

REVERSING WITH IDA PRO FROM SCRATCH

PART 20

Vulnerabilities.

In this part, we'll see a topic about vulnerabilities and how to analyze some of the easiest ones.

What is vulnerability?

In computer security, the word vulnerability refers to a system weakness allowing an attacker to violate confidentiality, integrity, availability, access control and system consistency or data and applications.

Vulnerabilities are the result of bugs or errors in a system design. Although, in a broader sense, they can also be the result of the same technological limitations because, firstly, there is not a 100% safe system. So, there are theoretical and real vulnerabilities known as exploits.

Definitions:

Bug: https://en.wikipedia.org/wiki/Software_bug

Exploit: https://en.wikipedia.org/wiki/Exploit_%28computer_security%29

A vulnerable program has bugs or programming errors and according to the kind of bugs, they can be exploited executing malicious code in it, but the authentication can also fail and allow the user do illegal actions, trigger crashes, allow privilege escalation, etc.

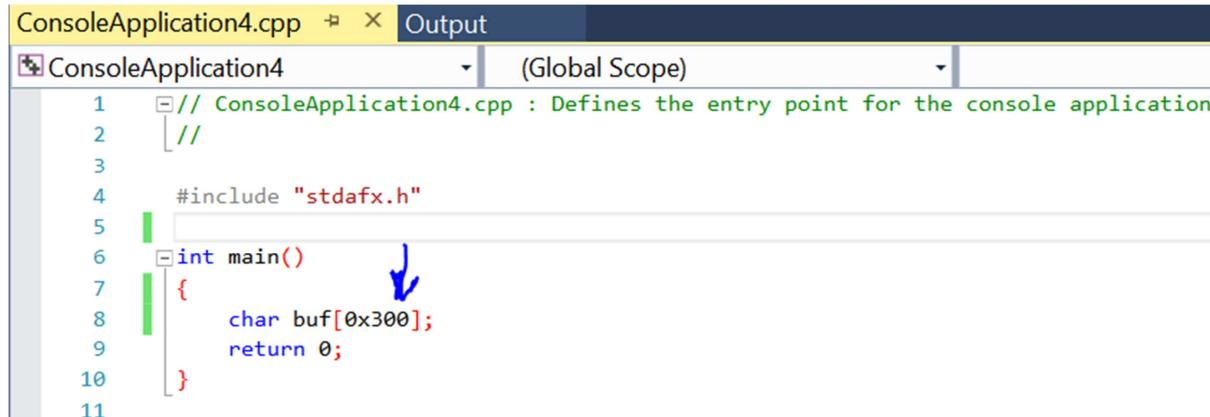
Of course, through memory corruption bugs. The easiest ones are the buffers overflows which are produced when a program reserves a memory zone or buffer to store data and, for some reason, the destination data size is not checked in an adequate way and the buffer overflows copying more than the reserved size being able to overwrite variables, arguments and pointers in the memory.

The easiest buffer overflow is the stack overflow that is when a buffer reserved in the stack overflows.

A buffer in a simple C program source code can be:

```
char buf[xxx];
```

xxx is the buffer size, in this case, it is a 0x300 bytes stack buffer.



```
1 // ConsoleApplication4.cpp : Defines the entry point for the console application
2 //
3
4 #include "stdafx.h"
5
6 int main()
7 {
8     char buf[0x300];
9     return 0;
10}
11
```

Obviously, the program does nothing, but we compile it and see it in IDA. It is attached in this tutorial as Compilado_1.exe.

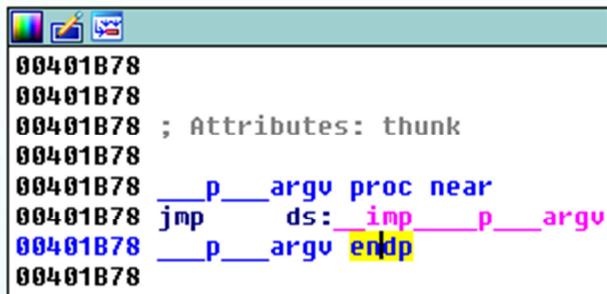
Name	Address
get startup argv mode	00401672
configure narrow argv	00401B42
p argc	00401B72
p argv	00401B78
imp p argv	0040207C
imp p argc	00402080
imp configure narrow argv	0040209C

Searching the **argc** or **argv** references we get the main. Double click to get there.



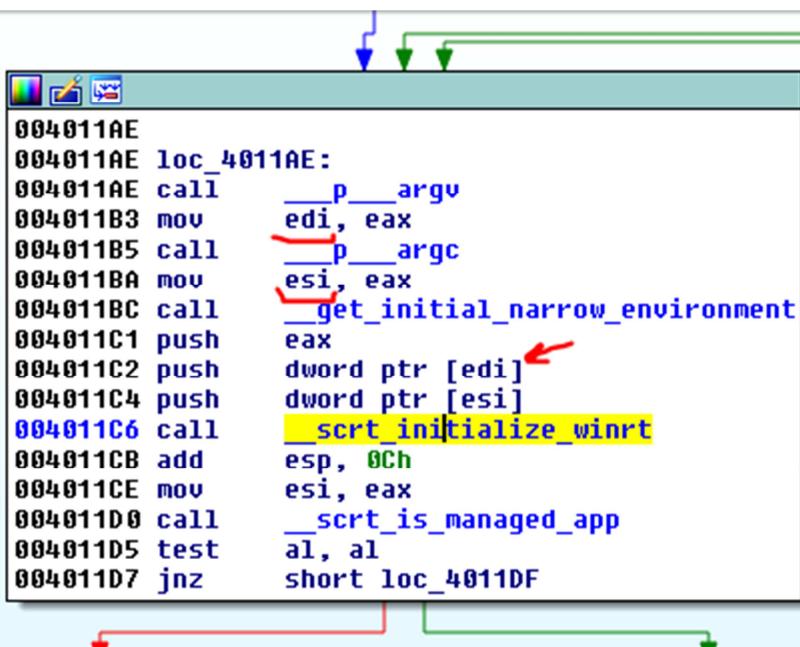
```
.idata:00402074      extrn __imp__c_exit:dword ; DATA XREF: __c_exit@r
idata:00402078      extrn __imp__register_thread_local_exe_atexit_callback:dword
idata:00402078          ; DATA XREF: __register_thread_local_exe_a
idata:0040207C      extrn __imp__p_argv:dword ; DATA XREF: __p_argv@r
idata:00402080      extrn __imp__p_argc:dword ; DATA XREF: __p_argc@r
idata:00402084 ; void __cdecl __noreturn __exit(int Code)
idata:00402084      extrn __imp__exit:dword ; DATA XREF: __exit@r
idata:00402088 ; void __cdecl __noreturn __exit(int Code)
idata:00402088      extrn __imp__exit:dword ; DATA XREF: __exit@r
idata:0040208C      extrn __imp__initterm_e:dword ; DATA XREF: __initterm_e@r
```

And press X.



```
00401B78
00401B78
00401B78 ; Attributes: thunk
00401B78
00401B78 __p__argv proc near
00401B78 jmp ds:_imp__p__argv
00401B78 __p__argv endp
00401B78
```

Searching references we get to the known block that calls the main, in this case, it renamed it. Maybe, because it does nothing.



```
004011AE
004011AE loc_4011AE:
004011AE call    __p__argv
004011B3 mov     edi, eax
004011B5 call    __p__argc
004011BA mov     esi, eax
004011BC call    __get_initial_narrow_environment
004011C1 push   eax
004011C2 push   dword ptr [edi]
004011C4 push   dword ptr [esi]
004011C6 call    __scrt_initialize_winrt
004011CB add    esp, 0Ch
004011CE mov     esi, eax
004011D0 call    __scrt_is_managed_app
004011D5 test   al, al
004011D7 jnz    short loc_4011DF
```

If we enter the function, it reserves nothing because it doesn't use the buffer and it returns 0 in EAX.

```
00401000 .model flat
00401000
00401000
00401000 ; Segment type: Pure code
00401000 ; Segment permissions: Read/Execute
00401000 _text segment para public 'CODE' use32
00401000 assume cs:_text
00401000 ;org 401000h
00401000 assume es:nothing, ss:nothing, ds:_data, fs:nothing, g:
00401000
00401000
00401000
00401000 __scrt_initialize_winrt proc near
00401000 xor    eax, eax
00401002 retn
00401002 __scrt_initialize_winrt endp
00401002
```

We have to use the buffer for it to finish reserving space.

```
ConsoleApplication4.cpp  Output  
ConsoleApplication4  (Global)  
4     #include "stdafx.h"  
5  
6     int main()  
7     {  
8         char buf[0x300];  
9  
10        gets_s(buf, 0x300);  
11  
12        return 0;  
13    }  
14  
15
```

Now, we use the `gets_s` function for the user to type something in the console and it saves it in the buffer.

... < Referencia de la biblioteca en tiempo de ejecución > Referencia ampliada de función

gets_s, _getws_s

Visual Studio 2015 | Otras versiones ▾

Para obtener la documentación más reciente de Visual Studio 2017 RC, consulte [dc](#) [Visual Studio 2017 RC](#).

Obtiene una línea del flujo `stdin`. Estas versiones de `gets`, `_getws` tienen mejoras que se describen en [Características de seguridad de CRT](#).

Sintaxis

```
char *gets_s(  
    char *buffer,  
    size_t sizeInCharacters  
)
```

That function has two arguments: the buffer and the character max length to avoid overflows. In the example, there is no overflow because the copied size is not greater than the created buffer of 0x300 bytes.

```
ConsoleApplication4.cpp  ✎ × Output
ConsoleApplication4 (Global Sca
4     #include "stdafx.h"
5
6     int main()
7     {
8         char buf[0x300];
9         gets_s(buf, 0x300);
10
11
12
13     return 0;
14 }
15
```

There is no possibility of overflow while what we copy is less or equal to 0x300 bytes.

Let's see it in IDA. This is attached as Compilado_2.exe.

IDA View-A Hex View-1 Structures Enums Imports

```
00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub    esp, 304h
00401009 mov     eax, _security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea     eax, [ebp+Buf]
00401019 push    300h          ; Size
0040101E push    eax, [Buf]
0040101F call    ds:_imp_gets_s
00401025 mov     ecx, [ebp+var_4]
00401028 add     esp, 8
0040102B xor     ecx, ebp
0040102D xor     eax, eax
0040102F call    _security_check_cookie
00401034 mov     esp, ebp
00401036 pop    ebp
00401037 retn
00401037 main endp
```

.00.00% (-109,565) (29,66) 00000400 00401000: main (Synchronized with Hex View-1)

I compiled it with symbols and IDA also found them in my PC.
It looks better. We see the main with its args and variables.

Let's analyze it with IDA, reversing it.

```
-00000018 db ? ; undefined
-00000017 db ? ; undefined
-00000016 db ? ; undefined
-00000015 db ? ; undefined
-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 db ? ; undefined
-0000000F db ? ; undefined
-0000000E db ? ; undefined
-0000000D db ? ; undefined
-0000000C db ? ; undefined
-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 dd ?
+00000000 S db 4 dup(?)
+00000004 R db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ?           ; offset
+00000010 envp dd ?           ; offset
+00000014 ; end of stack variables
```

SP+000002F8

If we double click on any variable or arg, we go to the stack static view.

We see the 3 args: **envp**, **argv** and **argc** that we don't use in the main.

They are pushed into the stack before calling the main.

```
004011E3
004011E3 loc_4011E3:
004011E3 call    __p__argv
004011E8 mov     edi, eax
004011EA call    __p__argc
004011EF mov     esi, eax
004011F1 call    __get_initial_narrow_environment
004011F6 push   eax ; envp
004011F7 push   dword ptr [edi] ; argv
004011F9 push   dword ptr [esi] ; argc
004011FB call    main
00401200 add    esp, 0Ch
00401203 mov     esi, eax
00401205 call    __scrt_is_managed_app
0040120A test   al, al
0040120C jnz    short loc_401214
```

It saves the RETURN ADDRESS that is the address it saves to know where to return after leaving the CALL. In this case, the return address will be 0x401200 if there is no randomization.

```
004011E3
004011E3 loc_4011E3:
004011E3 call    __p__argv
004011E8 mov     edi, eax
004011EA call    __p__argc
004011EF mov     esi, eax
004011F1 call    __get_initial_narrow_environment
004011F6 push   eax ; envp
004011F7 push   dword ptr [edi] ; argv
004011F9 push   dword ptr [esi] ; argc
004011FB call    main
00401200 add    esp, 0Ch
00401203 mov     esi, eax
00401205 call    __scrt_is_managed_app
0040120C jnz    short loc_401214
```



```
0040120E push   esi ; Code
0040120F call    _exit_0
```



```
00401214 loc_401214:
00401214 test   bl, bl
00401216 jnz    short loc_40121D
```

It will return there after executing the main. So, it must save that 0x401200 value in the stack just above the 3 args.

```

--vvvvvvvH          dd ? ; undefined
-00000009          db ? ; undefined
-00000008          db ? ; undefined
-00000007          db ? ; undefined
-00000006          db ? ; undefined
-00000005          db ? ; undefined
-00000004 var_4    db 4 dup(?)
+00000000 s        db 4 dup(?)
+00000004 r        db 4 dup(?)
+00000008 argc    dd ?
+0000000C argv    dd ? ; offset
+00000010 envp    dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Then, it starts executing the main. The first thing is the PUSH EBP.

```

00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push ebp
00401001 mov ebp, esp
00401003 sub esp, 30h
00401009 mov eax, __security_cookie
0040100E xor eax, ebp
00401010 mov [ebp+var_4], eax
00401013 lea eax, [ebp+Buf]
00401019 push 300h ; Size
0040101E push eax ; Buf
0040101F call ds:_imp__gets_s
00401025 mov ecx, [ebp+var_4]
00401028 add esp, 8
0040102B xor ecx, ebp
0040102D xor eax, eax
0040102F call __security_check_cookie
00401034 mov esp, ebp

```

That saves, in the stack, the EBP value used by the function that called the main. Just above the 0x401200 return address, we don't know what value it will have because it changes with the execution, but it is the stored EBP of the parent function of this.

IDA View-A Stack of main Hex View-1 Structures

```

-00000008          db ? ; undefined
-00000007          db ? ; undefined
-00000006          db ? ; undefined
-00000005          db ? ; undefined
-00000004 var_4    dd ?
+00000000 s        db 4 dup(?)
+00000004 r        db 4 dup(?)
+00000008 argc    dd ?
+0000000C argv    dd ? ; offset
+00000010 envp    dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

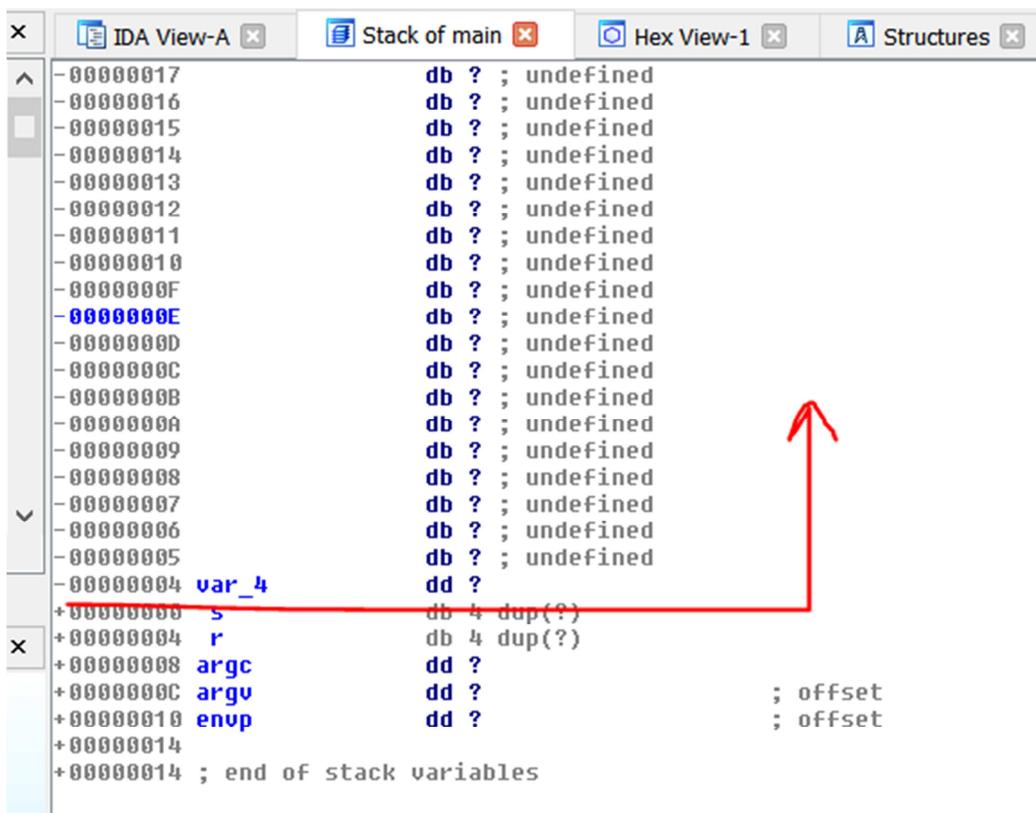
It will be saved in the stack above the return address.

The next thing executed is:

```
00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub    esp, 304h
00401009 mov     eax, _security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea     eax, [ebp+Buf]
```

It establishes EBP as the BASE in this function equaling it with ESP. This is a MOV. It just changes the EBP value not the stack.

Then, sub esp, 0x304 moves ESP upwards reserving space for local variables and stack buffers located above the stored EBP and ESP will be working in an EBP-based function. Just above the reserved space.



IDA View-A Stack of main Hex View-1 Structures

Address	Type	Value
-00000017	db ?	undefined
-00000016	db ?	undefined
-00000015	db ?	undefined
-00000014	db ?	undefined
-00000013	db ?	undefined
-00000012	db ?	undefined
-00000011	db ?	undefined
-00000010	db ?	undefined
-0000000F	db ?	undefined
-0000000E	db ?	undefined
-0000000D	db ?	undefined
-0000000C	db ?	undefined
-0000000B	db ?	undefined
-0000000A	db ?	undefined
-00000009	db ?	undefined
-00000008	db ?	undefined
-00000007	db ?	undefined
-00000006	db ?	undefined
-00000005	db ?	undefined
-00000004 var_4	dd ?	
+00000000 s	db 4 dup(?)	
+00000004 r	db 4 dup(?)	
+00000008 argc	dd ?	
+0000000C argv	dd ?	; offset
+00000010 envp	dd ?	; offset
+00000014		
+00000014 ; end of stack variables		

There, we see the reserved space for variables and buffers just above the s (STORED EBP).

The first variable almost always found is the stack protection CANARY. In this case, it is called var_4.

```
00401000 ; int __cdecl main(int argc, const char **argv
00401000     main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 var_4= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub    esp, 304h
00401009 mov     eax, _security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+var_4], eax
00401013 lea     eax, [ebp+Buf]
00401019 push    300h           ; Size
0040101E push    eax           ; Buf
0040101F call    ds:_imp_gets_s
00401025 mov     ecx, [ebp+var_4]
00401028 add    esp, 8
0040102B xor     ecx, ebp
0040102D xor     eax, eax
0040102F call    _security_check_cookie
00401031 mull   eax, ebx
```

There, we see that it reads the _security cookie value that is a random value that changes in each execution. It is XORed with EBP and saved in var_4. Let's rename it as CANARY.

```
00401000 main proc near
00401000
00401000 Buf= byte ptr -304h
00401000 CANARY= dword ptr -4
00401000 argc= dword ptr 8
00401000 argv= dword ptr 0Ch
00401000 envp= dword ptr 10h
00401000
00401000 push    ebp
00401001 mov     ebp, esp
00401003 sub    esp, 304h
00401009 mov     eax, _security_cookie
0040100E xor     eax, ebp
00401010 mov     [ebp+CANARY], eax
00401013 lea     eax, [ebp+Buf]
00401019 push    300h           ; Size
0040101E push    eax           ; Buf
0040101F call    ds:_imp_gets_s
00401025 mov     ecx, [ebp+CANARY]
00401028 add    esp, 8
0040102B xor     ecx, ebp
0040102D xor     eax, eax
```

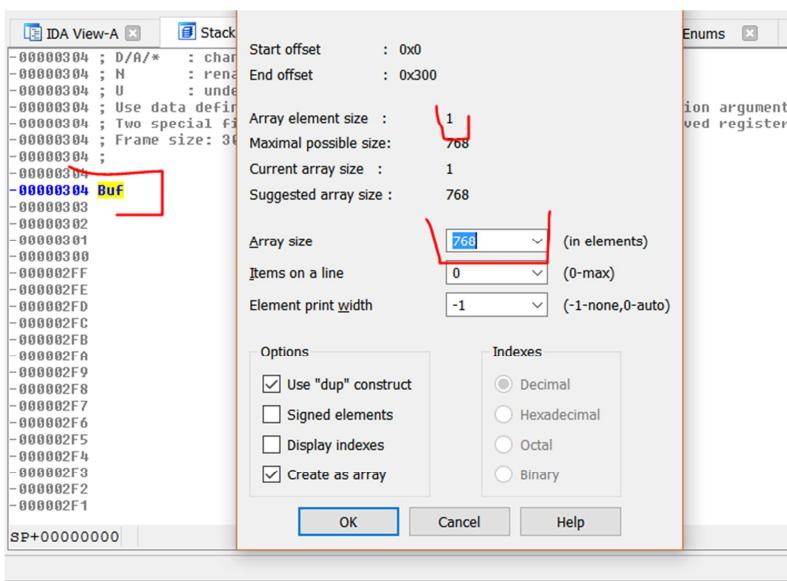
Above the CANARY is Buf. Let's see it in the stack static view.

```

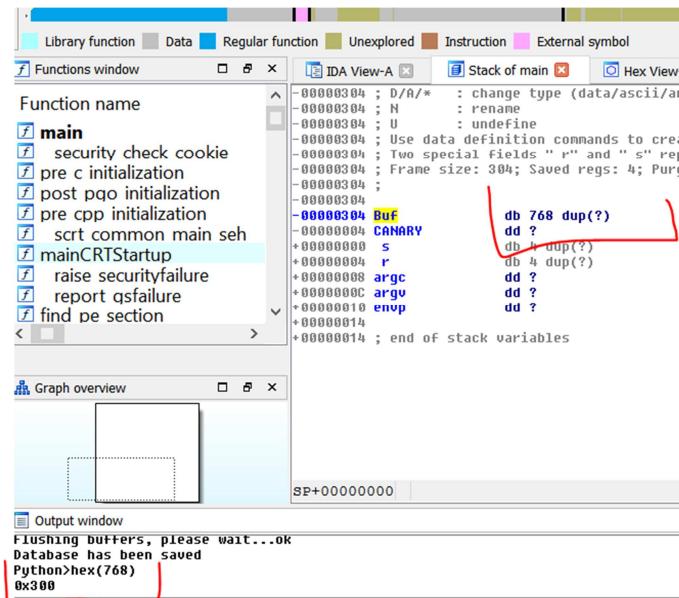
-00000304 ; Two special fields " r" and " s" represent return address an
-00000304 ; Frame size: 304; Saved regs: 4; Purge: 0
-00000304 ;
-00000304
-00000304 Buf
-00000303 db ?
-00000302 db ? ; undefined
-00000301 db ? ; undefined
-00000300 db ? ; undefined
-000002FF db ? ; undefined
-000002FE db ? ; undefined
-000002FD db ? ; undefined
-000002FC db ? ; undefined
-000002FB db ? ; undefined
-000002FA db ? ; undefined
-000002F9 db ? ; undefined
-000002F8 db ? ; undefined
-000002F7 db ? ; undefined
-000002F6 db ? ; undefined
-000002F5 db ? ; undefined
-000002F4 db ? ; undefined
-000002F3 db ? ; undefined
-000002F2 dh ? - undefined

```

When I see an empty space in the static view, there is a buffer possibly. Right click on Buf and select ARRAY.



Its size is 768 decimal by 1 that is the size of each element. So, 768 in hex is 0x300.



We accept it and Buf is defined as 0x300 bytes.

There, we see the gets_s CALL, 0x300 max size and the other arg is the buffer address got through the LEA.

```

0040100E xor    eax, ebp      --
00401010 mov    [ebp+CANARY], eax
00401013 lea    eax, [ebp+Buf]
00401019 push   300h          ; Size
0040101E push   eax           ; Buf
0040101F call   ds:imp_gets_s
00401025 mov    ecx, [ebp+CANARY]

```

We verified that the buf size is 0x300 and it copies 0x300 max through the gets_s.

Obviously, if we could have overflowed the buffer copying more than 0x300, we had overwritten the CANARY, STORED EBP and the RETURN ADDRESS that are below the BUFFER.

```

-00000304 ; D/A/* : change type (data/ascii/array)
-00000304 ; N : rename
-00000304 ; U : undefine
-00000304 ; Use data definition commands to create local variables a
-00000304 ; Two special fields " r" and " s" represent return address
-00000304 ; Frame size: 304; Saved regs: 4; Purge: 0
-00000304 ;
-00000304
-00000304 Buf db 768 dup(?)
-00000304 CANARY dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

But this is not the case. This is a good example of a buffer written in a correct way.

```

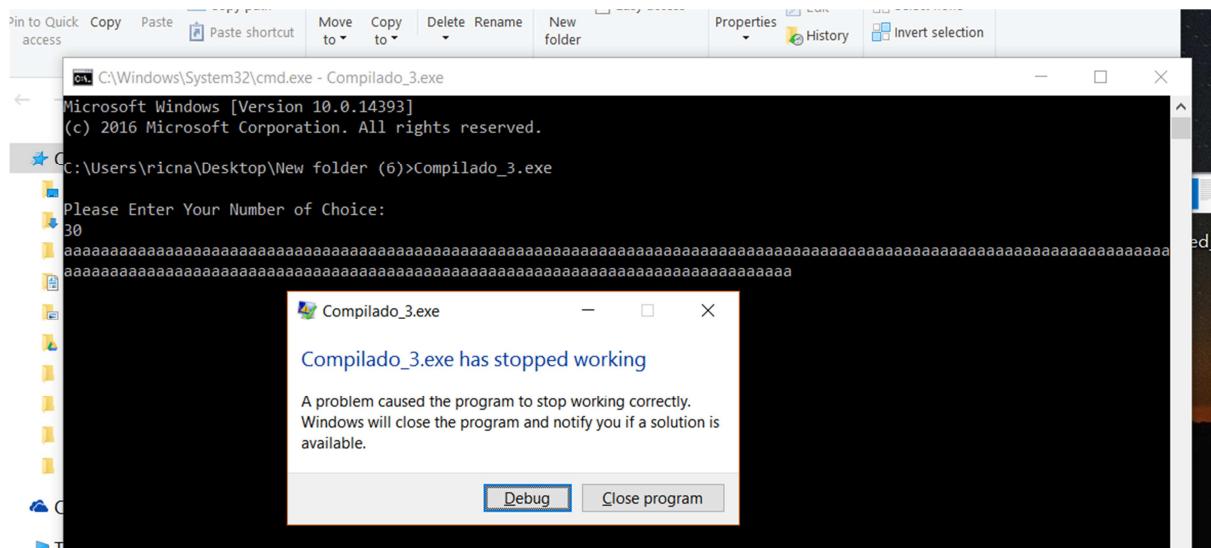
1 #include "stdafx.h"
2
3 int main(int argc, char *argv[])
4 {
5     char buf[0x300];
6     int size;
7     int c;
8
9     printf("\nPlease Enter Your Number of Choice: \n");
10
11    scanf_s("%d", &size);
12    while ((c = getchar()) != '\n' && c != EOF);
13
14    gets_s(buf, size);
15
16
17
18
19
20    return 0;
21 }

```

Many times, the user knows the data size to be entered. If so, that size should be checked well to avoid overflows.

```
ConsoleApplication4 (Global Scope)
4     #include "stdafx.h"
5
6     int main(int argc, char *argv[])
7     {
8         char buf[0x10]; ←
9         int size;
10        int c;
11
12        printf("\nPlease Enter Your Number of Choice: \n");
13
14        scanf_s("%d", &size); ←
15        while ((c = getchar()) != '\n' && c != EOF);
16
17        gets_s(buf, size);
18
19
20        return 0;
21 }
```

There, we see a 0x10-byte buffer or 16 decimal. The user has the possibility of typing the size of the gets_s buffer. There are no max value checks. If I compile it and execute it (Compilado_3.exe).



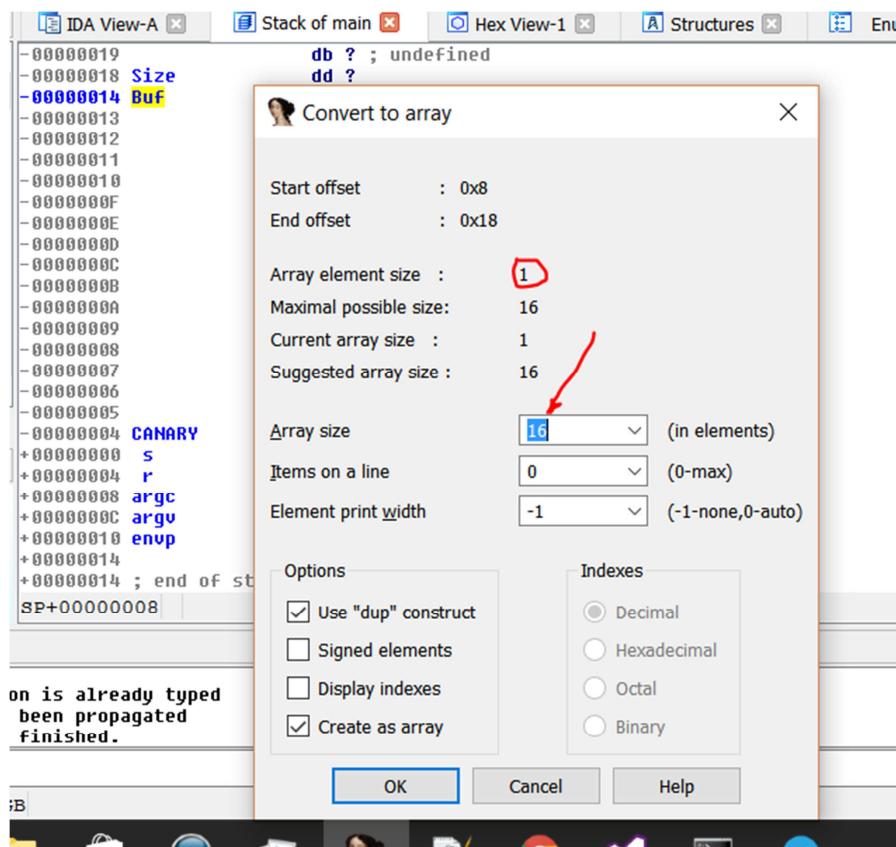
There is a BUG and the program is vulnerable. If we loaded in IDA even if we didn't have the source code.

```

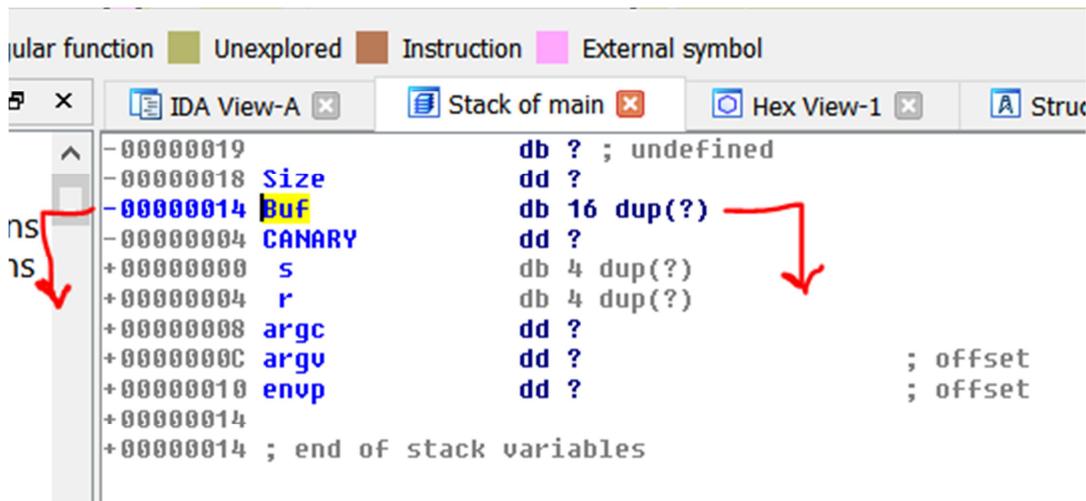
00401090 ; int __cdecl main(int argc, const char **argv, const
00401090 main proc near
00401090
00401090     Size= dword ptr -18h
00401090     Buf= byte ptr -14h
00401090     CANARY= dword ptr -4
00401090     argc= dword ptr  8
00401090     argv= dword ptr  0Ch
00401090     envp= dword ptr  10h
00401090
00401090     push    ebp
00401091     mov     ebp, esp
00401093     sub     esp, 18h
00401096     mov     eax, _security_cookie
00401098     xor     eax, ebp
0040109D     mov     [ebp+CANARY], eax
004010A0     push    esi
004010A1     push    offset aPleaseEnterYou ; "\nPlease Enter Your
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

```

The CANARY is there as before. Just above, it is the Buf. Let's see its size in the stack view.



The buffer length is 16 bytes by 1 that is the length of each element. So, it is 0x10 hex.



If we could copy more than 16 bytes in the buffer, it would overflow and overwrite the CANARY, STORED EBP and the RETURN ADDRESS.

Let's see the size variable.

```
00401071 mov    esp, ebp
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 Num
004010A6 call   printf
004010AB lea    eax, [ebp+Size]
004010AE push   eax
004010AF push   offset aD          ; "%d"
004010B4 call   scanf_s
004010D0 mov    esi, ds:4000h
```

After printing the “Please enter Your Number” message with `printf`, it calls `scanf_s` to enter a number with the keyboard that is saved in the size variable which is a dword and it receives the address with LEA.

Let's see the `scanf_s` function.

scanf_s, _scanf_s_l, w

Visual Studio 2015 | [Otras versiones](#) ▾

Publicado: octubre de 2016

Para obtener la documentación más reciente de Visual :

Lee datos con formato del flujo de entrada estándar. Es describe en [Características de seguridad de CRT](#).

Sintaxis

```
int scanf_s(
    const char *format [, 
    argument]...
);
```

The `scanf_s` function reads data from the `stdin` standard input stream and writes the data to the location provided in `argument`. Each argument must be a pointer to a variable of a type that corresponds to a type specifier in `format`. If the copying takes place between overlapping strings, the behavior is undefined.

So it's like the opposite of `printf`. Instead of printing with a format, it enters from console with a format to a Buffer, in this case the format is "%d" by which it is interpreted like a decimal number.

This way, when it calls gets_s using that size to be typed, it will copy that number of bytes and if it is greater than 0x10 it will overflow.

The screenshot shows assembly code in a debugger. A specific instruction, `004010CC jnz short loc_4010C2`, is highlighted with a red bracket and a green arrow pointing to the instruction. A red arrow also points from the assembly code area to the highlighted instruction. The assembly code is as follows:

```
004010CE
004010CE loc_4010CE: ; Size
004010CE push [ebp+Size]
004010D1 lea eax, [ebp+Buf]
004010D4 push eax ; Buf
004010D5 call ds:_imp__gets_s
004010DB mov ecx, [ebp+CANARY]
004010DE add esp, 8
004010E1 xor ecx, ebp
004010E3 xor eax, eax
004010E5 pop esi
004010E6 call __security_check_cookie
004010EB mov esp, ebp
004010ED pop ebp
004010EE retn
004010EE main endp
004010EE
```

A possible solution would be to check the length of the size before copying.

The screenshot shows C code in a debugger. A conditional statement `if (size > 0x10) { exit(1); }` is highlighted with a red bracket and a green arrow pointing to the `if` keyword. A red arrow also points from the code area to the highlighted conditional. The C code is as follows:

```
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;
```

It would be good to analyze it to see if this solution makes it not vulnerable or it still is, analyze it. It is called VULNERABLE_o_NO.exe and we will discuss it in the next part.

Ricardo Narvaja

Translated by: @IvinsonCLS