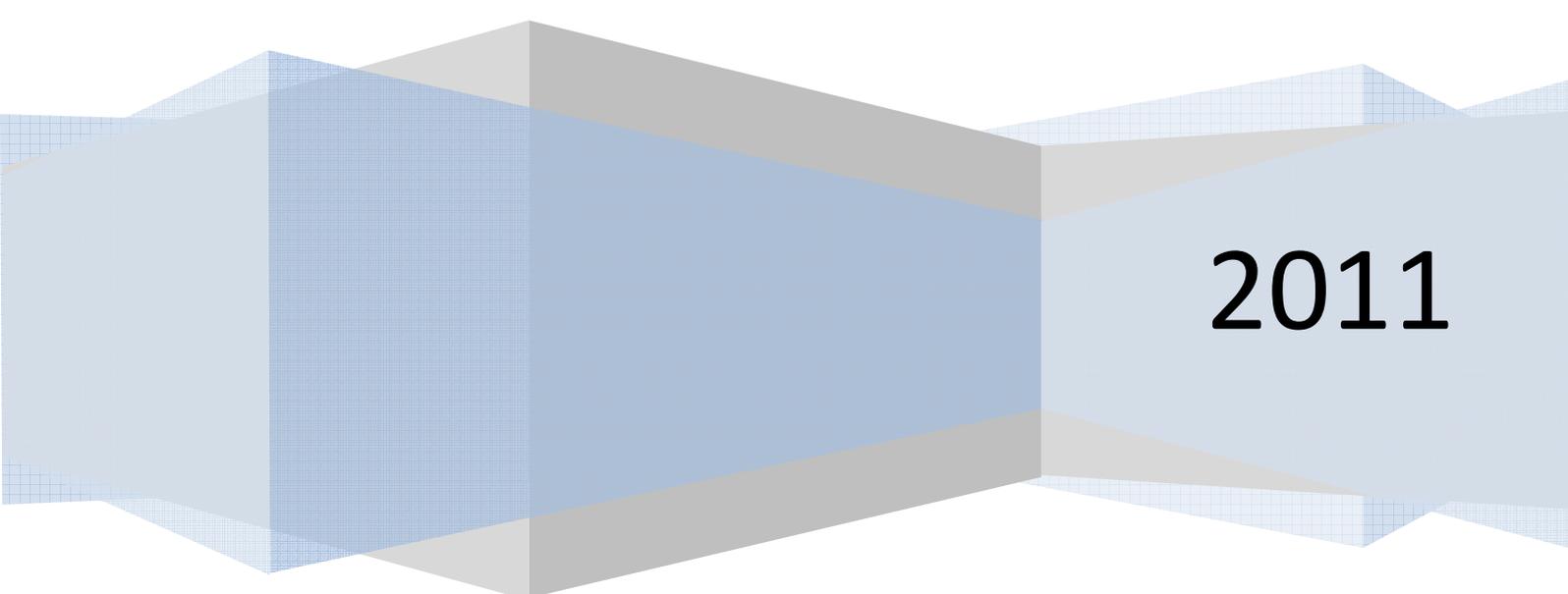


Reversing Microsoft patches to reveal vulnerable code

Harsimran Walia

Computer Security Enthusiast



2011

Abstract

The paper would try to reveal the vulnerable code for a particular disclosed vulnerability, which is the first and foremost step for making undisclosed exploit and patch verification. The process used herein could be used to create vulnerability based signatures which are far better than exploit signatures. Vulnerability signature is a superset of all the inputs satisfying a particular vulnerability condition whereas exploit based signature would only cater to one type of input satisfying that vulnerability condition. This paper would try to pin point the vulnerable code and the files in Microsoft products by reverse engineering the Microsoft patches.

The method used would be to take a binary difference of the file which was patched taken at two different instances, one is the most recent file before patching and the second is after applying the patch but finding the two files is in itself another problem. Windows now releases two different versions of patches, GDR (General distribution) which contains only security related updates and the other QFE (Quick Fix Engineering) or LDR (Limited Distribution Release) which has both security related and functional updates. The problem addressed is that the versions of the two files to be compared should match that is either both should be GDR or LDR. The file after patching can be obtained by extracting the patch of the considered vulnerability. The second file to be compared with a matching version with the first one could be extracted from some other vulnerability patch addressing the issue with the same software disclosed just before the vulnerability considered. The process of extraction of files from patches differs in Vista and Windows 7 from the traditional way used in Windows XP.

After obtaining the correct files to be compared, the next step would be to get a binary difference between the files which can be done very easily and effectively with the use of a tool called DarunGrim. The tool provides a well illustrated difference between the subroutines in the term of percentage match between them. Subroutines from both the files can be viewed in graph mode and can be compared to find the vulnerability. The change in the code is done to fix that particular vulnerability which may be removal of a piece of code and addition of another. Another problem arises at this point is that compiler optimizations happen every-time a code is compiled, so if both the files are compiled with different compilers or compiler versions, they would have different compiler optimizations and that would also show up as a change in code. Simple Instruction reordering keeps happening over different releases which give rise to another problem as when only the instructions are reordered, still it would show up as changed code. The code change in one of the functions out of several functions in the file before applying the patch would be the vulnerable code. From here knowledge of the reverse engineer would come into play as how effectively and fast he can find the vulnerability from the code shown as being changed from the previous file. Till now the process used was static analysis but from now onwards dynamic analysis would be used as breakpoints could be set at these changed functions and run the software. When a breakpoint is hit we can check in which of the functions is user input being dealt. Obtaining all this information can then be used to write an exploit.

This process of reversing the patch and finding the details about the vulnerability would definitely help in creating vulnerability signatures.

Introduction

We start by describing the life cycle of patch development. It starts with a 0day vulnerability being found and used to create an exploit and compromise systems. When the vulnerability reaches the vendor, it finds a fix and releases a patch of the vulnerability to its customer base in order for them to secure their systems from malicious activity. In this paper I will talk about how the patches released by Microsoft can be reverse engineered to exactly locate the code where the vulnerability exists. The paper would also highlight the difficulties faced during the process and how to overcome those difficulties wherever applicable.

In the paper, we would be using DarunGrim, a tool that gives the binary difference very easily and effectively. For a better understanding of the tool and its context of use, I would like to mention its working

How DarunGrim works?

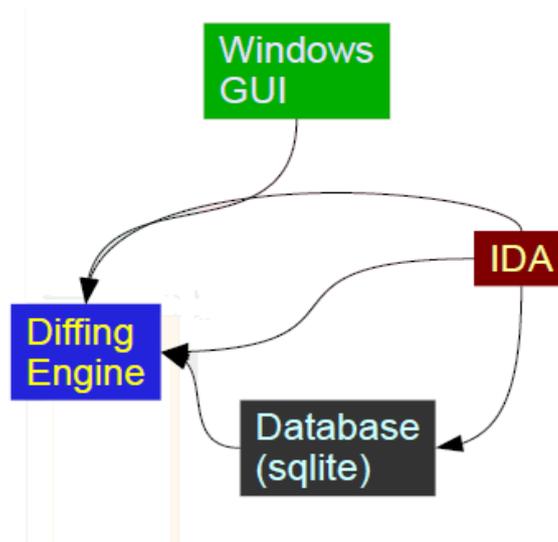


Figure 1 Schema of DarunGrim2

The schema of DarunGrim is shown in the figure above which comprises of sqlite database generated with the help of IDA Pro .The heart of DarunGrim is its Diffing Engine which does all the processing analogous to a CPU in the computer system.

In order to generate diffing results, both the binaries are first disassembled using IDA Pro which runs as a background process and is not visible on the screen. After generating the disassemblies the DarunGrim IDA plug-in is run automatically in the background. Finally both the files are fed to the DiffEngine, which runs and generate the diffing results.

Algorithm ?

The main algorithm used by DarunGrim for binary diffing is called **Basic block fingerprint hash map**. In this algorithm each basic block of assembly code is considered as a single entity and a fingerprint of this basic block is generated from the instruction sequences. The fingerprint is generated with the help of IDA Pro. Two fingerprint hash tables are generated from the basic blocks, one for the original binary and the other for the patched binary.

For the comparison, each unique fingerprint hash from the original binary is searched against the fingerprint hash table of the patched binary for a match. Likewise all the fingerprints from the original binary are marked as matched or unmatched. The main purpose of the comparison exercise is to serve a bigger purpose of finding unmatched functions. In order for a function to match, all the basic blocks inside the function should match. Match rate of the function is calculated based on the fingerprint string matching which is done like GNU diff works i.e. finding the longest common subsequence.

Vulnerability Vs Exploit based signatures

Exploit signatures are created by using byte string patterns or regular expressions. These signatures are exploit specific but are the ones used widely, the main reason being the ease of their creation. Exploit based signature would only cater to one type of input satisfying that vulnerability condition. The problem with these types of signatures is that different attacks can exploit the same vulnerability, in which case exploit based signatures will fail except for the one attack for which it is created. Consider the following exploit signature for a buffer overflow with a long string of A's.

ESig = "docx?AAAAAAAAAAAAA..."

This will stop all the exploits with the pattern shown above but it cannot stop the exploits if I change the A's to B's or any other alphabet.

On the contrary, vulnerability based exploits are based on the properties of the vulnerability and not on the properties of the exploit. Vulnerability signature is a superset of all the inputs satisfying a particular vulnerability condition. For the example above, the vulnerability based signature would be something like

VSig = MATCH_STR (Buffer,"docx?(.*)\$",limit)

The signature matches the string in buffer against the regular expression with the size of the string specified by limit. In this case it is effective against any alphabet which is based on how the vulnerability is actually exploited unlike exploit signature which is created for a particular exploit pattern. For a good vulnerability signature, it should exhibit three properties:

1. It should strictly not allow any false negatives as even one exploit can pwn the system and create a gateway for the attacker into the network.
2. It should allow very few false positives, as too many false positives may lead to a DoS attack for the system.
3. The signature matching time should not create a considerable delay for the software and services.

Need

- Whenever an exploit is to be created and if it is an undisclosed exploit, the first step would be to find the vulnerability and the vulnerable code in order to exploit it.
- To verify if the patch released by Microsoft is working as per it is designed.
- The process can be used to create vulnerability based signatures which are far better than exploit signatures.

Procedure

➤ Finding patches

To start off with, pick any vulnerability and search for the Microsoft security bulletin for that vulnerability. Let's consider MS10-016 for the sake of simplicity in the paper. Go to the Microsoft bulletin page and it will show all the affected OS/Software versions and correspondingly the bulletin just before this, which addresses a similar issue in the same version of OS/Software, in our case it is **None** which means the file version before this should be installed by default in the system, but may not always be the case. Sometimes it would refer to some other bulletin, in which case you should use the file from mentioned patch and not from the installed system.

Problem: Finding the two files is in itself another problem. Windows now releases two different versions of patches, GDR (General distribution) which contains only security related updates and the other QFE (Quick Fix Engineering) or LDR (Limited Distribution Release) which has both security and functional updates. The problem addressed is that the versions of the two files to be compared should match that is either both should be GDR or LDR. Now download the GDR version of the patch for Win XP.

For all supported x86-based versions of Windows XP

File name	File version	File size	Date	Time	Platform	SP requirement	Service branch
Moviemk.exe	2.1.4027.0	3,555,328	23-Oct-2009	14:27	x86	SP2	SP2GDR
Moviemk.exe	2.1.4027.0	3,555,328	23-Oct-2009	14:30	x86	SP2	SP2QFE
Moviemk.exe	2.1.4027.0	3,558,912	23-Oct-2009	15:28	x86	SP3	SP3GDR
Moviemk.exe	2.1.4027.0	3,558,912	23-Oct-2009	14:53	x86	SP3	SP3QFE

For all supported x64-based versions of Windows XP Professional x64 edition

Expand this table

File name	File version	File size	Date	Time	Platform	SP requirement	Service branch
Wmoviemk.exe	2.1.4030.0	3,536,896	26-Oct-2009	22:08	x86	SP2	SP2GDR\W
Wmoviemk.exe	2.1.4030.0	3,536,896	26-Oct-2009	22:04	x86	SP2	SP2QFE\W

[Back to the top](#)

Figure 2 Selection of correct file version

As shown in the image above, the patch would replace the existing Moviemk.exe file to version 2.1.4027.0. Fortunately, the file that came installed by default with XP SP2 system was 2.1.4026.0 which means this patch fixes the issue in the default installation. We would be using these two files for comparison.

Quick workaround: Use open source 'ms-patch-tools' queryMSDB.py python file to get the versions of the files to be compared. Use the file versions from two consecutive advisories.

```

C:\WINDOWS\system32\cmd.exe
C:\ms-patch-tools\msPatchInfo>C:\ms-patch-tools\msPatchInfo\queryMSDB.py -f moviemk.exe -p "XP"
-Querying db patch-info.db...
+
+
+++ MS10-016 +++
+
!-- [MOVIEMK.EXE 2.1.4027.0 <x86/x86>
!-- [MOVIEMK.EXE 2.1.4027.0 <x86/x86>
+
+
+++ MS10-050 +++
+
!-- [MOVIEMK.EXE 2.1.4028.0 <x86/x86>
  
```

Figure 3 ms-patch-tools

➤ Extraction of files

The traditional way of extracting the files from the patches that are delivered as exe file is:

```
<patchfilename>.exe /x
```

This works only till Windows XP and earlier versions of Windows.

Problem: The patches for Windows Vista and Win 7 are delivered as msu files and the way to extract them is completely different from the traditional method as:

Create a folder with the name "MSUFolder" in C: and enter following commands

1. `expand -F:* <Saved_MSU_File_Name>.msu C:\MSUFolder`
2. `expand -F:* <Saved_MSU_File_Name>.cab C:\MSUFolder`

After these commands, lots of files and folders are extracted but use the file inside the folder which has the correct GDR version of file to be compared.

➤ Binary Differencing

Using DarunGrim to get the binary difference between the files selected to prepare for the analysis step. Select both the binary files and give an output file in DarunGrim as shown in the figure and let it do the rest.

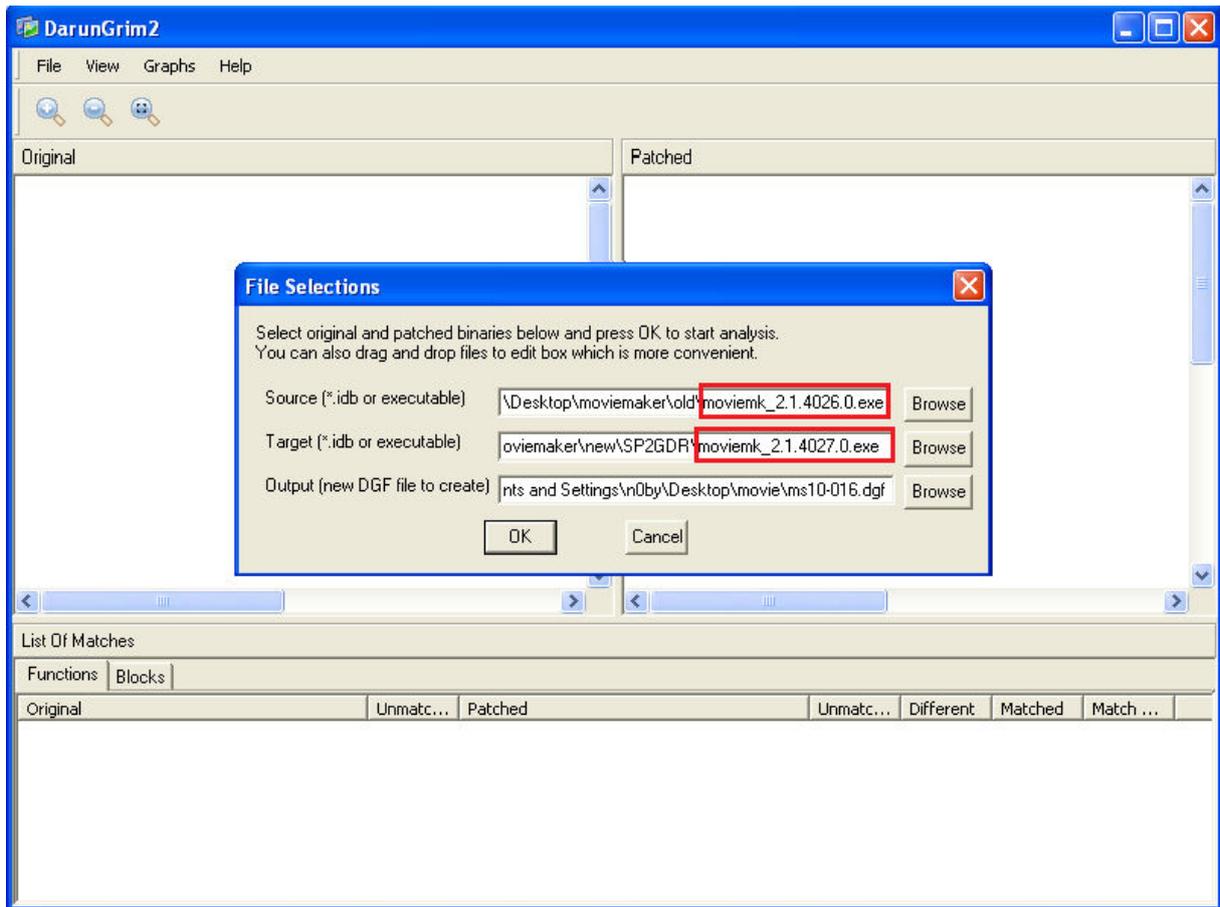


Figure 4 DarunGrim file selection

The tool matches every single function by the algorithm explained earlier between the two files and gives an illustrated detail on the percentage match for all the functions. The functions being patched to fix the vulnerability would definitely have percentage match less than 100.

Functions	Blocks						
Original	Unmatched	Patched	Unmatched	Different	Matched	Match...	
<input type="checkbox"/> sub_103C21D	11	sub_103C66A	12	0	9	43%	
<input type="checkbox"/> sub_116A3D7	0	sub_116AA93	0	1	0	50%	
<input type="checkbox"/> loc_128F99B	0	loc_128FF2E	0	0	4	72%	
<input type="checkbox"/> sub_11808C5	24	sub_1180C0D	17	34	84	72%	
<input type="checkbox"/> sub_11E76C8	1	sub_11FD696	0	1	0	77%	
<input type="checkbox"/> sub_11E1548	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E14BF	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E1436	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E13AD	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E1324	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E1298	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E1212	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> sub_11E1189	1	sub_11FD696	0	1	3	77%	
<input type="checkbox"/> loc_1295442	0	loc_1295627	1	0	2	80%	
<input type="checkbox"/> loc_128FA3C	0	loc_128FB92	1	0	2	80%	
<input type="checkbox"/> sub_1046C86	0	sub_1031C82	0	3	6	83%	
<input type="checkbox"/> loc_1283C00	0	loc_1283D5E	2	1	8	85%	
<input type="checkbox"/> loc_1277406	0	loc_1277564	2	1	8	85%	
<input type="checkbox"/> loc_12766F0	0	loc_127684E	2	1	8	85%	
<input type="checkbox"/> loc_1273573	0	loc_12736D1	2	1	8	85%	
<input type="checkbox"/> loc_12734C6	0	loc_1273624	2	1	8	85%	
<input type="checkbox"/> sub_10FD94B	0	sub_113D629	0	2	5	85%	
<input type="checkbox"/> sub_11F3A14	1	sub_11FD696	0	1	7	88%	
<input type="checkbox"/> sub_11BE1A8	0	sub_11BD35B	0	4	14	88%	
<input type="checkbox"/> sub_117F7E5	2	sub_117FB2A	6	5	48	88%	
<input type="checkbox"/> func_12CB590	13	func_12CB82C	10	4	131	90%	
<input type="checkbox"/> sub_117F252	0	sub_117F581	3	4	31	90%	
<input type="checkbox"/> sub_109302D	0	sub_1032B98	0	2	9	90%	
<input type="checkbox"/> sub_121E1C9	0	sub_1219158	0	1	5	91%	
<input type="checkbox"/> sub_11147B2	1	sub_1114242	0	2	17	92%	
<input type="checkbox"/> sub_11824A0	0	sub_11826FC	0	1	7	93%	
<input type="checkbox"/> sub_11145EB	0	sub_1114242	0	2	17	94%	
<input type="checkbox"/> sub_116B84B	0	sub_116B0DE	0	2	28	96%	

Figure 5 Darungrim results

Problem: Not every function whose Match percentage is less than 100% is the function changed. There are several problems like

Instruction reordering

A lot of reordering happens over different releases which breaks the matching algorithm and marks the same blocks as unmatched.

Split blocks

A block in the graph which has only parent and the parent has only one child leads to a split block which causes a problem in the matching process. This can be improved if the two blocks can be merged and treated as a single block.

Hot patching

Instructions like mov eax, eax at the start of functions are a sign of hot patching which also lead to a mismatch in the block. The solution to this is just ignoring the instruction mentioned before.

These problems create false positives that even same functions are shown different, which would have to be eliminated by manual inspection of the functions.

Patched version

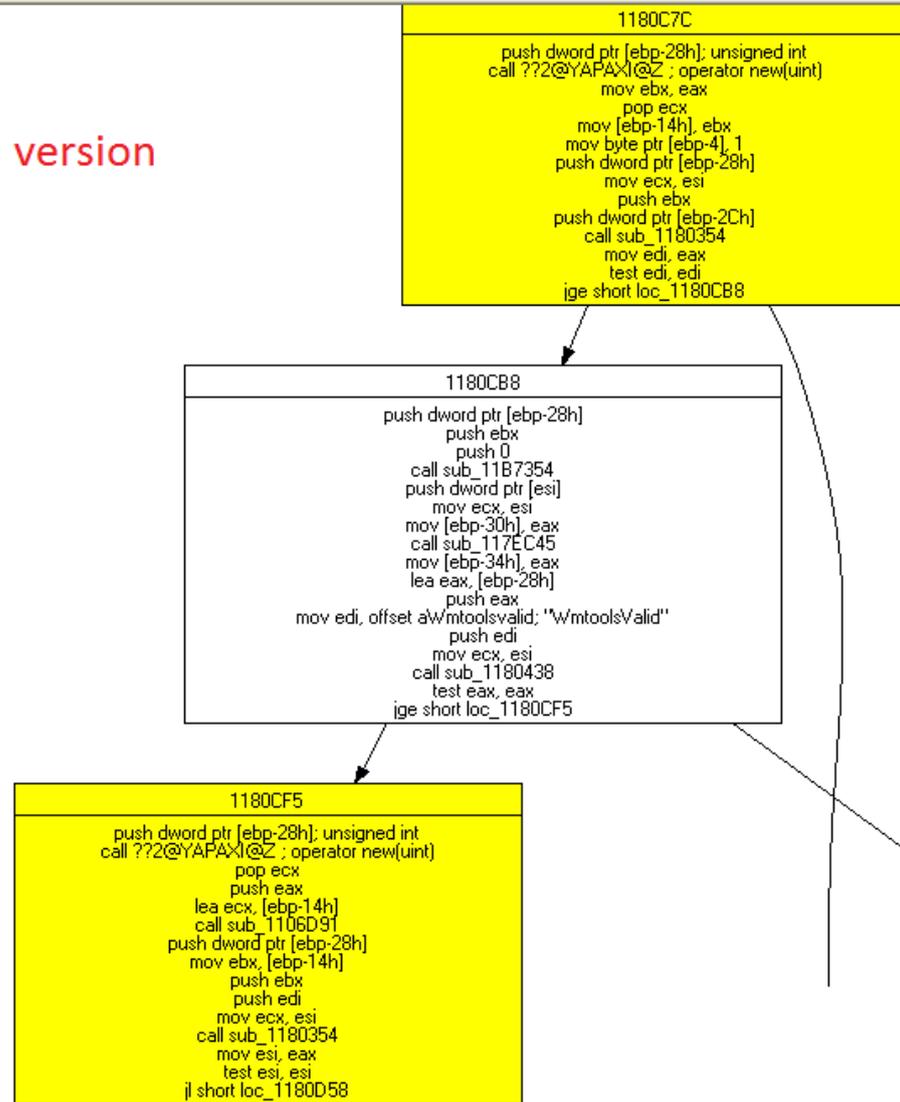


Figure 7 Patched version of moviemk.exe

Binary Analysis of the unpatched function

```

push    [ebp-2Ch]      ; unsigned int
call    ???@YAPAXI@Z  ; operator new(uint)
mov     ebx, eax
pop     ecx
mov     [ebp-18h], ebx
mov     [ebp-3Ch], ebx
mov     byte ptr [ebp-4], 1
push   dword ptr [ebp-2Ch]
mov     ecx, esi
push   ebx
push   [ebp-30h]
call   sub_118000C func(const *,void *,long)
mov     edi, eax
test   edi, edi
jge    short loc_1181503

```

When the first time new function is called, the pointer to the space is stored in **ebx** and a function **sub_118000C** is called and this pointer is passed as an argument. After doing a little bit of reverse engineering we could see that the function is used to fill content in the allocated space.

```
push    [ebp-2Ch]                ; unsigned int
call    ???@YAPAXI@Z            ; operator new(uint)
pop     ecx
mov     [ebp-14h], eax           ; ebp-14h = pBuffer
mov     [ebp-40h], eax
mov     byte ptr [ebp-4], 2
push    [ebp-2Ch]
mov     ecx, esi
push    ebx
push    edi
call    sub_118000C func(const *,void *,long)
mov     esi, eax
test    esi, esi
jge     short loc_118158A
```

The second time new function is called, but instead of passing the pointer to the space allocated by the second call to new function, the pointer to the space allocated by the first new call i.e. **ebx** is being passed to the same function **sub_118000C** again to fill in the space which is where the vulnerability might exist. Hence a larger data which was suppose to be copied to second space allocated can be copied to a small space of the first space allocation, causing Buffer overflow.

➤ Debugging

After figuring out the vulnerability, we can go for validating our find by trying to get a crash of the application. This step includes debugging of the application to create a file that can generate the crash. Put a breakpoint at each call to new function and run the application attached to immunity debugger and open a MSWMM windows movie maker file.

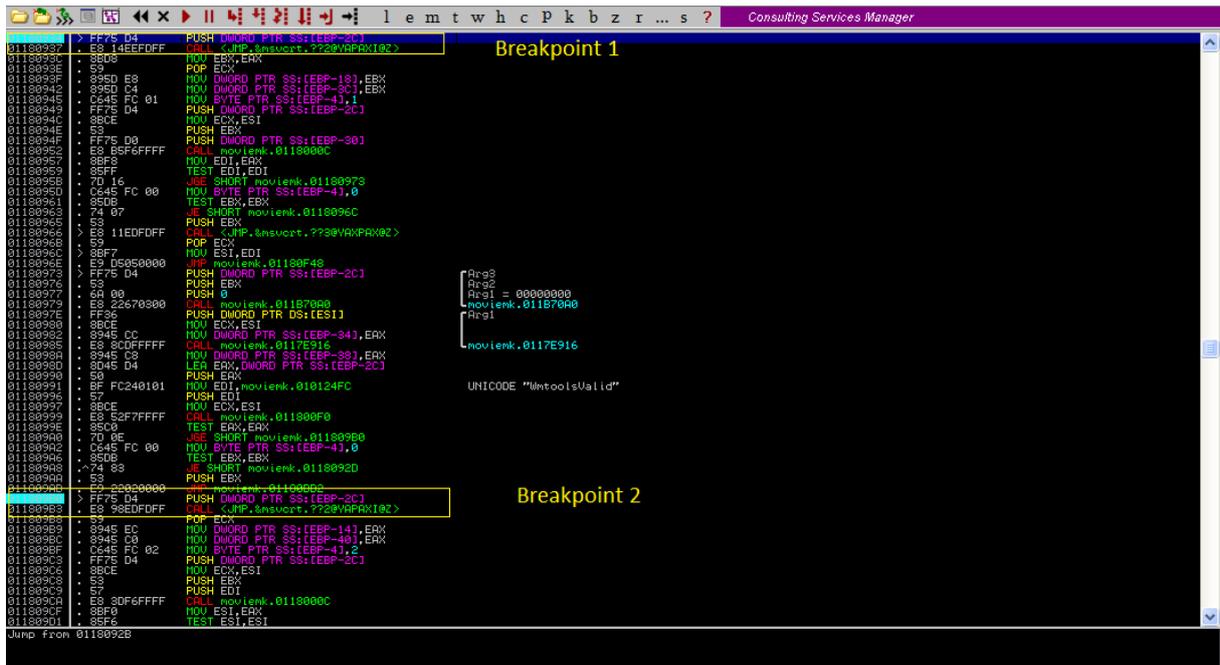


Figure 8 Immunity Debugger showing the two breakpoints

As it hits the breakpoint we can find out the size of the new space being allocated in both cases. Now create a MSWMM file which will make the size of first new call space smaller than the size of the second new call space in order to cause Buffer overflow thereby crashing the application.

Conclusion

This paper is an overview of how the 1-day exploits and vulnerability signatures can be created by reversing the patches supplied by the vendor. An attempt has been made to understand the process involved in reversing and the problems faced during the execution of the process. However, what all concepts have been presented here is needed to be perfect by interested reader via further research and practice. In this paper we have only talked of reversing Microsoft patches which served our purpose; however discussion on other vendor patches is left up to the reader.

Bibliography

1. David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha (May 2006). *Towards automatic generation of vulnerability-based signatures*.
2. Jeongwook Oh (July 2009). *Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries*.
3. Jeongwook "Matt" Oh (July 2010). *Exploit Spotting: Locating Vulnerabilities Out Of Vendor Patches Automatically*
4. Ryan Iwahashi, Daniela A.S. de Oliveira, Jong-Soo Jang (2008). *Towards Automatically Generated Double-Free Vulnerability Signatures Using Petri Notes*.
5. Nabil Schear, David R. Albrecht , Nikita Borisov. *High-speed Matching of Vulnerability Signatures*
6. *Intel Architecture Software Developer's Manual*. Volume 2:Instruction Set Reference
7. <http://www.breakingpointsystems.com/community/blog/microsoft-vulnerability-proof-of-concept/>
8. <http://www.mydigitallife.info/2007/02/15/extract-and-view-contents-of-microsoft-update-standalone-package-msu-for-windows-vista/>
9. <http://www.abyssec.com/blog/>
10. <http://en.wikipedia.org/wiki/Diff>