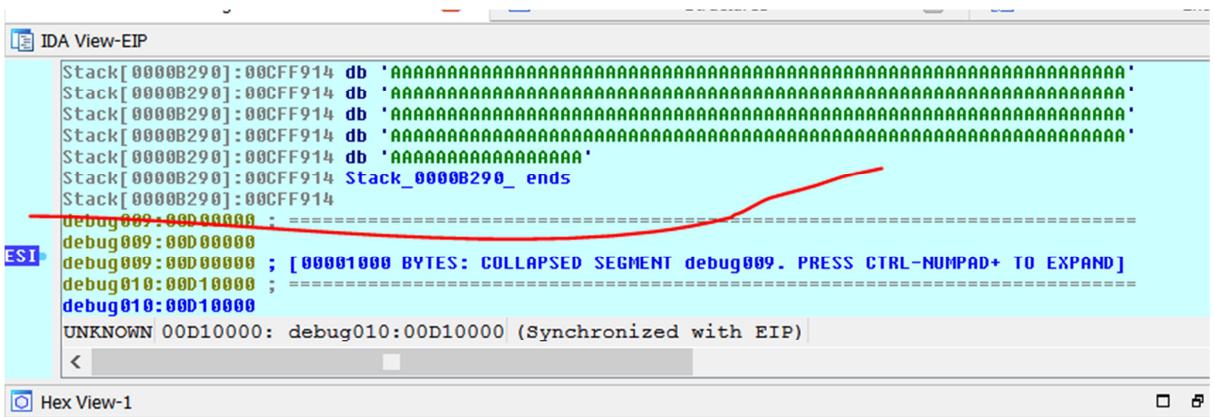
We can see better how much it copied if I transform it in string, at the beginning of the string, press A.

I see the thing is great. ☺

And if I transform it into ARRAY.



I also see that it reaches the end of the stack, overwriting the CANARY and all.



He filled it with Aes until the end of it. Just below, I see that it already changes to another section to debug09.

With this we verify that it is vulnerable. Now, how could it be fixed? Obviously, if instead of using a JL or JLE jump that considers the sign we use JB or JBE that does not consider it if we pass -1 will be 0xFFFFFFFF but in the comparison it will take it as positive and it will be bigger than 0x10 and it will go out.

In the source code it would be so.

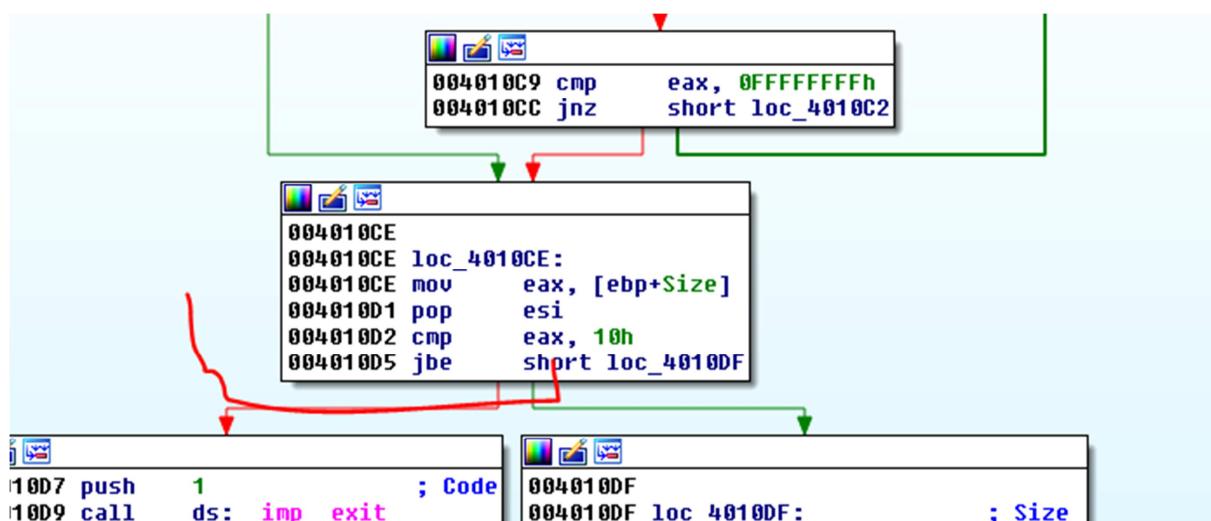
soleApplication4 (Global Scope)

```

4  #include "stdafx.h"
5  #include <windows.h>
6
7  int main(int argc, char *argv[])
8  {
9      char buf[0x10];
10     unsigned int size;
11     int c;
12
13     printf("\nPlease Enter Your Number of Choice: \n");
14
15     scanf_s("%d", &size);
16     while ((c = getchar()) != '\n' && c != EOF);
17
18     if (size > 0x10) { exit(1); }
19
20     gets_s(buf, size);
21
22
23     return 0;
24 }
```

A single word transforms it from VULNERABLE to NOT VULNERABLE. We compiled it.

NO\_VULNERABLE.exe is the name of the repaired program.



We see that just changing the type of variable to UNSIGNED, it changed when compiling the type of jump that does not consider the sign.

I fix the script by renaming it to load this new executable. I leave the rest the same.

I analyze the new executable in IDA in the LOADER and then started the script and before pressing ENTER, I attach the LOCAL DEBUGGER and I set a BREAKPOINT in the comparison of the size so that it stops there to see what happens.

```
00AE10CE
00AE10CE loc_AE10CE:
00AE10CE mov     eax, [ebp+Size]
00AE10D1 pop    esi
00AE10D2 cmp     eax, 10h
00AE10D5 jbe    short loc_AE10DF
```

If I see, in size, the value is still 0xFFFFFFFF.

```
00AE10CE
00AE10CE loc_AE10CE:
00AE10CE mov     eax, [ebp+Size]
00AE10D1 pop    esi
00AE10D2 cmp     eax, 10h      [ebp+Size]=[Stack[...]]
00AE10D5 jbe    short loc_AE10DF
```

But If I keep tracing.

```
00AE10CE
00AE10CE loc_AE10CE:
00AE10CE mov     eax, [ebp+Size]
00AE10D1 pop    esi
00AE10D2 cmp     eax, 10h
00AE10D5 jbe    short loc_AE10DF
```

```
00AE10D7 push 1 ; Code
00AE10D9 call ds:_imp_exit
```

,731 | (204,32) 000004D5 00AE10D5: main+45 (Synchronized with EIP)

Now, it goes to EXIT and it avoids overflow since JBE considers that 0xFFFFFFFF as it does not care the sign, like a big positive number and greater than 0x10, with which the program is repaired.

Therefore the answer to the exercise is that it was VULNERABLE and that it is repaired by changing the size of INT that is SIGNED to UNSIGNED INTEGER, thereby changing the JLE jump by JBE of a jump that considers the sign to one that does not.

Ricardo Narvaja

Translated by: @IvinsonCLS