



Bypassing Address Space Layout Randomization

Toby 'TheXero' Reynolds

April 15, 2012

Contents

1	Introduction	3
2	Method 1 - Partial overwrite	3
3	Method 2 - Non ASLR	5
4	Method 3 - Brute force	6
5	Conclusion	6

1 Introduction

Most modern day Operating Systems include some form of memory protection such as DEP and ASLR. This article will focus on ASLR, its implementation, limitations and finally various techniques which can be used to circumvent the protection.

With a very basic Buffer Overflow vulnerability, you would normally overwrite EIP with a return address which has a JMP instruction to where you can find your shellcode. What ASLR does here is randomize the base addresses of the system core libraries so that you will not reliably be able to get your desired JMP instruction to get to your shellcode and thus will only crash the application resulting in no code execution.

For ASLR to be effective, all running process and libraries need to be compiled with ASLR in mind so that they can be loaded at different memory addresses after each reboot.

Address	Hex	Instruction
760FFAE1	FFE4	JMP ESP
760FFAE3	0300	ADD EAX, DWORD PTR DS:[EAX]
760FFAE5	0076 18	ADD BYTE PTR DS:[ESI+18], DH
760FFAE8	81FF E5030000	CMP EDI, 3E5
760FFAE E	0F86 05080000	JBE USER32.761002F9
760FFAF4	81FF E8030000	CMP EDI, 3E8
760FFAF A	0F87 F9070000	JA USER32.761002F9
760FFB00	FF75 14	PUSH DWORD PTR SS:[EBP+14]
760FFB03	FF75 10	PUSH DWORD PTR SS:[EBP+10]
760FFB06	57	PUSH EDI
760FFB07	50	PUSH EAX
760FFB08	E8 0A000000	CALL USER32.760FFB17
760FFB0D	E9 F8070000	JMP USER32.7610030A
760FFB12	90	NOP
760FFB13	90	NOP
760FFB14	90	NOP
760FFB15	90	NOP
760FFB16	90	NOP
760FFB17	8BFF	MOV EDI, EDI

Address	Hex	Instruction
75DAFAE1	FFE4	JMP ESP
75DAFAE3	0300	ADD EAX, DWORD PTR DS:[EAX]
75DAFAE5	0076 18	ADD BYTE PTR DS:[ESI+18], DH
75DAFAE8	81FF E5030000	CMP EDI, 3E5
75DAFAEE	0F86 05080000	JBE USER32.75DB02F9
75DAFAF4	81FF E8030000	CMP EDI, 3E8
75DAFAF A	0F87 F9070000	JA USER32.75DB02F9
75DAFB00	FF75 14	PUSH DWORD PTR SS:[EBP+14]
75DAFB03	FF75 10	PUSH DWORD PTR SS:[EBP+10]
75DAFB06	57	PUSH EDI
75DAFB07	50	PUSH EAX
75DAFB08	E8 0A000000	CALL USER32.75DAFB17
75DAFB0D	E9 F8070000	JMP USER32.75DB030A
75DAFB12	90	NOP
75DAFB13	90	NOP
75DAFB14	90	NOP
75DAFB15	90	NOP
75DAFB16	90	NOP
75DAFB17	8BFF	MOV EDI, EDI

The above screenshots look at exactly the same instructions in USER32.DLL however their addresses in memory vary slightly (760F vs 75DA) and this is because this is a system core library with which ASLR has been enabled.

2 Method 1 - Partial overwrite

If you remember from a few years ago the infamous ANI exploit, all versions of Microsoft Windows up to and including Windows Vista were vulnerable. As ASLR was not implemented before Windows Vista, the exploitation process on older systems was fairly simple, just a means of jumping to a PTR EBX then making a couple of short jumps across the ANI header until you land at the beginning of your shellcode.

With Windows Vista and ASLR, this meant that you couldn't just look at a system library like SHELL32.DLL to get to the chosen register as the base address would also change after each reboot. As this crash happens in the USER32.DLL library, what was done for the Windows Vista exploit was really quite special and a partial overwrite of EIP is used to achieve the required jump to the beginning of our ANI file.

Monitoring Internet Explorer under a debugger while crashing it using the ms07-017 proof of concept exploit, we see that we have fully overwritten EIP with 43434343.

```

Registers (FPU)
EAX 41414141
ECX 00000000
EDX 00000000
EBX 02FEF5FC
ESP 02FEF540
EBP 42424242
ESI 02FEF574 ASCII "an ih$"
EDI 02FEF540
EIP 43434343
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDA000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

Registers (FPU)
EAX 41414141
ECX 00000000
EDX 00000002
EBX 02F0F214
ESP 02F0F158
EBP 42424242
ESI 02F0F18C ASCII "an ih$"
EDI 02F0F158
EIP 761A4343 USER32.761A4343
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDA000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)

```

By shortening out buffer by two bytes, we can achieve a partial overwrite of EIP and bypass ASLR. We have to edit the of the header in the ANI file, otherwise it may be not be possible to get a working return address as you would have to use a bruteforce method (not fun).

By only overwriting the two lower bytes of EIP for an address in USER32.DLL and leaving the remaining half blank, causes the latter two bytes to not change, from from what they were originally so we just need to find the OP code so that we will we have gained code execution and enabled us to successfully bypass ASLR.

No register points directly to our buffer, however the first 4 bytes of EBX point to a memory address which leads to the beginning of our ANI header, so we need to find a JMP PTR [EBX] instruction inside the current library USER32.DLL.

Once found we take the higher two bytes to replace the 4343 for our partial overwrite of EIP

```

Immunity Debugger - iexplore.exe - [CPU - thread 00000F1C, module USER32]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?

761A700B FF23 JMP DWORD PTR DS:[EBX]
761A700D C3 RETN
761A700E 1E PUSH DS
761A700F 76 2C JBE SHORT USER32.761A703D
761A7011 C3 RETN
761A7012 1E PUSH DS
761A7013 ^76 81 JBE SHORT USER32.761A6F96
761A7015 E3 FF JECXZ SHORT USER32.761A7016
761A7017 7F FF JG SHORT USER32.761A7018
761A7019 FFE9 JMP FAR ECX
761A701B DB
761A701C 6200 BOUND EAX,QWORD PTR DS:[EAX]
761A701E 0033 ADD BYTE PTR DS:[EBX],DH
761A7020 C0E9 ED SHR CL,0ED
761A7023 6200 BOUND EAX,QWORD PTR DS:[EAX]
761A7025 00FF ADD BH,BH
761A7027 75 28 JNZ SHORT USER32.761A7051
761A7029 FF75 24 PUSH DWORD PTR SS:[EBP+24]
761A702C FF75 20 PUSH DWORD PTR SS:[EBP+20]
761A702F 33C0 XOR EAX,EAX
761A7031 66:837B 02 01 CMP WORD PTR DS:[EBX+2],1
761A7034 8575 01 JZ SHORT USER32.761A7035

Registers (FP
EAX 41414141
ECX 00000000
EDX 00000002
EBX 02E0F45C
ESP 02E0F3A0
EBP 42424242
ESI 02E0F304
EDI 02E0F3A0
EIP 761A700B
C 0 ES 0023
P 1 CS 001B
A 0 SS 0023
Z 1 DS 0023
S 0 FS 003B
T 0 GS 0000
D 0
O 0 LastErr
EFL 00000246

```

As you can see above we have set a breakpoint on our return address as we execute that JMP PTR instruction we are taken to the very beginning of our ANI file. We can't simple replace the ANI header with our shellcode because then Windows wouldn't recognize the file as an ANI file and will not cause the crash, but luckily since we landed at the beginning there are no bytes that will mess up our stack before we can reach a couple of bytes which will allow us to make a

short jump of we have found a few bytes within the ANI header which can be replaced and will still cause the crash.

At bytes 5 -6 of our ANI header we can replace these bytes with a short jump of 22 bytes and from here we can use another 2 bytes to jump 123 bytes and to a payload of our choice.

3 Method 2 - Non ASLR

Another method that works quite well, and is similar to the method above which is a partial overwrite is to use a hard coded address of an existing non ASLR process or library.

For instance, let's assume you have found a bug in a music media player and you're on a system with ASLR such as Windows Vista and these types of programs tend to load several libraries at runtime, which could be something like MP3.DLL etc.

In this example we are going to port an exploit for 'Free MP3 CD Ripper 1.1' onto Windows Vista and the original exploit can be found here <http://www.exploit-db.com/exploits/17727/>. As the current exploit's return address is pointing to a system core DLL which is loaded at '76B43ADC' this is the precise address for a JMP ESP instruction specifically on 'Windows XP SP3 English'. This means that this exploit is not likely to result in code execution on a 'Windows XP SP2 English' machine or even if the target application is running on anything other than 'Windows XP SP3 EN.'

By looking for a JMP ESP instruction without resulting to a system library, not only are your chances of making the exploit universal but avoiding ASLR may also become a possibility.

While searching the loaded program libraries for 'Free MP3 CD Ripper 1.1' I noticed that neither the program itself or its loaded libraries come with support for ASLR, which means that they will get loaded into the same memory address each time they are opened. I was then able to study the loaded libraries for a reliable JMP ESP instruction, however no such an instruction was available.

This meant that I had to look inside the program itself for a JMP ESP to get to my shellcode to achieve full control over the process. This technique will only work however if the process or library which holds the instruction is non ASLR but also you have to be aware of bad characters with this as generally a null byte is will either terminate our buffer or something else. In this case however we were lucky as it didn't affect our buffer in anyway and as long as the machine doesn't have DEP in the way, it should work just fine across multiple Operating Systems.

The screenshot shows the Immunity Debugger interface for the process fcrip.exe. The assembly window displays a series of instructions starting from address 00463BE9. The instructions are:

Address	Disassembly	Comment
00463BE9	JMP ESP	
00463BEB	JMP EBX	
00463BED	JMP EDX	
00463BEF	JMP ECX	
00463BF1	JMP EAX	
00463BF3	DB FF	
00463BF4	DB DF	
00463BF5	DB FF	
00463BF6	DB DE	
00463BF7	DB FF	
00463BF8	DB DD	
00463BF9	DB FF	
00463BFA	DB DC	
00463BFB	DB FF	
00463BFC	DB DB	
00463BFD	DB FF	
00463BFE	DB DA	
00463BFF	DB FF	
00463C00	DB D9	
00463C01	DB FF	

The Registers (FPU) window shows the following values:

Register	Value
EAX	00000000
ECX	00001102
EDX	00001102
EBX	41414141
ESP	02F3FED4
EBP	41414141
ESI	41414141
EDI	41414141
EIP	00463BE9 fcrip.00463BE9
C 0	ES 0023 32bit 0(FFFFFF)
P 1	CS 001B 32bit 0(FFFFFF)
A 1	SS 0023 32bit 0(FFFFFF)
Z 0	DS 0023 32bit 0(FFFFFF)
S 0	FS 003B 32bit 7FFD3000
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS

With network based applications which generally end their communication with a null byte, which is generally considered a bad character. Another technique you can try with these types of applications is actually have a look at try to identify any bad characters as there is always the possibility that a bad char will get converted to a null byte or something similar which may help you bypass ASLR in this situation. Once you have bypassed ASLR, like in most instances you find a place for your payload and have control over the target application.

4 Method 3 - Brute force

The last of the techniques is a brute force, where you repeatedly send your exploit to the target until you get a valid return address. This would not be very good when used for a Client Side attack, or if the target service doesn't automatically restart once it has crashed.

This method is both unreliable and very slow as it will require a large number of different base addresses to be brute forced in order to get code execution, so this process take a very long time. As the exploit code will likely need to be sent as a full payload each time, the chances of detection may be much higher especially if any network security devices such as an IDS (Intrusion Detection System) or even an IPS (Intrusion Prevention system) are in place.

Even though this method is horribly unreliable and time consuming, there are public exploits available that use this method when it comes to tackling ASLR, such as the Samba trans2open overflow vulnerability, which is available in the metasploit framework:

<http://www.exploit-db.com/exploits/16861/>

5 Conclusion

Although ASLR was first introduced to the Windows Operating System with Windows Vista, very few software companies (including Open Source vendors) are very slow at implementing this memory protection mechanism. For instance, as of June 2010, Mozilla (makers of the Firefox web browser) had not yet implemented full ASLR support for their browser, also prior to June

2010; Google had not yet implemented Full ASLR into their Chrome web browser. Considering their Open Source Software methodology you would have thought that new memory protection mechanisms like ASLR would have been pretty quick however this is not the case with the majority of vendors.

Microsoft has enabled ASLR support for all recent software packages that they have produced since 2007, which can render some software vulnerable but not exploitable. When by a Buffer overflow vulnerability is not exploitable however it will likely cause a DoS (Denial of Service) which may render an application/service unavailable, which could be an external facing web-server and could potentially have a financial impact.

Although I have really only discussed ASLR under Microsoft Windows Vista, the same techniques will apply to other versions of Windows (supporting ASLR) such as Windows Server 2008 and above, but also Linux based distributions and possibly other systems like the Mac OS and Solaris.