Bypassing Address Space Layout Randomization

Toby 'TheXero' Reynolds

April 15, 2012

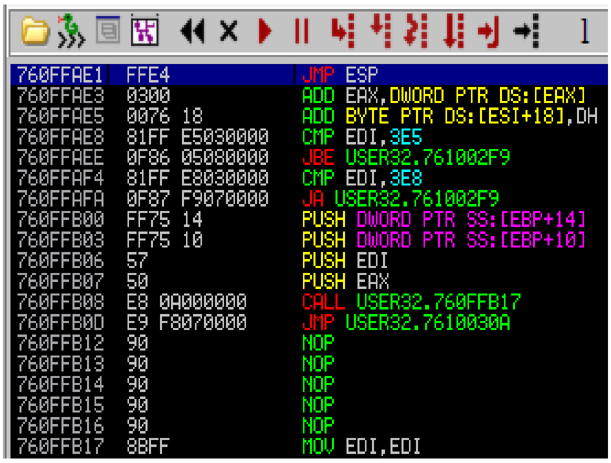
Contents

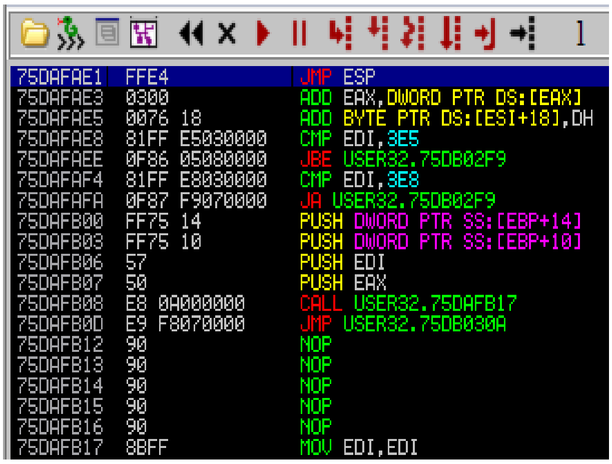
1 Introduction

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.





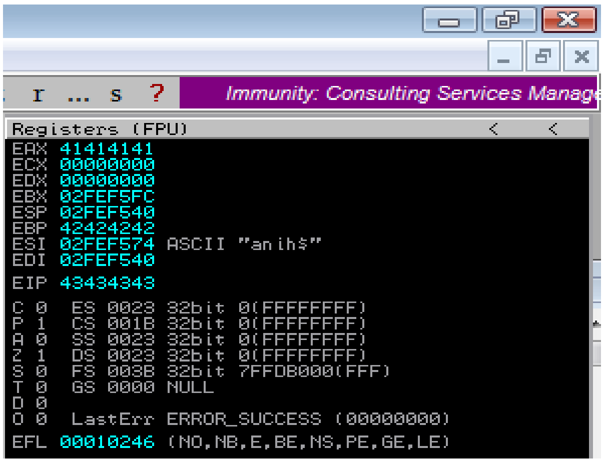
The above screenshots look at exactly the same instructions in USER32.DLL however their addresses in memory vary slightly (760F vs 75DA) and this is because this is a system core library with which ASLR has been enabled.

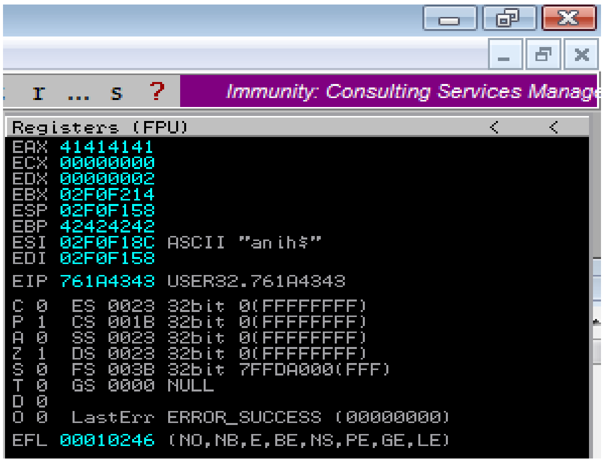
2 Method 1 - Partial overwrite

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 a 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.



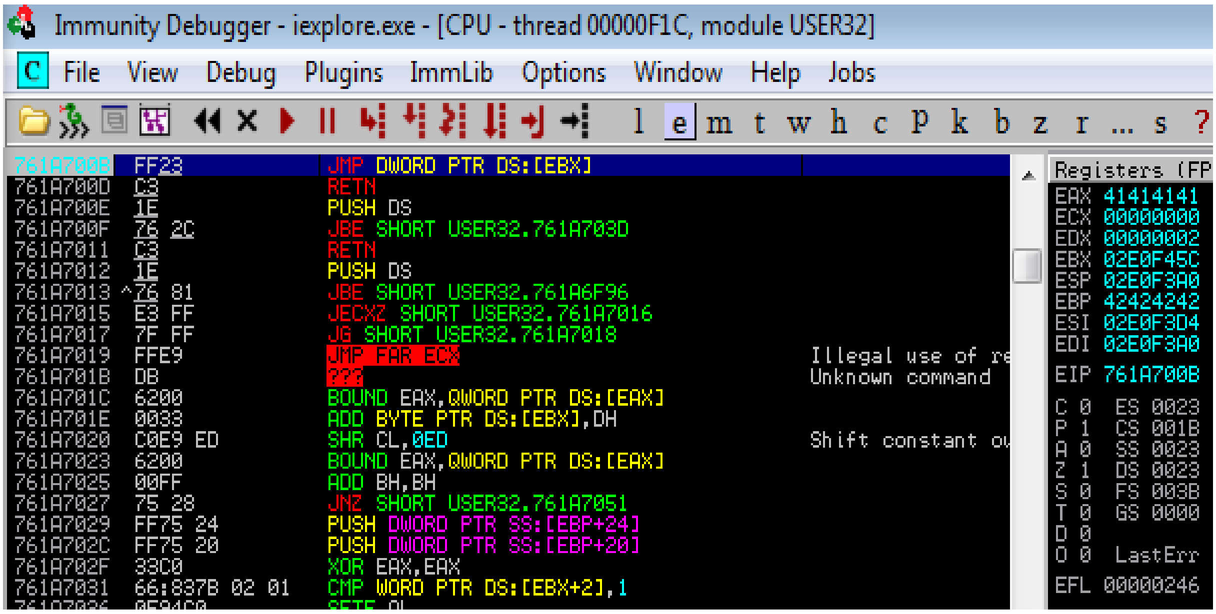


By shortening out buffer by two bytes, we can achieve a partial overwrite of EIP and bypass ASLR. We have to edit the of the header in the ANI file, otherwise it may be not be possible to get a working return address as you would have to use a bruteforce method (not fun).

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 orginally 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 leeds 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



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.

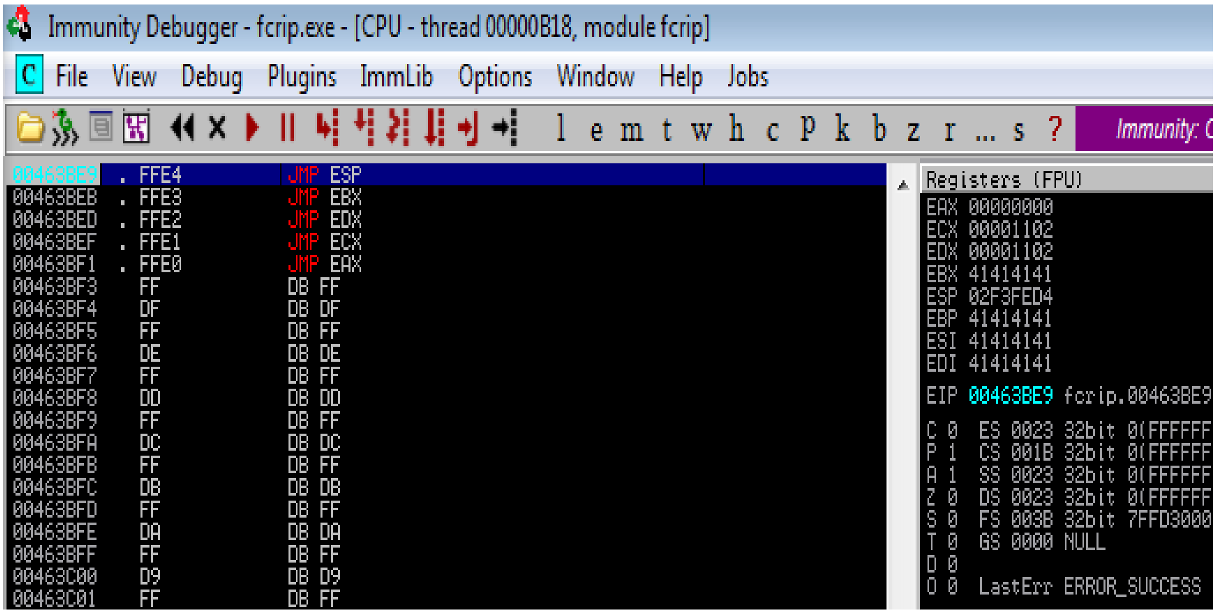
For instance, let's assume you have found a bug in a music media player and you're on a system with ASLR such as Windows Vista and these types of programs tend to load several libraries at runtime, which could be something like MP3.DLL etc.

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.



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.