ExploitSpotting: Locating Vulnerabilities Out Of Vendor Patches Automatically

Black Hat USA 2010, Las Vegas

Jeongwook Oh

oh.jeongwook@gmail.com, mat@monkey.org, http://twitter.com/ohjeongwook

Sr Security Researcher, WebSense Inc.

Security Patch Analysis

There are already many kinds of binary patch analysis systems out there. There are commercial ones and free ones. But the current implementations only concentrate on finding the differences between binaries. What security researchers really need from the patch analysis is security patch. Sometimes it's very hard to locate security patches because they are buried inside normal feature updates. The time for locating the security patches will increase drastically as more feature updates are mixed in the released patches. This is especially true with all the Adobe and Oracle product patches. They tend to mix security patches and feature updates.

In that case, we need another way to boost the speed of the analysis. The automatic way to locate the security patches is that. This can be done by analyzing the patched parts and see if it has some specific patterns that the usual security patches have. For example, an integer overflow will have some comparison against the boundary integer values. And buffer overflow can involve the vulnerable "strcpy" or "memcpy" replaced with safer functions. Even use-after-free type bug has their own patch patterns. We will present all the common patterns that we saw and also present way to locate them using pattern matching. But there can be more thing to be done in addition to this simple approach. You can introduce static taint analysis to binary diffing world. You can trace back all the suspicious variables(expressed as register value or memory location) found in the patch by using binary diffing. And you can see if they are controllable or taint-able from the user controllable input like network packets or user supplied file input.

IPS Signatures

Personally, I worked in IPS industry last 5 years. And I felt that this patch analysis is very important. The IPS and vulnerability scanners need signatures to catch attacker's exploitation attempts. To make signatures you need to understand the vulnerability itself. Many times this involves reverse engineering the patches. You determine what has been changed in the patch and you can know what different input to the vulnerable target can result in different behavior. For example, let's assume that a program has a buffer overflow condition with some data supplied by the user. Let's assume if you provide data with more than 500 bytes then it'll crash. If you're a signature writer for vulnerability scanner, how can you infer the exact number to use for your signature? If you're a IPS rule writer, how do you determine the minimum size of string to detect as suspicious? That involves patch analysis. If you want to perform signature writing correctly, it should always involve patch analysis.

There were some efforts from the vendors, though. Microsoft was the company that were mostly attacked by vulnerability researchers last 10 years. They release patches every 2nd Tuesday of each month. We call it patch Tuesday. So to supply some useful information to the security vendors, they announced a program called MAPP. I expected this program will eliminate the needs for patch analysis

of Microsoft binaries. But I was wrong. There were still many cases that the MAPP didn't cover. It gives you POC and few lines of detection guidelines. And that's it. You might need very specific technical details of the patches. MAPP is not enough in this case. Also, Microsoft is still sneak additional patches along with announced patches and doesn't let the MAPP subscriber or the public to know about that. Basically, MAPP can help you a lot, but if you're a serious about writing good signatures, you're still on your own. You still need patch analysis in this case. That's where this binary diffing thing kicks in.

Finding Security Patches

The purpose of binary diffing in patch analysis is locating security patches as quickly as possible. Sometimes the differential analysis results are not clear because of a lot of noises. The noises are caused by feature updates, code cleanup, massive refactoring, compiler option changes.

Not all patches are security patches. Sometimes, there are too much of noises, it's like finding needles in the sand. We need ways for effectively locating patches with strong security implication.

I suggest two more improvements over traditional binary diffing to achieve effective security patch analysis. One is signature based matching and the other is tracing data flow. Signature based matching is implemented with DarunGrim3 release and tracing data flow functionality will be available with separate package called IDATracker.

Let's look at them one by one.

Signature based matching

So basically this is about signaturing the usual suspects. We already know some patterns by experience. We know something like “strcpy” or “sprintf” can make problems. By applying these patterns to the inserted or deleted basic blocks and finding them inside the blocks you can locate the code parts patching buffer overflow vulnerability. Currently we are only using basic string matching patterns, we have extensive list of vulnerable functions that is usually used by Windows-based software. Also, we have list of secure functions that is supposed to fix old vulnerable functions. One of them is string safe APIs provided and recommend by Microsoft. Also the pattern can be internal symbol name that is used to fix some issues. We can see this pattern from CVE-2010-0249-Vulnerability in Internet Explorer Could Allow Remote Code Execution case.

This is very simple idea but also very powerful. If you look into other fields of security, you can find many places that is using signature concept. Basically anti-virus or anti-spyware is all about patterns and signatures. Web filters are also all about signatures and patterns. You need to collect the samples and need to extract best and optimal signatures to use and put them into the signature database. And you can distribute the updated signatures. This can applied to binary diffing, we will collect patch samples, analyze them and extract best signatures for that vulnerability class. And we make them as a analyzing module or database. The DarunGrim3 module will come with basic set of signatures that we found during our research, but the signatures can be easily extended by 3rd party researchers. Also they can feedback us and we can reflect them to the distributed signatures.

We are going to talk about usual patterns in Patterns Of Vulnerabilities and Patches.

Tracing Data Flow

This is the fundamental way to determine whether some security patch has security implication or not. This is what reverse engineers are actually doing when they are analyzing binary files or patches. Tracing the data flow is the basic skill set for the reverse engineers. But how about you need to do that for massive amount of files? That will be very time consuming and expensive work. We built some basic IDA module that can trace the flow of data automatically. It still needs some human intervention like pointing which variable to analyze. But this can be improved later.

DarunGrim3

DarunGrim3 was intended to fix the issues with traditional binary diffing tools.

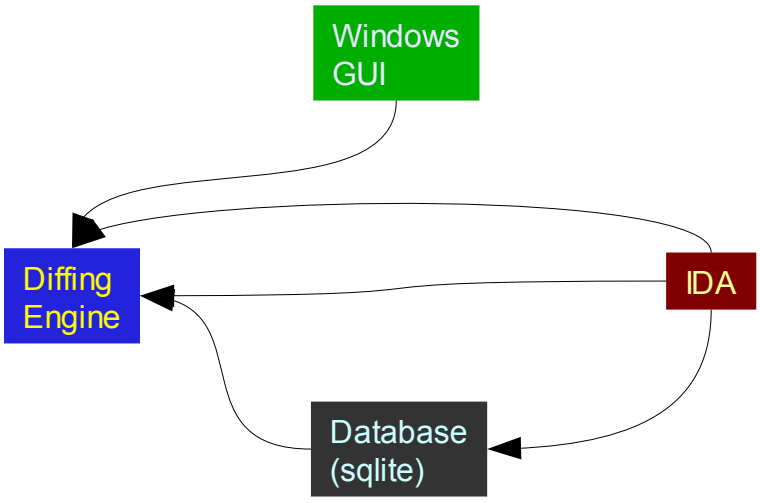
It is totally open-source as DarunGrim2. And it provides Python scripting interface to the "Diffing Engine". The Python layer is a thin layer for the core engine which is written in C++. So there is not that much of performance degradation.

Also, DarunGrim 3 provides basic set of pattern matching which can help to write signatures for vulnerabilities. We calculate "Security Implication Score" using this Python interface. The pattern matching should be easy to extend as the researcher get to figure out new patterns from new instances.

Managing files are boring job. It involves downloading patches, Storing old binaries, loading the files manually by searching the files from your manually constructed file storage. Bin Collector from DarunGrim3 can help you in this case.

You can download the tool and documentations from http://darungrim.org.

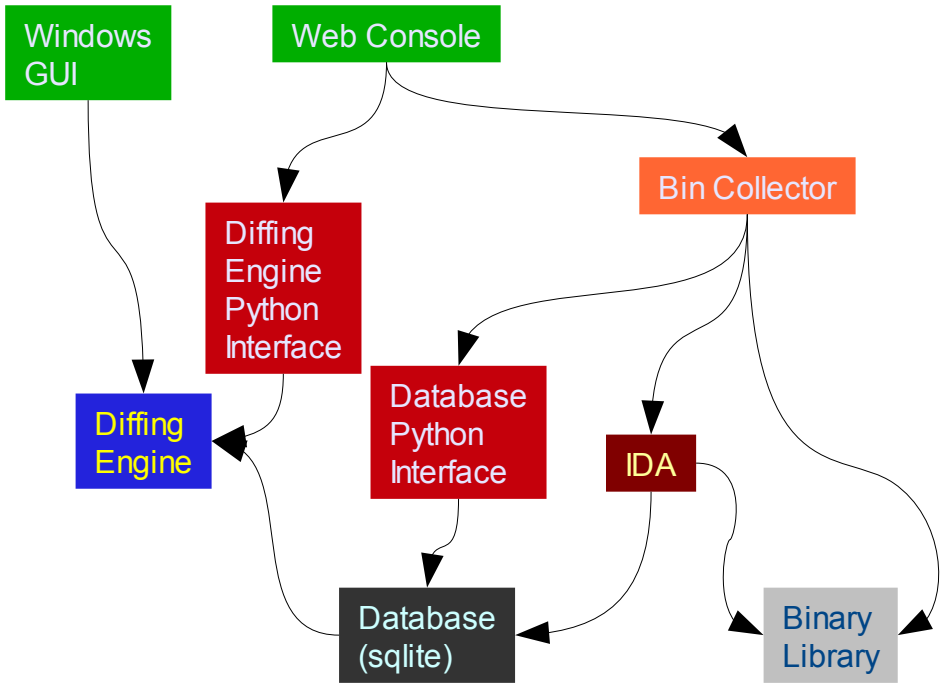
Picture 1 shows you the basic components of DarunGrim2. It heavily depends on IDA for generating disassembly listing from the binaries and store them as sqlite database file. The "Diffing Engine" is the core part of the components and the functions of the engine can be accessed through "Windows GUI" component.



Picture 1: Basic components of DarunGrim2

In contrast to simple and basic DarunGrim2 components, DarunGrim3 is extending it extensively. Look at Picture 2. It includes all the components of DarunGrim2, but still extending it wrapping the engines

and database using Python. All the new additional components are mainly written in Python. They are not performance intensive components but involves more of the database and web interactions, and I think Python is a good choice in this case.



Picture 2: Basic components of DarunGrim3

Python Interface

One of the main Python module DarunGrim3 provides is “DarunGrimEngine” module, it exposes the “Diffing Engine” functionality from the core engine. You can use this module to initiate diffing process.

The following Python code snippet shows how you can use the engine. Import DarunGrimEngine module and just call a function “DiffFile” from the module. The function will handle all the process of opening the unpatched, patched files and initiating the disassemblying using IDA if there is no idb file generated for the files. Also it will load them to DarunGrim database and run it through “Diffing Engine”. Everything is automated through this one function. And this works like charm. You just need to provide the paths for the target files and output files to generate. You can read the files using DarunGrim Windows GUI or Web Interface after the analysis is done.

```
import DarunGrimEngine


DarunGrimEngine.DiffFile(
        unpatched_filename,
        patched_filename,
        output_filename, log_filename, ida_path
)
```

The next important module is “DarunGrimDatabaseWrapper”. This module provides the abstract class for DarunGrim database. The database itself is a relational database. And this module provides abstraction layer for the database in Python class by utilizing sqlalchemy Python module.

The following code snippet shows an example of querying diffing analysis results from the database and printing it to the console. Basically the abstract class has full access to the database and provides every fields that are in the database tables. Everything that can be done through GUI can be achieved using script.

```
import DarunGrimDatabaseWrapper


database = DarunGrimDatabaseWrapper.Database( filename )
for function_match_info in database.GetFunctionMatchInfo():
        if function_match_info.non_match_count_for_the_source > 0 or function_match_info.non_match_count_for_the_target > 0:
                print function_match_info.source_function_name + hex(function_match_info.source_address) + '\t',
                print function_match_info.target_function_name + hex(function_match_info.target_address) + '\t',
                print str(function_match_info.block_type) + '\t',
                print str(function_match_info.type) + '\t',
                print str( function_match_info.match_rate ) + "%" + '\t',


                print database.GetFunctionDisasmLinesMap( function_match_info.source_file_id,
function_match_info.source_address )
                print database.GetMatchMapForFunction( function_match_info.source_file_id, function_match_info.source_address )
```

Security Implication Score

So how do we determine if a function has more probability of having security patches than other functions? We use index called “Security Implication Score” to calculate and show this probability. DarunGrim3 calculates the number based on signatures. And also each patterns has weight value that will be used to calculate this number.

This index number shows you what functions have more security related patches inside it. It currently employs “signature based matching” we talked about in chapter Signature based matching. The web interface shows you this number and you can sort by this score and start analyzing the most explicit one first to expedite whole analysis process.

Bin Collector

“Bin Collector” is an engine that collects and organizes the binary files. Currently it only supports PE files. It's possible to manually load or store files to the binary library. It maintains indexes and version information on the binary files from the vendors. You can manage the files from Web interface called “Web Console”. For some of Microsoft's binaries, it supports automatic patch download and extraction.

It collects the binaries for you and it looks for matching binary to diff automatically. But the

automation functionality is pretty limited right now, only supporting Microsoft's binaries and only supporting core Windows components for automatic extraction. For example, Office binaries are not supported by this automatic feature, also Windows Vista and Windows 7 binaries are not supported. But still this is a first attempt to automate the binary management for patch analysis and will be covering more types of binaries in the future.

This “Bin Collector” component also exposes a few Python interfaces. It's scriptable if you want automate file collection process in a way you want. The whole code is written in Python. If some parts are not working as you expected, you can easily customize or extend the functionality.

So how can it download the patches automatically? Here's how.

1. It visits each vendors patch pages
2. Use mechanize python package to scrap MS patch pages
3. Use BeautifulSoup to parse the html pages
4. It extracts and archives binary files
5. Use sqlalchemy to index the files
6. Use PE version information to determine store location like following.

   <Company Name>\<File Name>\<Version Name>
7. Copy the files to the destination folder

Web Interface

DarunGrim3 provides web interface called ”Web Console”. Web Console is an user friendly and easy to use interface compared to original DarunGrim2 GUI. By just clicking through the web pages, you can get the diffing results. This is possible combined with Bin Collector engine.

Usage

There are multiple ways to run DarunGrim diffing engine. One is through interactive interface and the other is using scripts like Windows batch file or Python script file.

Using DarunGrim2.exe UI

Just put the path for each binaries to the start-analysis dialog box and DarunGrim2.exe will do the rest of the job. We have a good example of this procedure at Using Darungrim2.exe(Windows GUI).

DarunGrim2.exe + Two IDA sessions

This is the original way to using DarunGrim2 UI, but not recommended anymore. It's too complicated and confusing. But if you still like this approach, it's still supported.

1. First launch DarunGrim2.exe

2. Launch two IDA sessions
3. First run DarunGrim2 plugin from the original binary
4. Secondly run DarunGrim2 plugin from the patched binary
5. DarunGrim2.exe will analyze the data that is collected through shared memory

Using DarunGrim Web Console: a DarunGrim 3 Way

Web Console provides user friendly user interface. It includes "Bin Collector", “Security Implication Score” support

Using DarunGrim2C.exe command line tool

This was supported in DarunGrim2, but many of the public didn't know about this. This is a handy, batch-able, quick way to bulk binary diffing.

Using DarunGrim Python Interface: a DarunGrim3 Way

This method is utilizing DarunGrim3 Python wrapper. You can use DarunGrimEngine module to initiate the analysis and can use DarunGrimDatabaseWrapper module to retrieve the analysis results. The detail is here at Python Interface section.

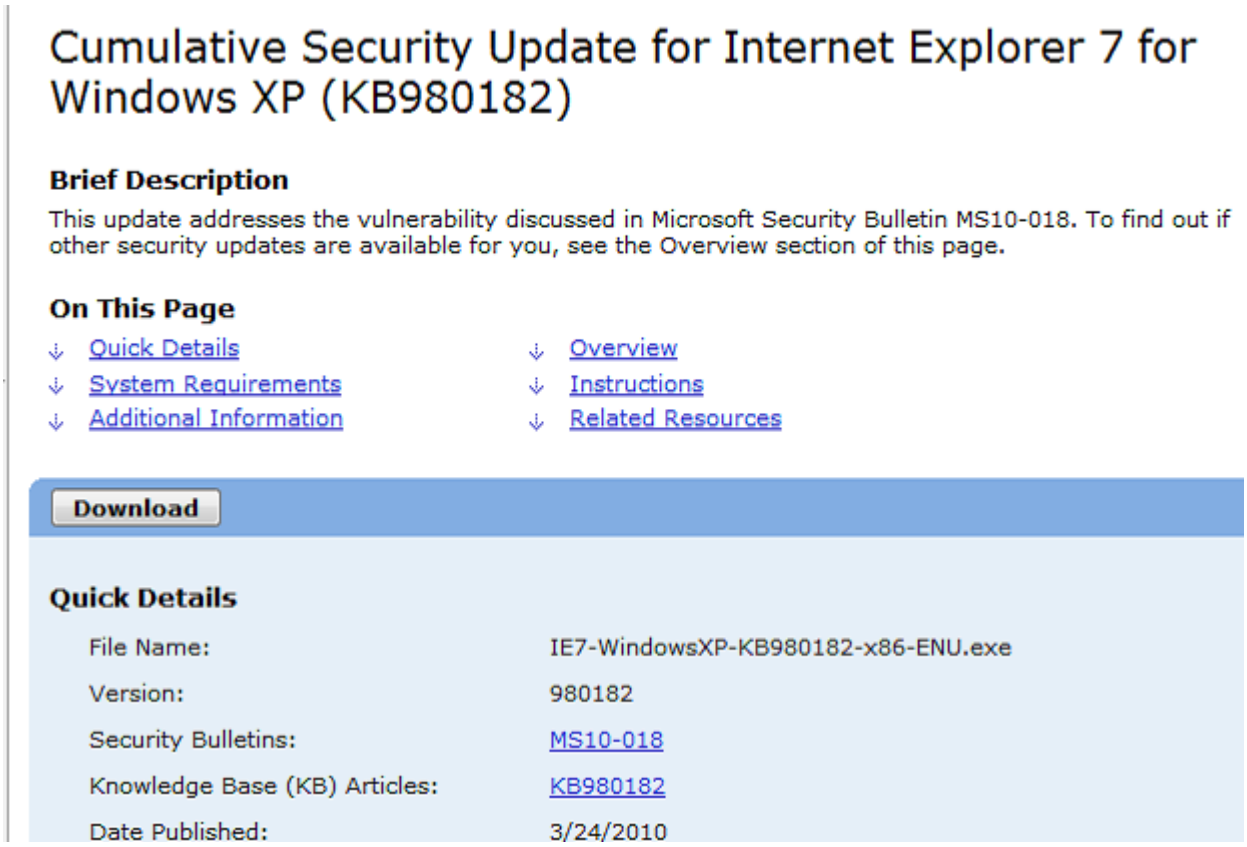
Using Darungrim2.exe(Windows GUI)

You might want to get some basic idea what binary diffing process looks like. Here I'll show you whole process for a typical binary diffing using DarunGrim2 GUI.

The patch(MS10-018) is for “CVE-2010-0806” vulnerability. You can look up the detailed information about the vulnerability from the CVE page.

Use-after-free vulnerability in the Peer Objects component (aka iepeers.dll) in Microsoft Internet Explorer 6, 6 SP1, and 7 allows remote attackers to execute arbitrary code via vectors involving access to an invalid pointer after the deletion of an object, as exploited in the wild in March 2010, aka "Uninitialized Memory Corruption Vulnerability.

Download the patch by visiting patch page(MS10-018) and following the OS and IE version link. For XP IE 7, I used following link from the main patch page to download the patch file from here. The download page typically looks like Picture 3.

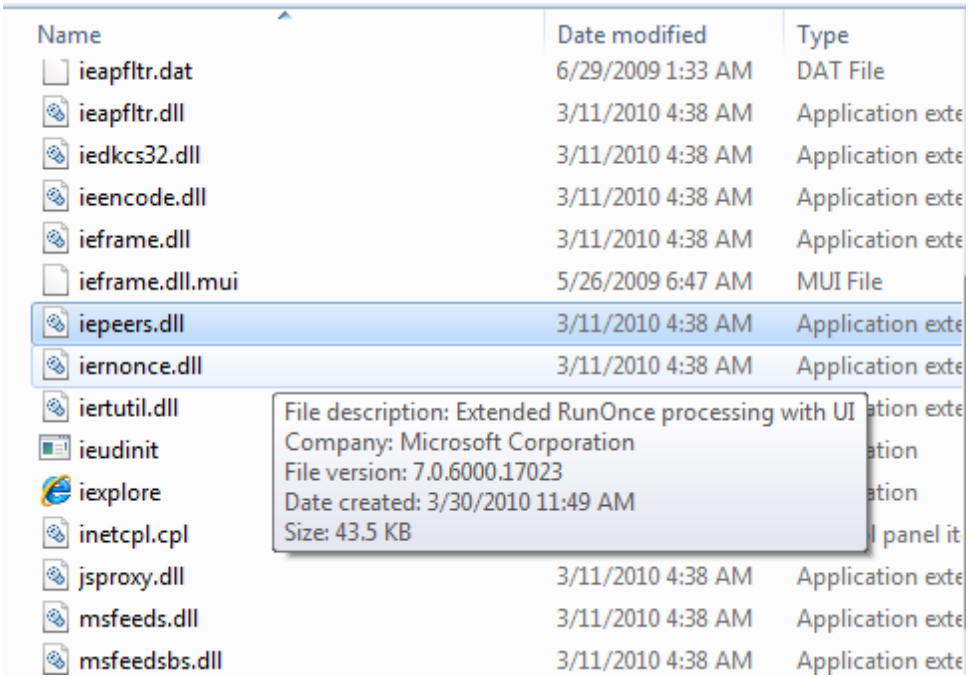


Picture 3: Microsoft patch download page

The downloaded filename is "IE7-WindowsXP-KB980182-x86-ENU.exe" in this case. You can execute following command line from command prompt to extract the files to out folder.

```
IE7-WindowsXP-KB980182-x86-ENU.exe /x:out
```

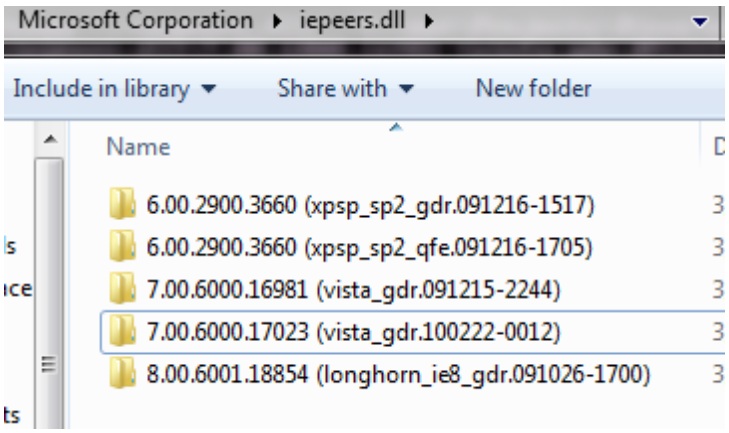
The extracted files looks like Picture 4. We already know that the vulnerable file name is "iepeers.dll". You can see the file's version string is "7.0.6000.17023". You need to note the version string because this is important in the next step.



Picture 4: Extracted files from the patch file

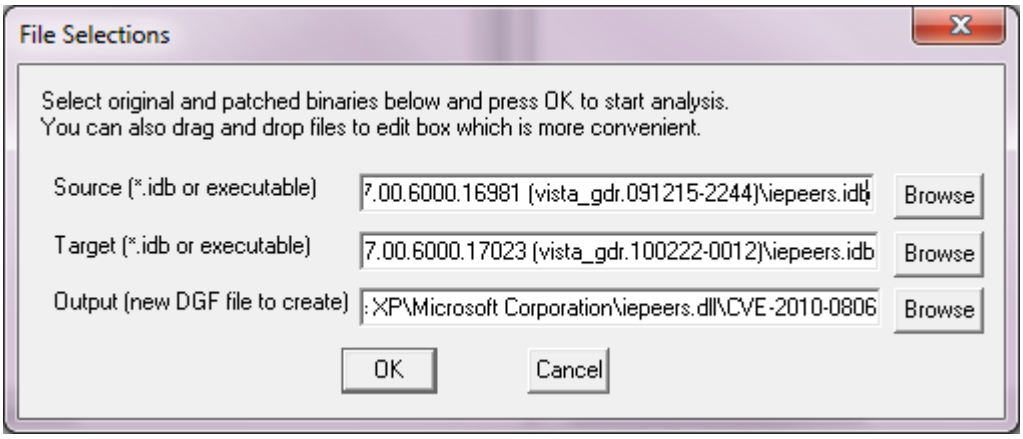
To diff the patched file, you need a file to compare with. You need to collect unpatched files from the operating systems that the patch is supposed to be installed.

I used SortExecutables.exe from DarunGrim2 package to consolidate the files. The files will be organized in a fashion that they reside inside directories with version number strings.



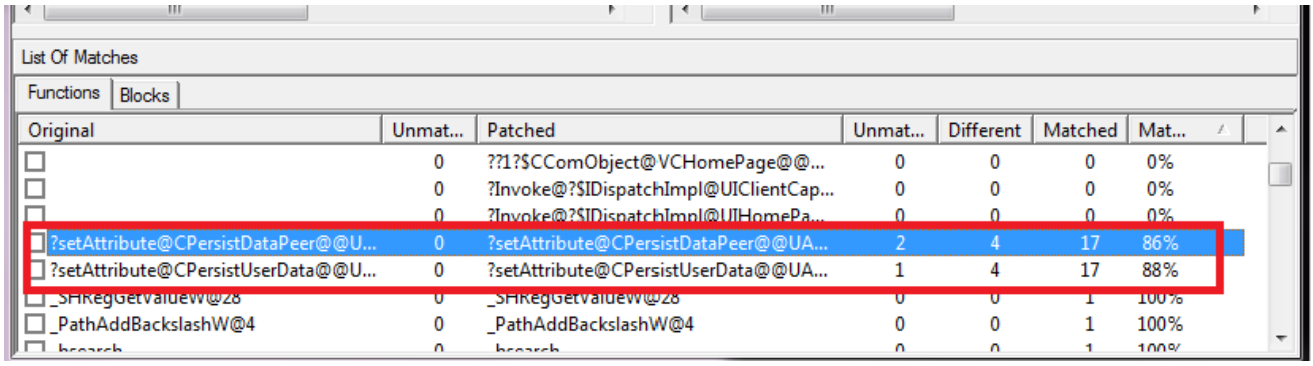
Picture 5: Series of files with different versions

Launch DarunGrim2.exe and select "File → New Diffing from IDA" from the menu. You need to wait from few seconds to few minutes depending on the binary size and disassembly complexity.



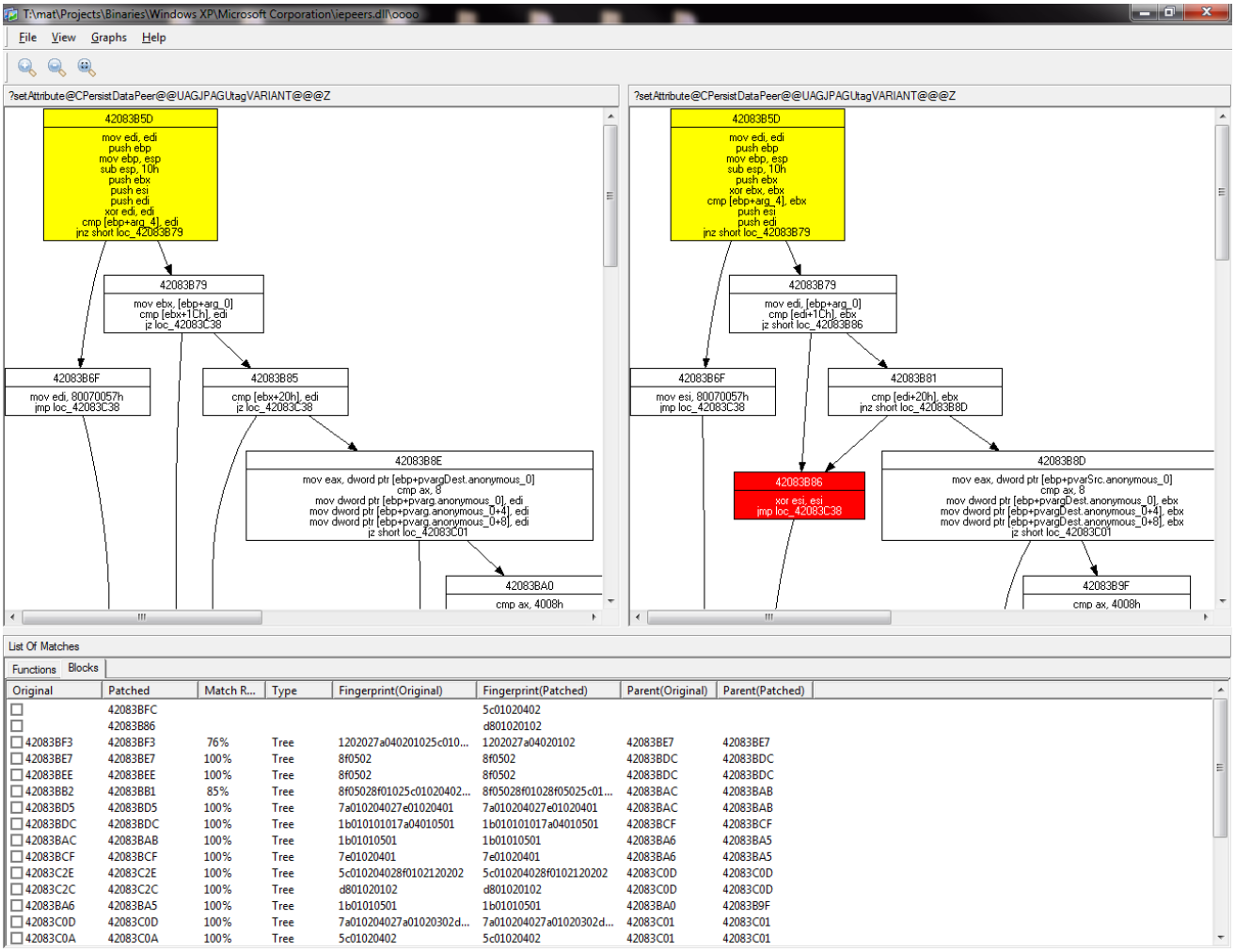
Picture 6: Loading unpatched and patched files from DarunGrim2 GUI

Now you have the list of functions like something shown in Picture 7.



Picture 7: List of functions that has been matched

The next step is finding any eye catching functions. From Picture 7, the match rates(the last column values) 86% and 88% are strong indications that they have some code changes which have security implications. Match rate 0% means there is no matching functions, so you might ignore those functions.



Picture 8: Flow graph showing modified and inserted basic blocks

If you click the function match row, you will get a matching graphs.

Color code meanings are like following table Table 1.

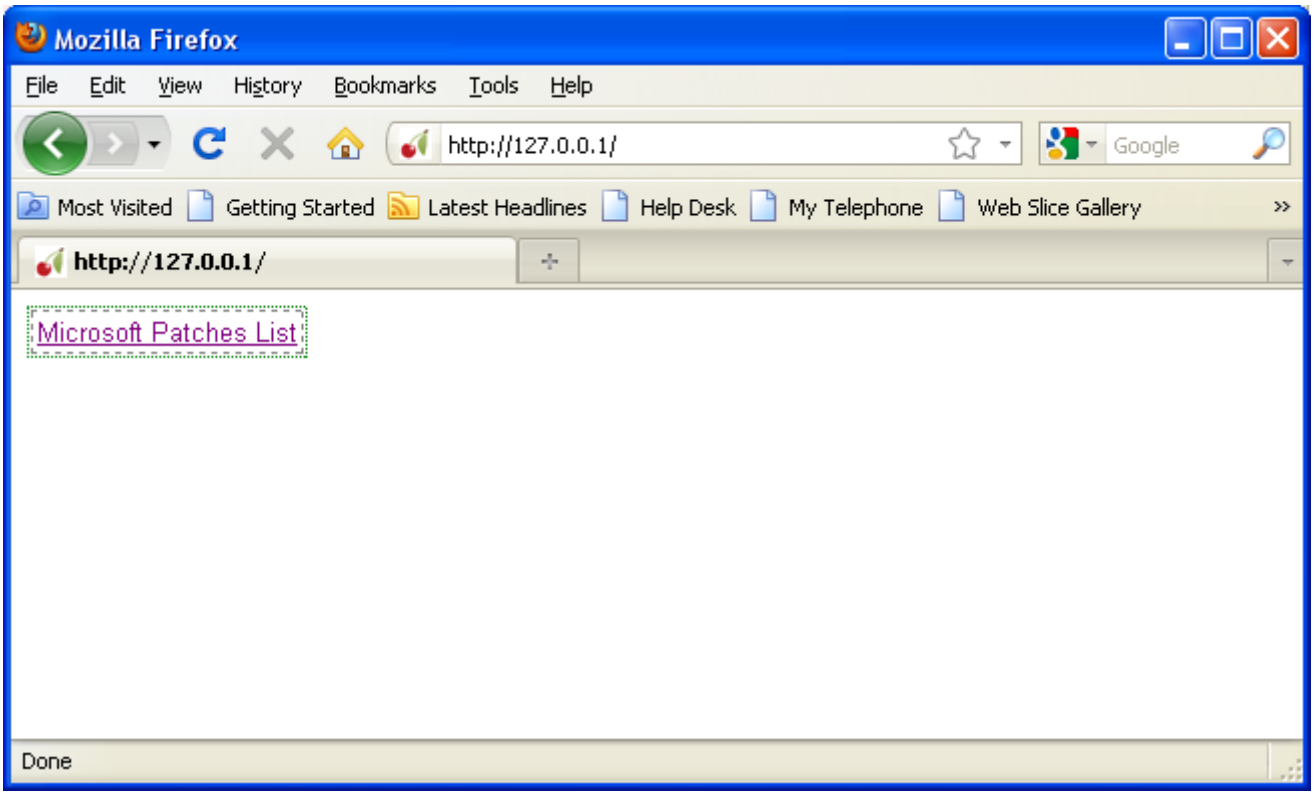
Color	**Meaning**	**Description**
White blocks	Matched blocks	This blocks are same in each pane.
Yellow blocks	Modified blocks	This blocks are modified in each pane.
Red blocks	Unmatched blocks	Unmatched block means that the block is inserted or removed.

Table 1: Meanings of color codes

So in this case, the red block is in patched part which means that block has been inserted in the patch. You need to use reverse engineering skills to track down the root cause of the security issues.

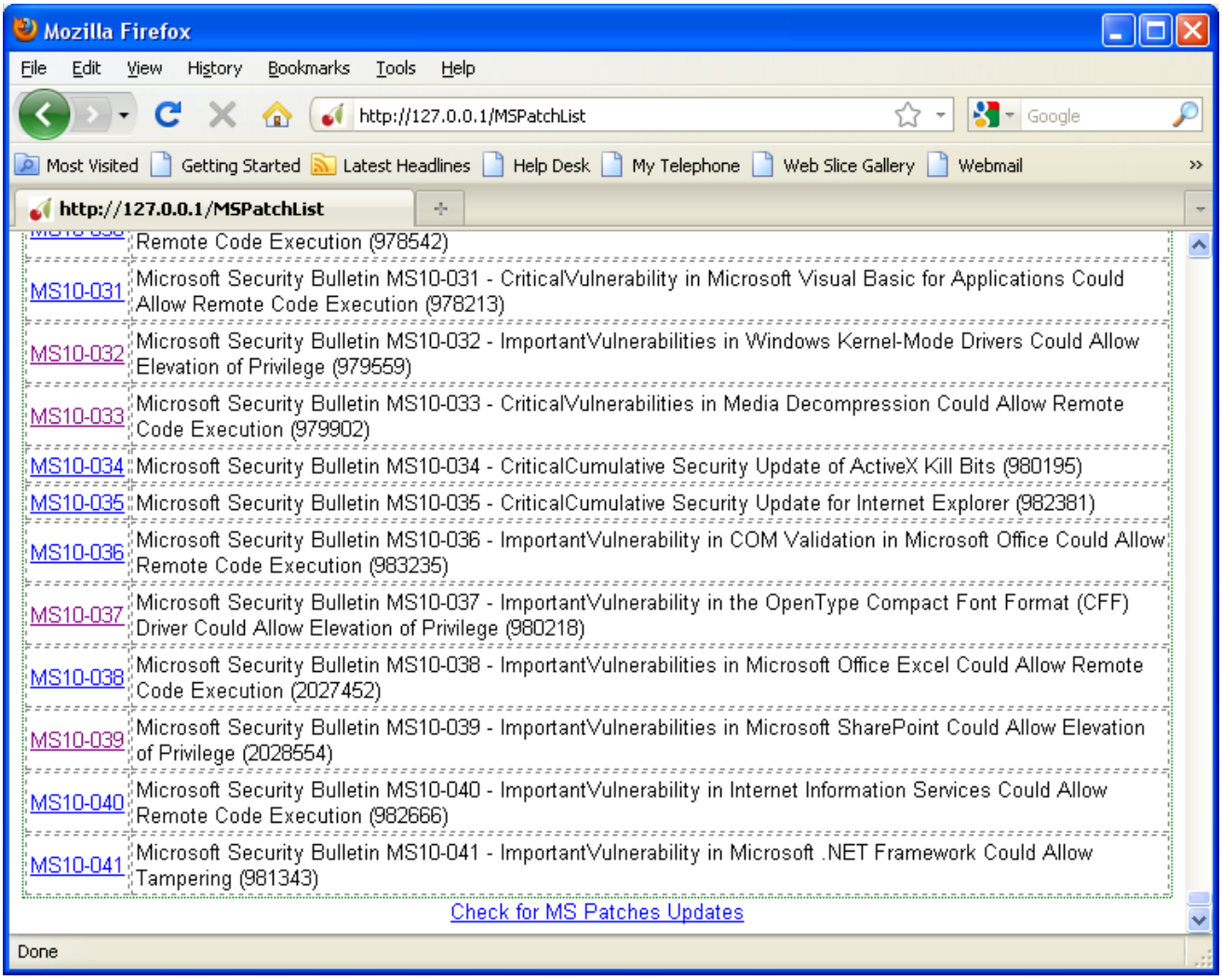
Using Web Console

Let me introduce you through the “Web Console” diffing process. If you launch “WebServer.py” from the package, you can use any browser you have to connect to the Web Console through http://127.0.0.1 hyperlink. Picture 9 shows the main page of the tool. Basically, it just lists “Microsoft patches List” because we support only their binaries right now. The list will grow up soon.



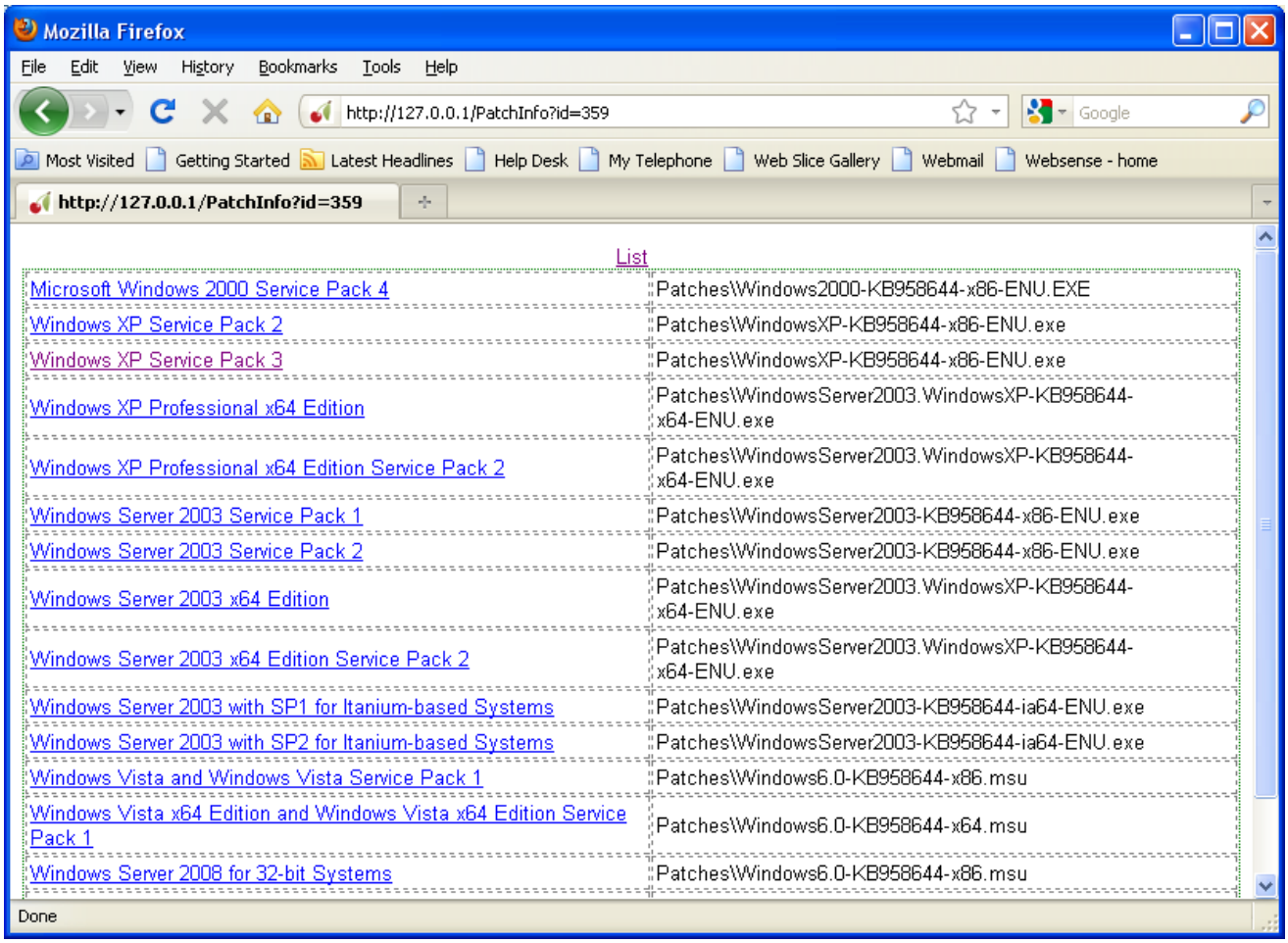
Picture 9: Main page of DarunGrim3 Web Console

If you click the “Microsoft Patches List” link, you can have the full list of Microsoft's patches that you have collected. You just need to click the patches you are interested in.



Picture 10: Microsoft patch list page of Web Console

If you click the patch number you want to analyze, it'll show the list of OSes it supports as seen at Picture 11. Usually Windows XP SP2 or SP3 patches are good choice, because majority of Windows users are still using them.



Picture 11: Microsoft patch os list page of Web Console

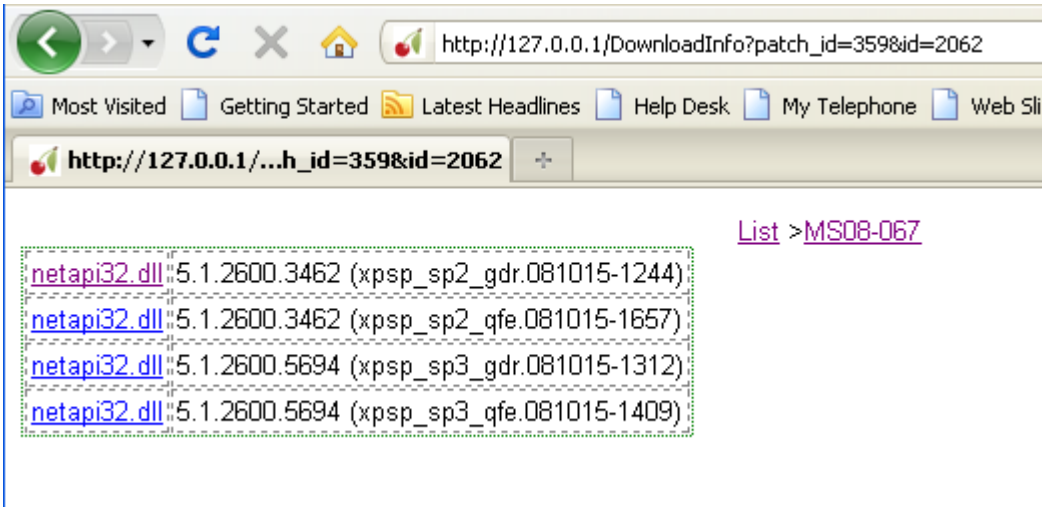
If you choose the OS to analyze by clicking the link, it will show list of files the actual patch file contains. For example, Picture 12 shows list of netapi32.dll for XP SP2 and SP3. Also each service pack has both GDR or QFE releases noted by identifier in the version string.

You might wonder what GDR or QFE means. Here's what I got from googling the Internet for the meaning.

GDR(General Distribution): a binary marked as GDR contains only security related changes that have been made to the binary

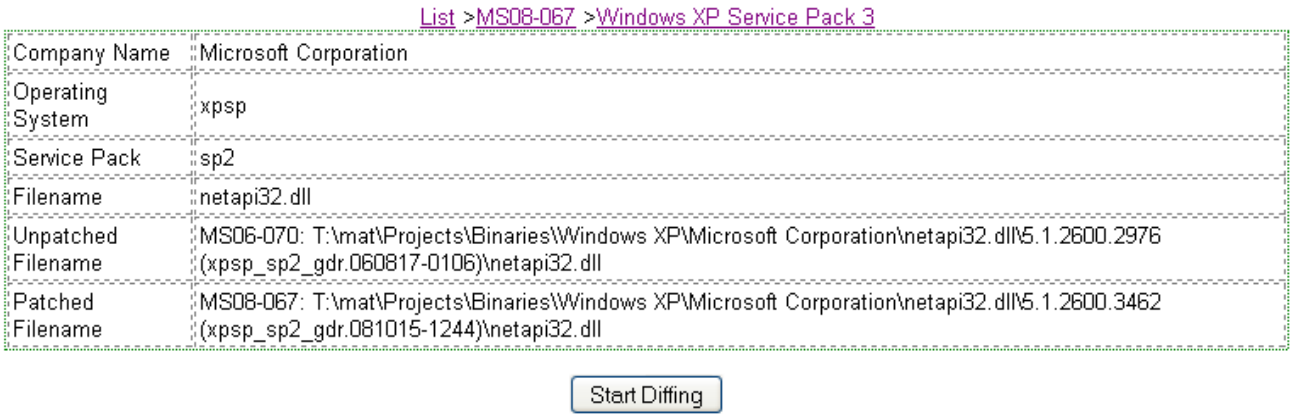
QFE(Quick Fix Engineering)/LDR(Limited Distribution Release): a binary marked as QFE/LDR contains both security related changes that have been made to the binaries well as any functionality changes that have been made to it.

So basically GDR is more general and something most users will get. GDR release is a better candidate for analysis.



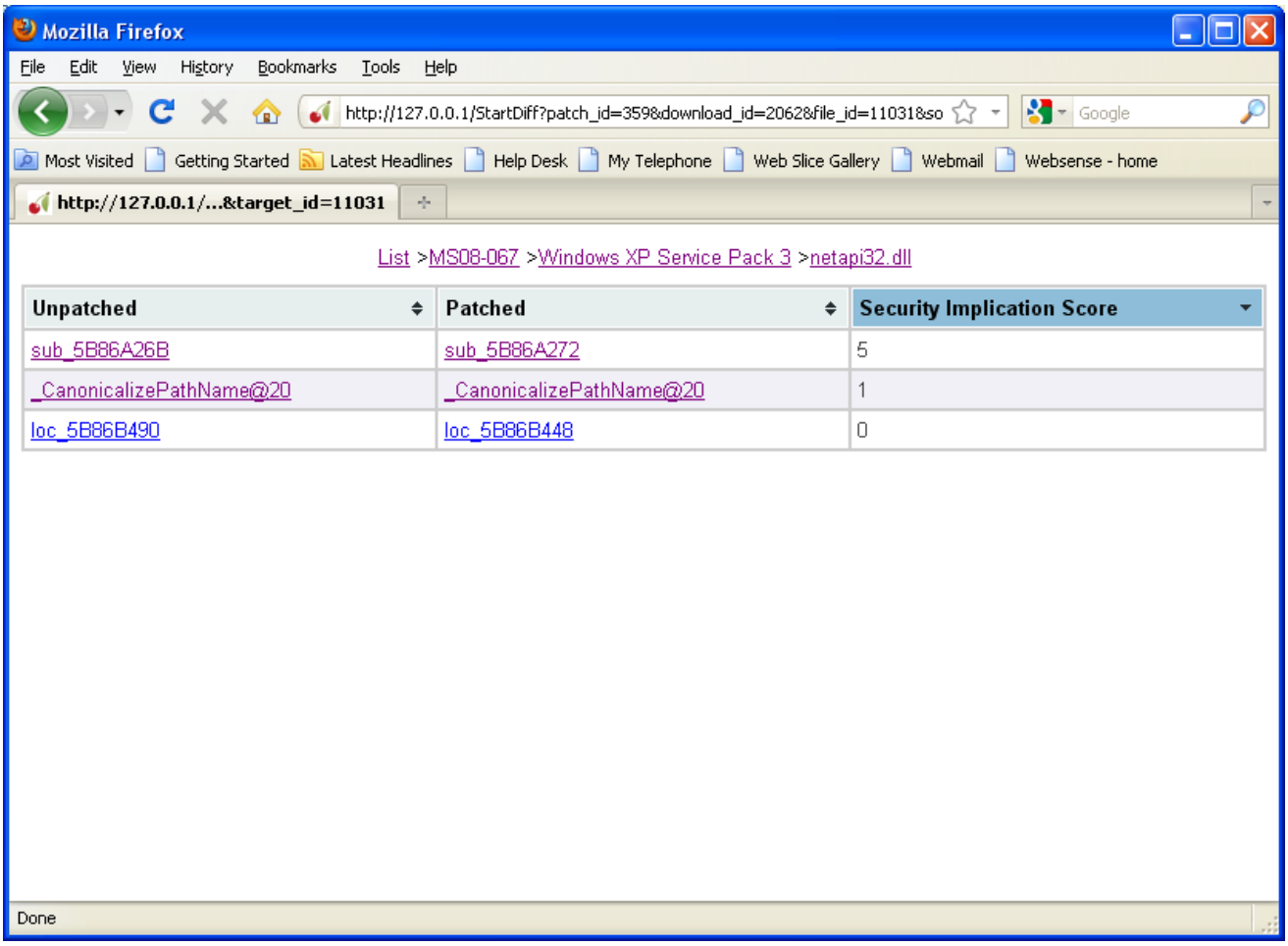
Picture 12: List of files to analyze

If you choose the filename you want to analyze, the screen like Picture 13 appears which showing unpatched filename automatically. The original file should have been indexed by the "Bin Collector" system for this automatic match to work. So before using Web Console system, constructing the binary library is crucial.



Picture 13: Diff page: The unpatched file is automagically guessed based on the file name and version string.

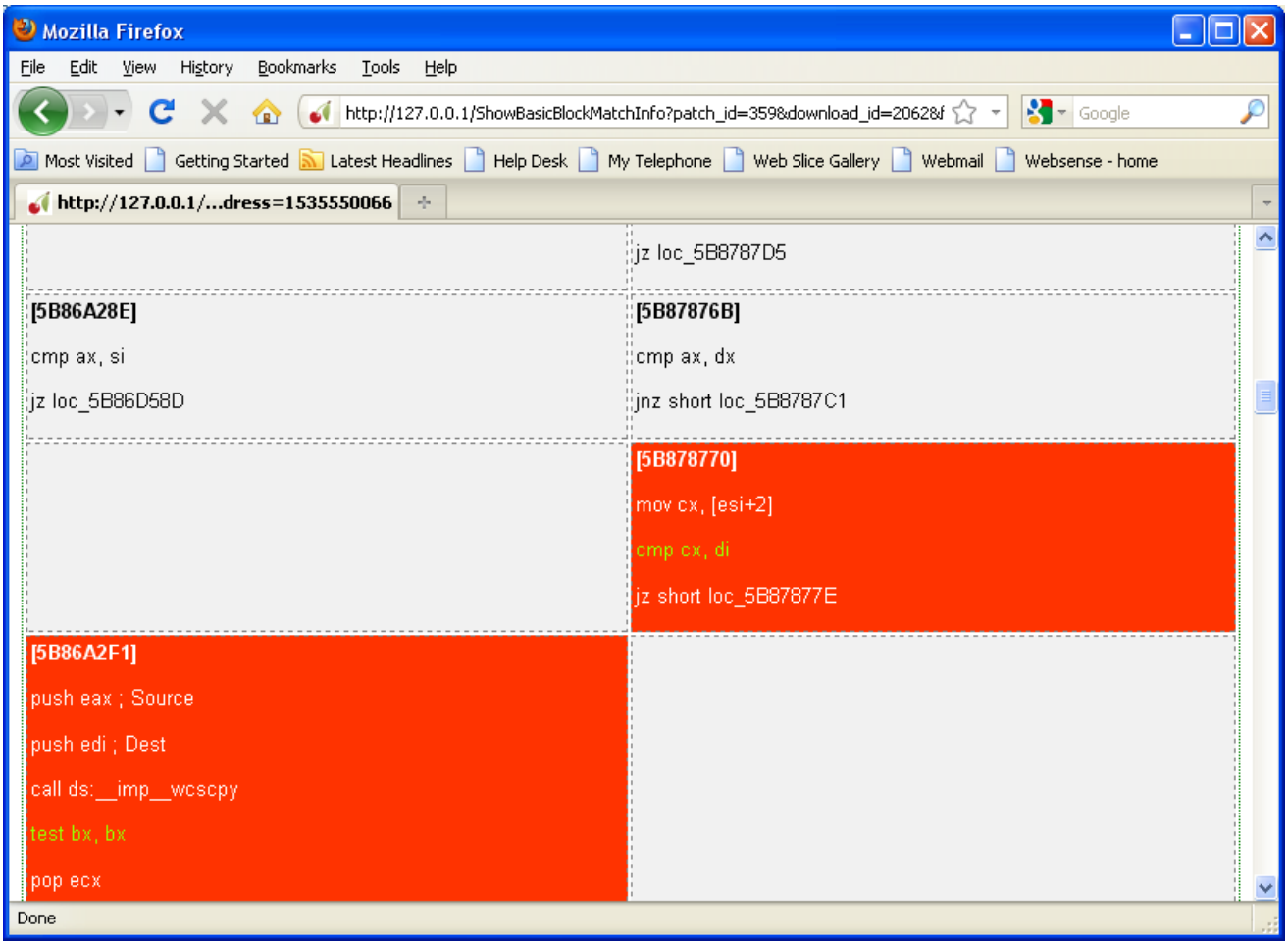
When you click "Start Diffing" button from this page, the actual diffing process starts. So it can take few minutes to long minutes for this process to finish. Sometimes the Web UI times out, but still the engine will be working in the background, you just need to check the Web Console's output log to check if the process has been finished.



Picture 14: The diff results: function list

If the whole process is finished you can see a screen like Picture 14. It shows the function names that has been modified by the patch and also shows the “Security Implication Score”. With this score, you can guess which function has more security implication in the patch. By clicking the column header, you can sort them out.

If you click one of the functions in the table, you will get a screen looks like Picture 15. It's a block representation of diffing results. And the color code is same as DarunGrim2's graph view. White blocks are same blocks and red block are removed or inserted blocks. Also yellow blocks are modified blocks.



Picture 15: Function level view of basic blocks

IDATracker

IDATracker is a tool working as a plugin for IDA that traces critical variables in the disassembly. It can help to trace variables you are interested in. Currently it works only manually when you designate any interesting variables. But future implementation might decide which variable to trace using some kind of heuristics. For example, an argument is passed to “malloc” API using push instruction, the heuristics engine can decide to trace that length variable and see if it's controllable from the user data. User data is something like coming from the network through recv API call or RPC. Or it can be the data that is coming from HTML pages or multimedia files. It's not easy to define the user data and critical variable, though.

Patterns Of Vulnerabilities and Patches

This is to show examples for each vulnerability classes. And you can grab an idea how you can extract useful patterns out of the patches. Both DarunGrim2 and DarunGrim3 examples are used, but they are basically same in concept. And also “Security Implication Score” are shown for some examples.

Buffer Overflow

Buffer overflow is a classic vulnerability that has been exploited over two decades now. The first worm used buffer overflow condition in Sendmail and it was more than 20 years ago. After that long time, the problem still exists in major products from the major vendors.

Examples

MS06-070

This is the vulnerability that is actually found by me in 2006. The function that is vulnerable is “_NetpManageIPCConnect@16” with security implication score 6.

List >MS06-070 >Microsoft Windows XP Service Pack 2 — >netapi32.dll

Unpatched	Patched	Security Implication Score
_NetpManageIPCConnect@16	_NetpManageIPCConnect@16	6
sub_5B88F5EB	sub_5B869B96	2

Picture 16: Patched functions(showing security implication score)

Actually the vulnerability comes from the ”swprintf” function used in insecure manner as shown in Picture 17. The third argument for the API call is coming from user controllable RPC argument. The attackers just need to send long hostname and the overflow happens and that's it.

[5B885192]	[5B8851C9]
push esi ; Format	push ebx ; Format
push offset aWslpc ; "%ws\\IPC$"	push offset aWslpc ; "%ws\\IPC$"
push eax ; String	push esi ; String
call ds:__imp__swprintf	call ds:__imp__swprintf
add esp, 0Ch	add esp, 0Ch
test [ebp+arg_C], 2	test [ebp+arg_C], 2
jz short loc_5B885217	jz short loc_5B885257

Picture 17: Vulnerable code located in MS06-070 patches

The thing you can notice here is that even though the “swprintf” call is vulnerable they didn't fix the part, but instead, they added length check at the start of the function where is before this code is called. You can see the patched code from Picture 18. It's using “wcslen” API to check the length of string before it's used by unsafe call to “swprintf” API.

Picture 18: Inserted wcslen call and cmp instruction

So you can see that "wcslen" can be used for pattern in recognizing buffer overflow vulnerabilities.

MS08-067

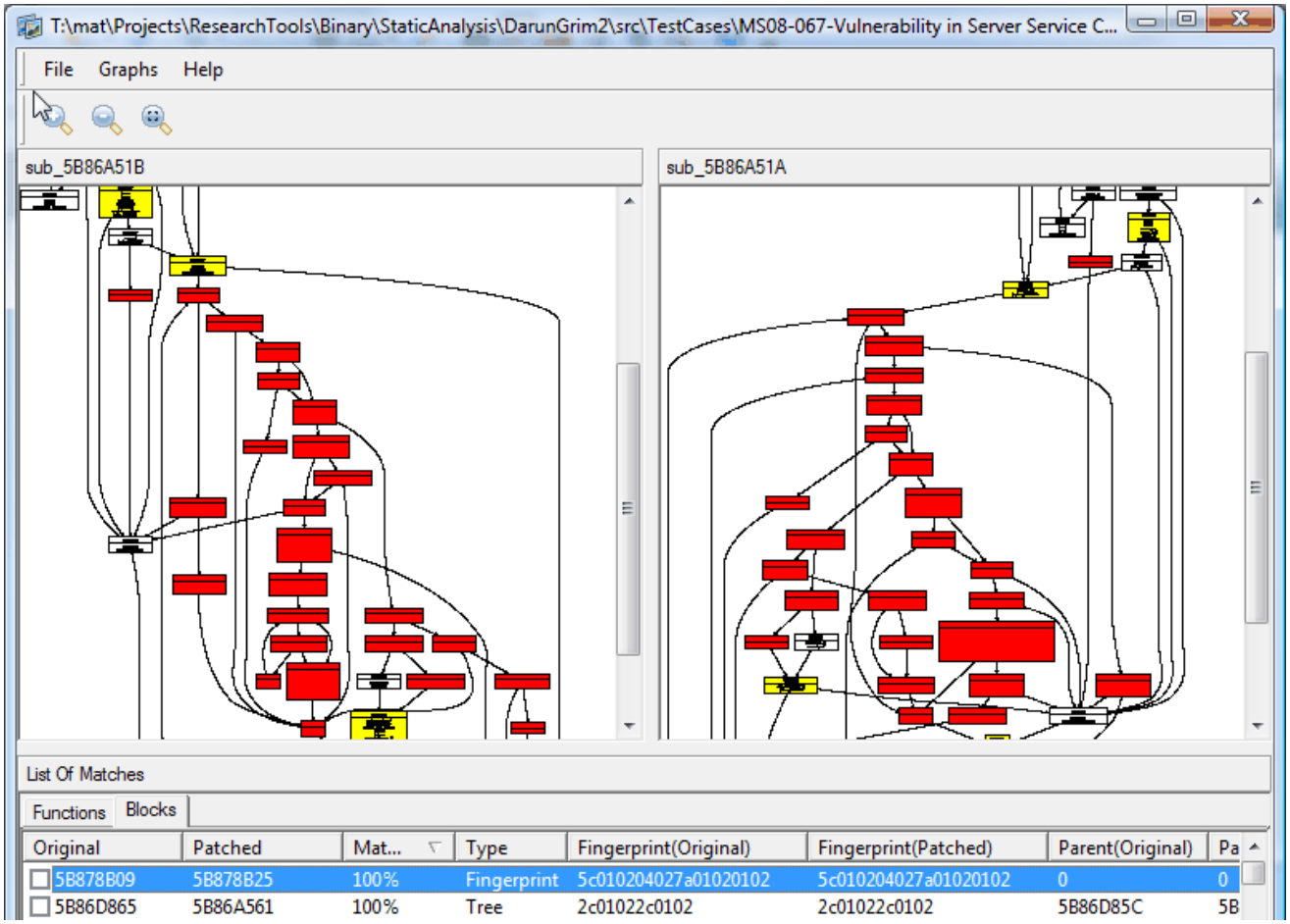
The next sample is about the vulnerability patched by MS08-067. Conficker worm exploited this vulnerability to propagate through internal network. Actually this patch is very easy target for binary diffing because only 2 functions changed. One is a change in the calling convention. The other is the function that has the vulnerability. Picture 19 shows 3 functions modified but the last one that starts with "loc_" is not valid function. You can also see from this picture that "sub_5B86A26B" and "sub_5B86A272" pair has most high security implication score of 20 compared to 1 of the next one. You can be very sure that this match has the security patches we want.

List >MS08-067 >Windows XP Service Pack 2 >netapi32.dll

Unpatched	Patched	Security Implication Score
sub_5B86A26B	sub_5B86A272	20
_CanonicalizePathName@20	_CanonicalizePathName@20	1
loc_5B86B490	loc_5B86B448	0

Picture 19: Modified functions in MS08-067 patch(showing security implication score)

Picture 20 shows DarunGrim2 graphs view of the sub_5B86A51B vs sub_5B86A272 pair. You might notice that a lot of red blocks in both sides which means many codes are removed and inserted. So looks like they rewrote whole code here.



Picture 20: sub_5B86A51B vs sub_5B86A272: the function with most security implication score

So because the programmer decided to rewrite basically whole part of the function, you might think that signaturing might not be effective in this case. Anyway, the original problem was mainly from the logic error, not a problem from single API call like “swprintf” as in previous example from MS06-070. But the truth is you can still use signaturing in this case. Let's see the following code blocks from the patch.

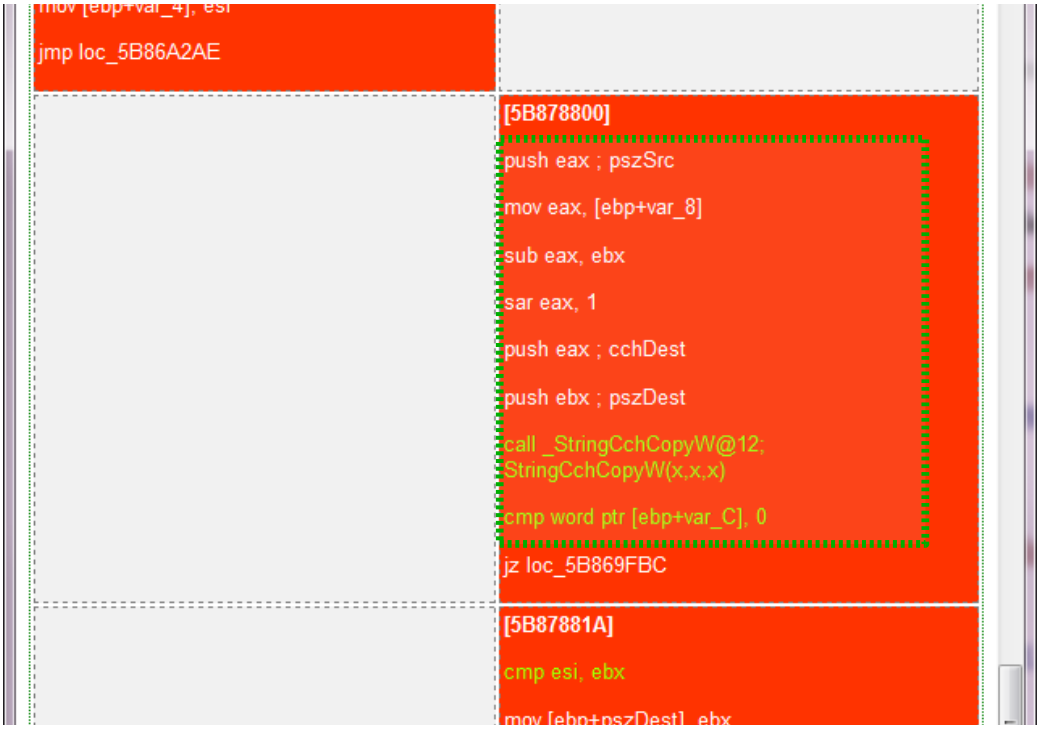
Picture 21: Inserted wcslen call and cmp instruction

The patch in Picture 21shows that “wcslen” is inserted to the patch. It's checking some input argument before the main logic starts. So “wcslen” pattern is still useful in this case. But that's not all. Let's see the example from Picture 23. The newly inserted block has a call to “StringCchCopyW” API which is a security improved version of strncpy. You can look up the API reference from StringCchCopyW.

Compared to the functions it replaces, **StringCchCopy** provides additional processing for proper buffer handling in your code. Poor buffer handling is implicated in many security issues that involve buffer overruns. **StringCchCopy** always null-terminates a non-zero-length destination buffer.

Behavior is undefined if the strings pointed to by *pszSrc* and *pszDest* overlap.

Picture 22: StringCchCopyW description from MSDN



Picture 23: Inserted new call to StringCchCopyW

So looks like we can use these safe string manipulation routines as our signatures for buffer overflow.

Signatures

Pattern matching for string length checking routines is a good sign for stack or heap based overflow.

There are variations of string length check routines.

```
strlen, wcslen, _mbslen, _mbstrlen
```

Pattern matching for safe string manipulation functions are good sign for buffer overflow patches.

Strsafe Functions are like following:

```
StringCbCat, StringCbCatEx, StringCbCatN, StringCbCatNEx, StringCbCopy, StringCbCopyEx, StringCbCopyN, StringCbCopyNEx,
StringCbGets, StringCbGetsEx, StringCbLength, StringCbPrintf, StringCbPrintfEx, StringCbVPrintf, StringCbVPrintfEx, StringCchCat,
StringCchCatEx, StringCchCatN, StringCchCatNEx, StringCchCopy, StringCchCopyEx, StringCchCopyN, StringCchCopyNEx,
StringCchGets, StringCchGetsEx, StringCchLength, StringCchPrintf, StringCchPrintfEx, StringCchVPrintf, StringCchVPrintfEx
```

Other Safe String Manipulation Functions are like following:

```
strcpy_s, wcscpy_s, _mbscpy_s
strcat_s, wcscat_s, _mbscat_s
strncat_s, _strncat_s_l, wcsncat_s, _wcsncat_s_l, _mbsncat_s, _mbsncat_s_l
strncpy_s, _strncpy_s_l, wcsncpy_s, _wcsncpy_s_l, _mbsncpy_s, _mbsncpy_s_l
sprintf_s, _sprintf_s_l, swprintf_s, _swprintf_s_l
```

Removal of unsafe string routines is also a good signature.

```
strcpy, wcscpy, _mbscpy
strcat, wcscat, _mbscat
sprintf, _sprintf_l, swprintf, _swprintf_l, __swprintf_l
vsprintf, _vsprintf_l, vswprintf, _vswprintf_l, __vswprintf_l
vsnprintf, _vsnprintf, _vsnprintf_l, _vsnwprintf, _vsnwprintf_l
```

Integer Overflow

Integer overflow is one of the vulnerabilities that can cause heap overflow by miscalculating the allocation size of the memory. If you supply huge number and if the vulnerable code is adding or multiplying some number to the original number to put some additional stuff that can be a problem. Because the memory allocated for the integer has fixed size, the number it can express has lower and upper limit. If the number is going over the limit it rolls over to the minimum number and becomes the minimum number it can express. And this cause security problem when buffer manipulation is involved with this wrongfully calculated length.

Examples

MS10-030

This vulnerability is from Microsoft Outlook mail client. The problem happens during processing responses from IMAP server. If the attacker runs some hostile IMAP server and convince the users to connect to it or by intercepting user's traffic and injecting this malicious IMAP traffic, he can attack the client by overflowing the buffer using integer overflow.

If you run the diffing, you can see the vulnerable functions as in Picture 24.

List >MS10-030 >Microsoft Outlook Express 6 >inetcomm.dll

Unpatched	Patched	Security Implication Score
?RootProps_EndChildren@CHTTPMailTransport@@QAEJXZ	?ContactInfo_EndChildren@CHTTPMailTransport@@QAEJXZ	5
_STR_ATT_COMBINED	_STR_ATT_RENDERED	4
?ResponseSTAT@CPOP3Transport@@AAEXXZ	?ResponseSTAT@CPOP3Transport@@AAEXXZ	4
?ResizeMsgSeqNumTable@CImap4Agent@@UAGJK@Z	?ResizeMsgSeqNumTable@CImap4Agent@@UAGJK@Z	4
_STR_ATT_NORMSUBJ	_STR_ATT_RENDERED	3
_STR_ATT_PRIORITY	_STR_ATT_RENDERED	3
?ResponseGenericList@CPOP3Transport@@AAEXXZ	?ResponseGenericList@CPOP3Transport@@AAEXXZ	3
?ProcessTransactTestResponse@CNNTPTransport@@AAEJXZ	?StartLogon@CNNTPTransport@@AAEXXZ	3
?GetMsgSeqNumToUIDArray@CImap4Agent@@UAGJPAPAKPAK@Z	?GetMsgSeqNumToUIDArray@CImap4Agent@@UAGJPAPAKPAK@Z	3
_STR_ATT_SERVER	_STR_ATT_FORMAT	2
??1CActiveMovie@@UAE@XZ	??1CBGImage@@UAE@XZ	2
?CheckForCompleteResponse@CImap4Agent@@AAEXPADKPAW4IMAP_RESPONSE_ID@@@Z	?CheckForCompleteResponse@CImap4Agent@@AAEXPADKPAW4IMAP_RESPONSE_ID@@@Z	2
_STR_ATT_STOREMSGID	_STR_ATT_RENDERED	1
_STR_ATT_FORWARDTO	_STR_ATT_FORMAT	1
?ExclusiveUnlock@CExShareLockWithNestAllowed@@QAEXXZ	?ExclusiveUnlock@CExShareLock@@QAEXXZ	1

Picture 24: Modified functions from MS10-030 Outlook Express 6 patch

Now look into ResponseSTAT method from CPOP3Transport class on the 3rd line of Picture 24. It has an additional call to "ULongLongToULong".

```
[7618DCF0]
mov ecx, ebx
shl ecx, 2
lea eax, [esi+584h]
push ecx ; unsigned __int32
lea edi, [esi+580h]
push eax ; void **
mov [edi], ebx
call ?HrAlloc@@YGJPAPAXK@Z; HrAlloc(void * *,ulong)
test eax, eax
mov [ebp+var_10], eax
jl short loc_7618DD36
```

```
[7618DE07]
lea eax, [ebp+var_C]
push eax ; unsigned __int32 *
push 4
pop ecx
mov eax, ebx
mul ecx
push edx
push eax ; unsigned __int64
mov [ebp+var_C], edi
mov [ebp+var_10], edi
call ?ULongLongToULong@@YGJ_KPAK@Z;
ULongLongToULong(unsigned __int64,ulong *)
cmp eax, edi
mov [ebp+var_14], eax
jl short loc_7618DE68
```

Picture 25: Additional call to ULongLongToULong

This routine is to convert 64 bit integer to 32bit integer. The original integer is coming from eax and edx register (64bit) which is calculated by multiplying 4 to the original integer. From line ".text:7618DE1F" it compares return value of the function with 0. The return value smaller than 0 means error during processing the number and it'll lead to immediate return of the function.

```
.text:7618DE07 lea     eax, [ebp+var_C]
.text:7618DE0A push    eax             ; unsigned __int32 *
.text:7618DE0B push    4
.text:7618DE0D pop     ecx
.text:7618DE0E mov     eax, ebx
.text:7618DE10 mul     ecx
.text:7618DE12 push    edx
.text:7618DE13 push    eax             ; unsigned __int64
.text:7618DE14 mov     [ebp+var_C], edi
.text:7618DE17 mov     [ebp+var_10], edi
.text:7618DE1A call    ?ULongLongToULong@@YGJ_KPAK@Z ; ULongLongToULong(unsigned __int64,ulong *)
.text:7618DE1F cmp     eax, edi
```

So if you look up the reference manual of the API "ULongLongToULong", you can see the following description. Basically the call to " ULongLongToULong" fails when the input is overflowing 32bit value range.

```
In the case where the conversion causes a truncation of the original value, the function returns
INTSAFE_E_ARITHMETIC_OVERFLOW and this parameter is not valid.
```

Intsafe.h from Microsoft's SDK show the error code. And it's like following.

```
#define INTSAFE_E_ARITHMETIC_OVERFLOW   ((HRESULT)0x80070216L)  // 0x216 = 534 =
ERROR_ARITHMETIC_OVERFLOW
```

And also the same file shows the function implementation.

```
//
// ULONGLONG -> LONG conversion
//
__inline
HRESULT
ULongLongToLong(
    __in ULONGLONG ullOperand,
    __out __deref_out_range(==,ullOperand) LONG* plResult)
{
    HRESULT hr;

    if (ullOperand <= LONG_MAX)
```

```
    {
        *plResult = (LONG)ullOperand;
        hr = S_OK;
    }
    else
    {
        *plResult = LONG_ERROR;
        hr = INTSAFE_E_ARITHMETIC_OVERFLOW;
    }

    return hr;
}
```

Basically it checks the argument against LONG_MAX and if it's too big, returns error status code with INTSAFE_E_ARITHMETIC_OVERFLOW.

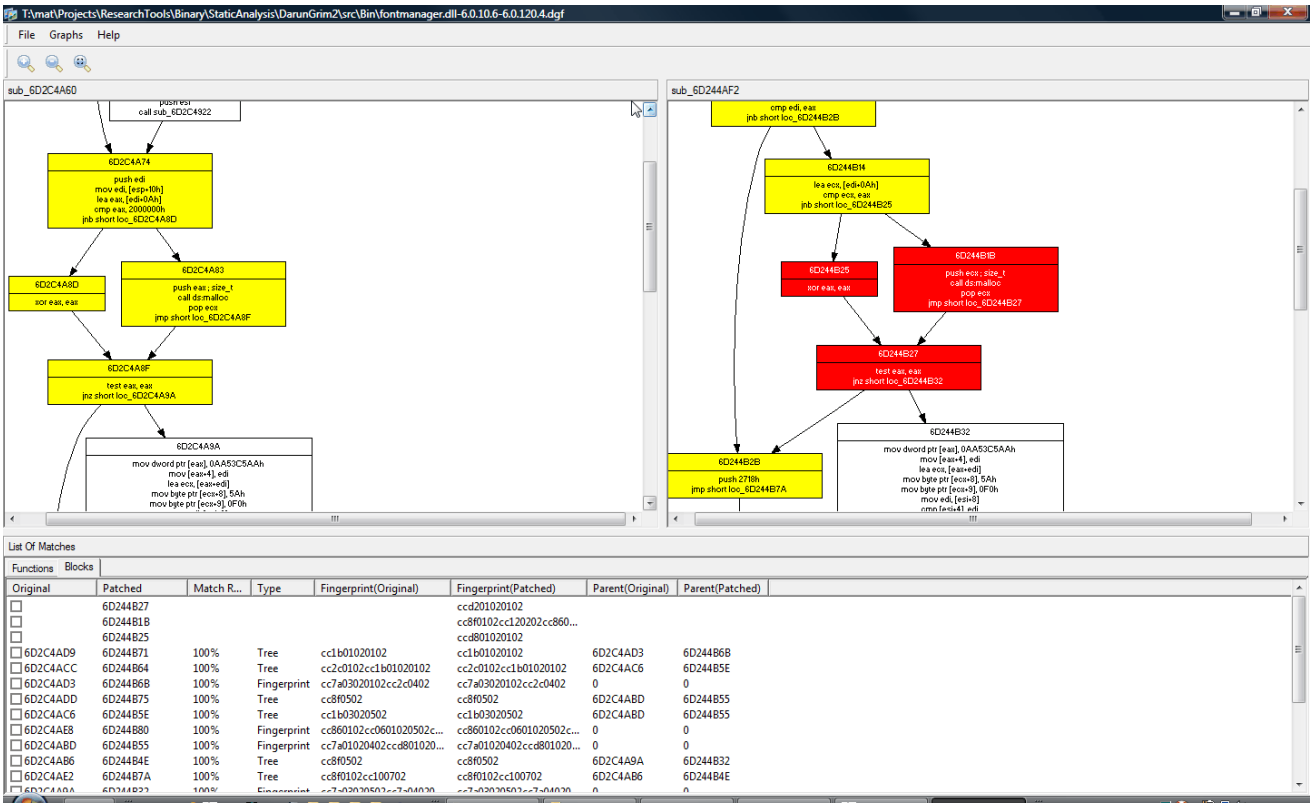
If you trace back where it's getting the original input for ULongLongToULong call, you can see that actually it's coming from the “StrToIntA” API and the argument for the API is “[ebp+lpSrc]” which is retrieve from the IMAP server supplied status line.

Picture 26: Call to StrToIntA to convert string to integer

So looks like Microsoft is fixing integer overflow issue using newly introduced safer number handling functions. You can read Michael Howard's blog on this issue from his blog post Safe Integer Arithmetic in C. So we can use these safe integer arithmetic functions to generate signature for integer overflow. One thing to note is that even though the blog says the functions are inlined, and also the intsafe.h include file also shows that the functions are inlined, for some reason this patch file doesn't have the function call inlined. Anyway even though it's inlined, we can create specific signature for that pattern using INTSAFE_E_ARITHMETIC_OVERFLOW constant value.

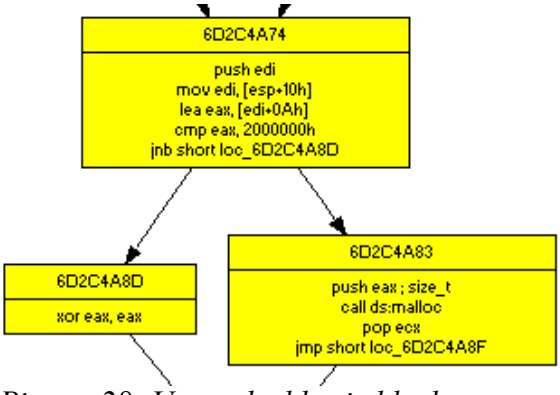
JRE Font Manager Buffer Overflow(Sun Alert 254571)

This example is from Oracle JRE font manager vulnerability. The overflow happens due to integer overflow. Picture 27 Shows the overview of the patched function.

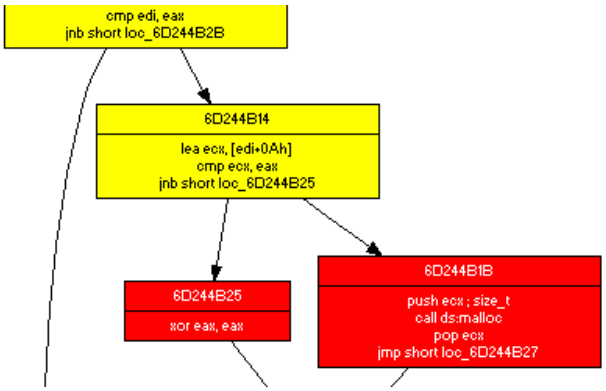


Picture 27: Patched basic blocks

If you zoom in, you can see the following basic blocks. What you can notice here is the "malloc" APIs in the both sides. You can just feel that this is about heap overflow and involving integer overflow.



Picture 28: Unpatched basic blocks



Picture 29: Patched basic blocks

If you look closer, unpatched codes only have one "cmp" instruction, but the patched on has one more additional "cmp" instruction. The red colored lines on the right side of Table 2 is to do additional sanity check of the length. Originally it was checking the result of "edi+0x0a" against 0x2000000, but with

new code it's also checking “edi” value itself against 0x2000000. So if the original “edi” value was bigger than 0x2000000 and also bigger enough to overflow back into smaller integer value than 0x2000000 when 0x0a is added, then it could have passed original sanity test leading to integer overflow by allocating too small value with big length for copying data.

Original		Patched	
.text:6D2C4A75	mov edi, [esp+10h]	.text:6D244B07	mov edi, [esp+10h]
		.text:6D244B0B	mov eax, 2000000h
		.text:6D244B10	**cmp edi, eax**
		.text:6D244B12	**jnb short loc_6D244B2B**
.text:6D2C4A79	lea eax, [edi+0Ah]	.text:6D244B14	lea ecx, [edi+0Ah]
.text:6D2C4A7C	cmp eax, 2000000h	.text:6D244B17	cmp ecx, eax
.text:6D2C4A81	jnb short loc_6D2C4A8D	.text:6D244B19	jnb short loc_6D244B25
.text:6D2C4A83	push eax ; size_t	.text:6D244B1B	push ecx ; size_t
.text:6D2C4A84	call ds:malloc	.text:6D244B1C	call ds:malloc

Table 2: Comparison between the basic blocks

In this case the fix was adding additional cmp instruction, and it's happening around malloc API. So you can use both additional “cmp” instruction and “malloc” API as the signatures for integer overflow.

Signatures

Safe integer conversion functions can be used to check sanity of an integer derived from string.

Here's the list of arithmetic functions that is doing safer operation.

```
UnsignedMultiply128, Int8ToUChar, Int8ToUInt8, Int8ToUShort, Int8ToUInt, Int8ToULong, Int8ToULongLong, UInt8ToInt8,
UInt8ToChar, ByteToInt8, ByteToChar, ShortToInt8, ShortToUChar, ShortToChar, ShortToUInt8, ShortToUShort, ShortToUInt,
ShortToULong, ShortToULongLong, UShortToInt8, UShortToUChar, UShortToChar, UShortToUInt8, UShortToShort, IntToInt8,
IntToUChar, IntToChar, IntToUInt8, IntToShort, IntToUShort, IntToUInt, IntToULong, IntToULongLong, UIntToInt8, UIntToUChar,
UIntToChar, UIntToUInt8, UIntToShort, UIntToUShort, UIntToInt, UIntToLong, LongToInt8, LongToUChar, LongToChar,
LongToUInt8, LongToShort, LongToUShort, LongToInt, LongToUInt, LongToULong, LongToULongLong, ULongToInt8,
ULongToUChar, ULongToChar, ULongToUInt8, ULongToShort, ULongToUShort, ULongToInt, ULongToUInt, ULongToLong,
LongLongToInt8, LongLongToUChar, LongLongToChar, LongLongToUInt8, LongLongToShort, LongLongToUShort, LongLongToInt,
LongLongToUInt, LongLongToLong, LongLongToULong, LongLongToULongLong, ULongLongToInt8, ULongLongToUChar,
ULongLongToChar, ULongLongToUInt8, ULongLongToShort, ULongLongToUShort, ULongLongToInt, ULongLongToUInt,
ULongLongToLong, ULongLongToULong, ULongLongToLongLong, UInt8Add, UShortAdd, UIntAdd, ULongAdd, SizeTAdd,
SIZETAdd, ULongLongAdd, UInt8Sub, UShortSub, UIntSub, ULongSub, SizeTSub, SIZETSub, ULongLongSub, UInt8Mult,
UShortMult, UIntMult, ULongMult, SizeTMult, SIZETMult, UlongLongMult
```

You can also reference this page from MSDN for more information.

In case safe version of the string comparison routine is inlined, you can use constant value of INTSAFE_E_ARITHMETIC_OVERFLOW, which is 0x80070216L, as a signature for safer integer conversion routines.

Also you can check the existence of atoi, _atoi_l, _wtoi, _wtoi_l or StrToInt Function functions on both sides of files.

Additional cmp x86 operation is a good sign of integer overflow check. It will perform additional range check for the integer before and after of the arithmetic operation. Counting additional number of "cmp" instruction in patched function might help deciding integer overflow.

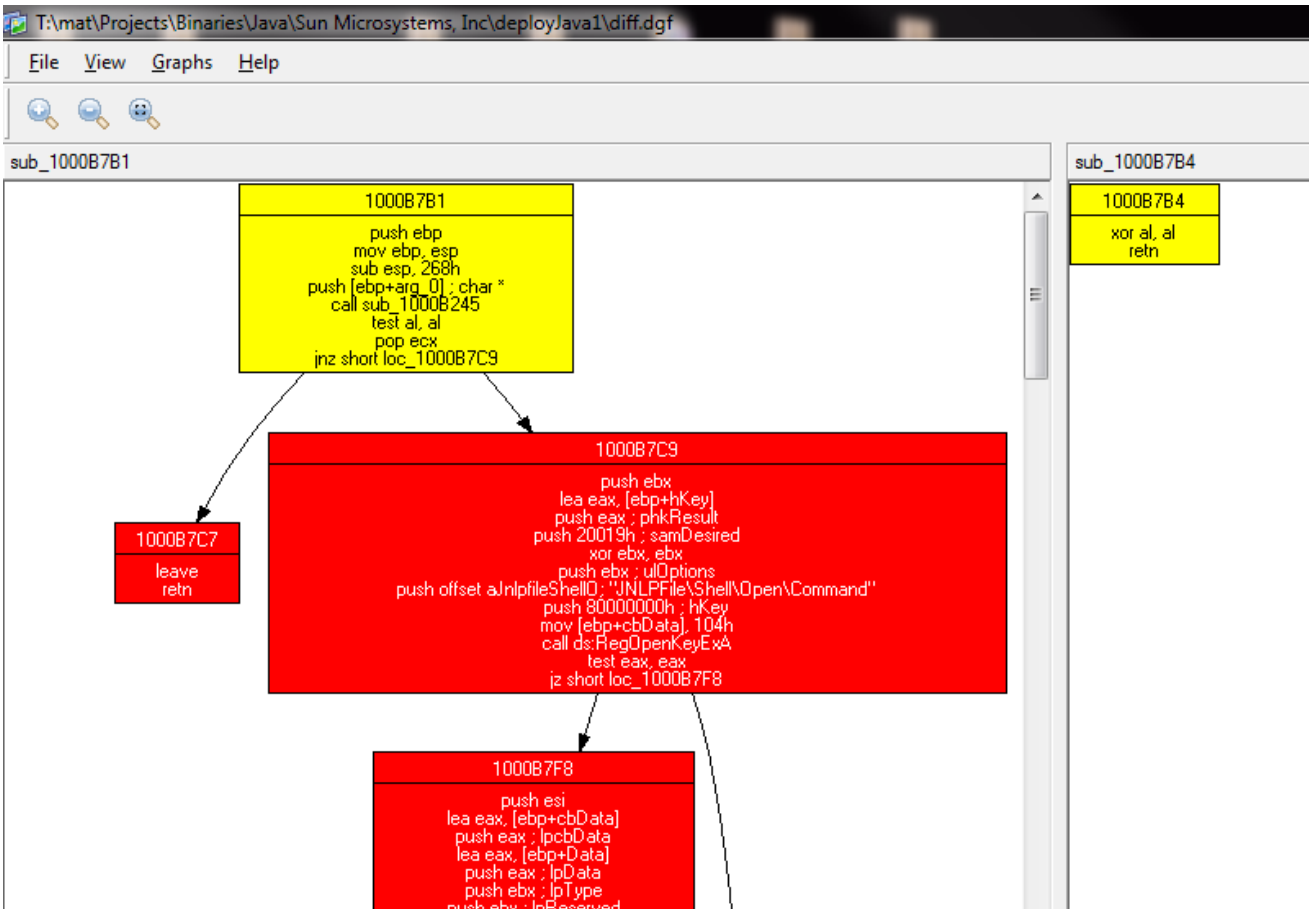
Invalid Validation of Parameters

There are also some bugs that is not related to memory corruption. One of them is invalid validation of parameters. It can happen for process creation or also for calling some privileged APIs. I'll show you two examples of parameter validation failures and their patch methods. There is no easy way to detect these patches in general way, but still it's possible to create signatures case by case.

Examples

Insufficient Validation of Parameters Java Deployment Toolkit

The first example is the vulnerability found by Tavis Ormandy. The issue is passing dangerous parameters to “javaw.exe” command line through web pages. If you run the vulnerable dll through DarunGrim, you can get some result like Picture 30.



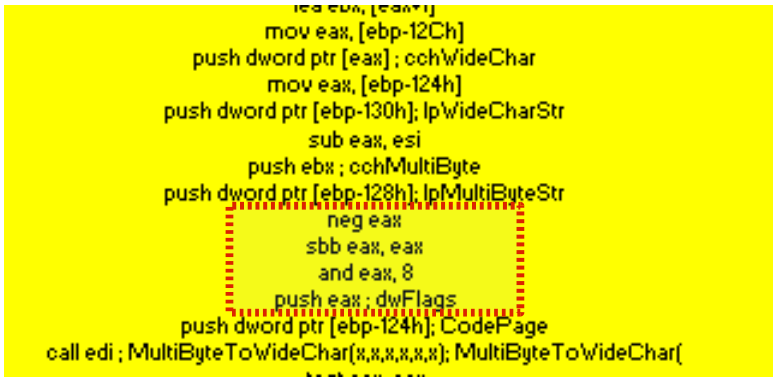
Picture 30: Patch for "Insufficient Validation of Parameters Java Deployment Toolkit" vulnerability

You can see unpatched one has whole a lot of red and yellow blocks, but in the patched function, the whole function's basic blocks have been removed. With further investigation, the function is

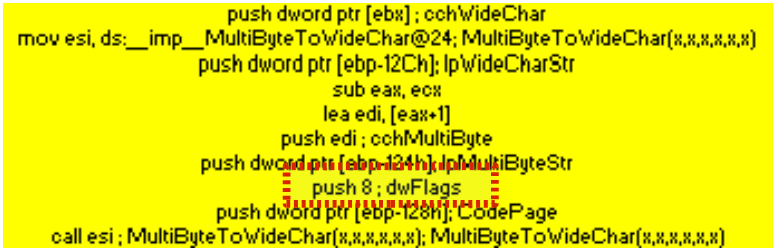
responsible for querying registry key for JNLPFile Shell Open key and launching it using CreateProcessA API.

MS09-020:WebDav ACL failure

This vulnerability was found by Thierry Zoller. The detail is in his blog IIS 6 / IIS 5 / IIS 5.1+ Webdav auth bypass (update #7). The details from Microsoft SRD team is at More information about the IIS authentication bypass.



Picture 31: Original call to MultiByteToWideChar



Picture 32: Patched call to MultiByteToWideChar

MultiByteToWideChar Function explains meaning of dwFlags value 8 like following:

MB_ERR_INVALID_CHARS

Windows Vista and later: The function does not drop illegal code points if the application does not set this flag.

Windows 2000 Service Pack 4, Windows XP: Fail if an invalid input character is encountered. If this flag is not set, the function silently drops illegal code points. A call to GetLastError returns ERROR_NO_UNICODE_TRANSLATION.

The problem is that unpatched logic was using the return value from call to “FIsUTF8Url(char const *)” function, and if it detects UTF-like string inside the string using some heuristics, it will set the dwFlags to 0, otherwise 8. So if it thinks that the string is in UTF-8 format, it will not set MB_ERR_INVALID_CHARS(8) flag even though there are some invalid UTF-8 characters. And this logic failure leads to ACL check failure.

Signatures

If validation of parameters are related to process creation routine, we can check if the original or patched function has a process creation related APIs like CreateProcess function in modified functions.

The WebDav issue is related to string conversion routine like MultiByteToWideChar Function, we can check if the modified or inserted, removed blocks have these kinds of APIs used in it. If the pattern is found in modified blocks, it's a strong sign of UTF-8 conversion issue

.

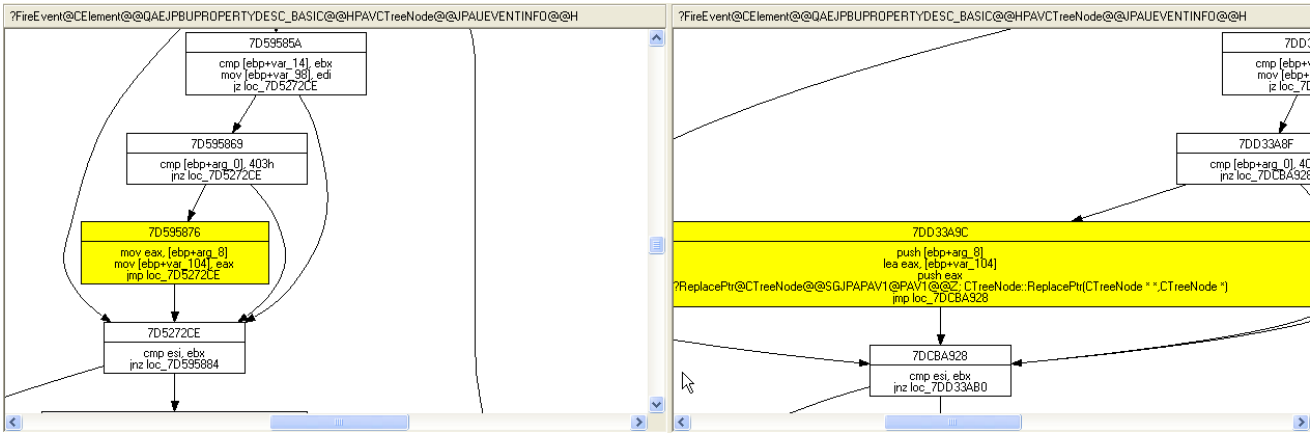
Use-After-Free

Use-After-Free bug is a new kind of bug class that is very popular these days. It's usually used together with heap-spraying. If you don't have an ability to create excessive amount memory to fill in the freed memory, the success rate of the attack using this vulnerability will drop drastically. So that's why usually this bugs are found in Internet Explorer.

Examples

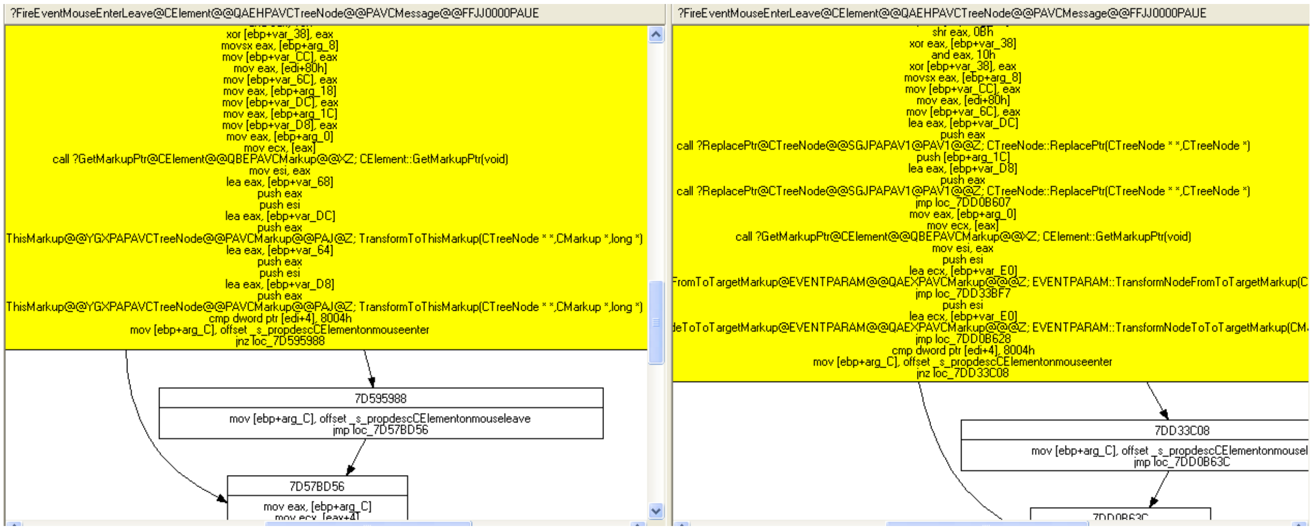
CVE-2010-0249-Vulnerability in Internet Explorer Could Allow Remote Code Execution

This vulnerability is from Internet Explorer. And this was exploited in the wild before it's reported to Microsoft. The issue happens when event is fired for some object that is already released in the memory. The binary diffing result will look like following Picture 33.



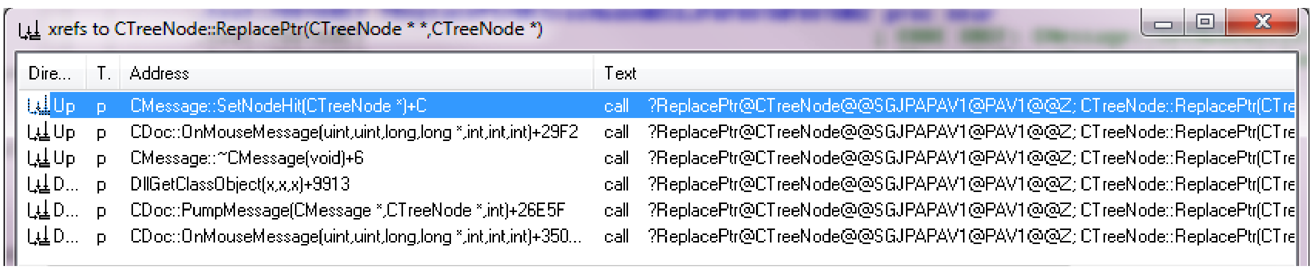
Picture 33: An addition of CTreeNode::ReplacePtr method

What you can notice from this diffing result is that “CTreeNode::ReplacePtr” is used in patched version of the function. If you look into other modified functions related to event generation, you can see that it also involves a lot of “CTreeNode::ReplacePtr” calls.



Picture 34: Additions of CTreeNode::ReplacePtr methods

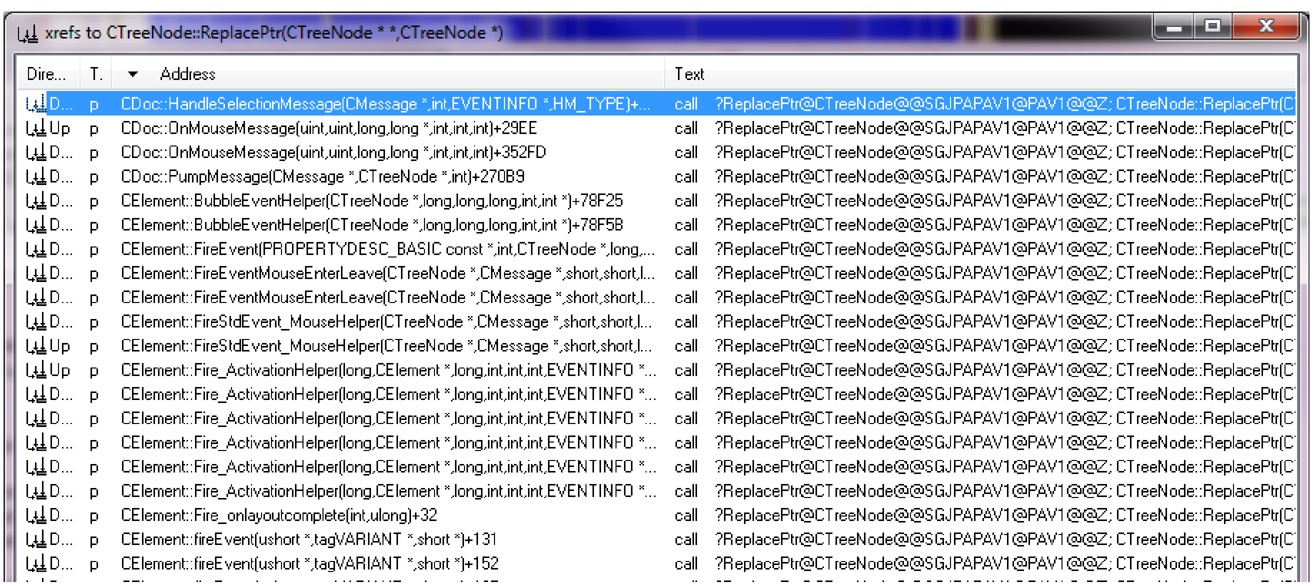
You can confirm this using IDA by checking the call reference to “CTreeNode::ReplacePtr” method.


xrefs to CTreeNode::ReplacePtr(CTreeNode * *,CTreeNode *)

Dire...	T.	Address	Text
Up	p	CMessage::SetNodeHit(CTreeNode *)+C	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(CTre
Up	p	CDoc::OnMouseMessage(uint,uint,long,long *,int,int,int)+29F2	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(CTre
Up	p	CMessage::~CMessage(void)+6	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(CTre
D...	p	DllGetClassObject(x,x,x)+9913	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(CTre
D...	p	CDoc::PumpMessage(CMessage *,CTreeNode *,int)+26E5F	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(CTre
D...	p	CDoc::OnMouseMessage(uint,uint,long,long *,int,int,int)+350...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(CTre

Picture 35: Unpatched file's call reference to CTreeNode::ReplacePtr

The unpatched function has only 6 references to “CTreeNode::ReplacePtr” method.

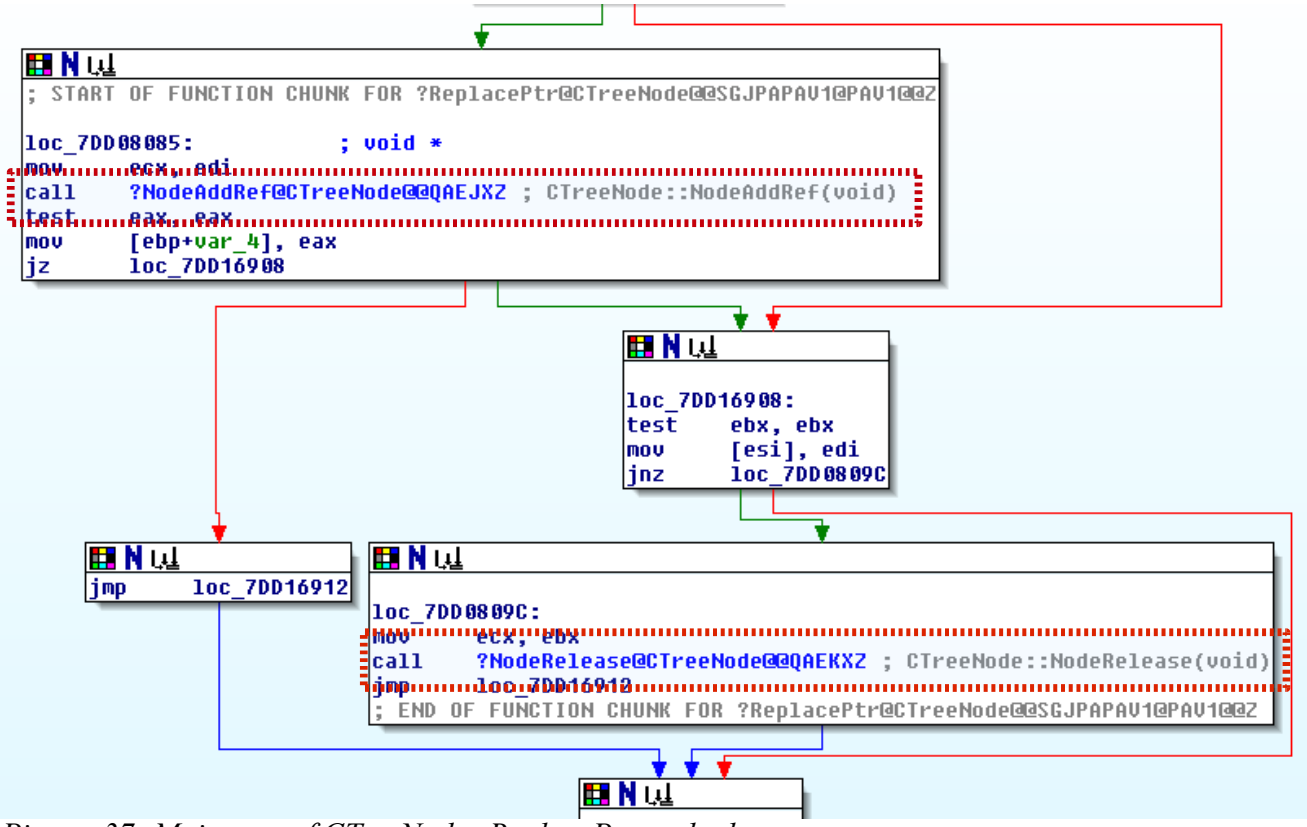

xrefs to CTreeNode::ReplacePtr(CTreeNode * *,CTreeNode *)

Dire...	T.	Address	Text
D...	p	CDoc::HandleSelectionMessage(CMessage *,int,EVENTINFO *,HM_TYPE)+...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
Up	p	CDoc::OnMouseMessage(uint,uint,long,long *,int,int,int)+29EE	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CDoc::OnMouseMessage(uint,uint,long,long *,int,int,int)+352FD	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CDoc::PumpMessage(CMessage *,CTreeNode *,int)+270B9	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::BubbleEventHelper(CTreeNode *,long,long,long,int,int *)+78F25	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::BubbleEventHelper(CTreeNode *,long,long,long,int,int *)+78F5B	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::FireEvent(PROPERTYDESC_BASIC const *,int,CTreeNode *,long,...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::FireEventMouseEnterLeave(CTreeNode *,CMessage *,short,short,l...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::FireEventMouseEnterLeave(CTreeNode *,CMessage *,short,short,l...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::FireStdEvent_MouseHelper(CTreeNode *,CMessage *,short,short,l...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
Up	p	CElement::FireStdEvent_MouseHelper(CTreeNode *,CMessage *,short,short,l...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
Up	p	CElement::Fire_ActivationHelper(long,CElement *,long,int,int,int,EVENTINFO *...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::Fire_ActivationHelper(long,CElement *,long,int,int,int,EVENTINFO *...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::Fire_ActivationHelper(long,CElement *,long,int,int,int,EVENTINFO *...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::Fire_ActivationHelper(long,CElement *,long,int,int,int,EVENTINFO *...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::Fire_ActivationHelper(long,CElement *,long,int,int,int,EVENTINFO *...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::Fire_ActivationHelper(long,CElement *,long,int,int,int,EVENTINFO *...	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::Fire_onlayoutcomplete(int,ulong)+32	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::fireEvent(ushort *,tagVARIANT *,short *)+131	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C
D...	p	CElement::fireEvent(ushort *,tagVARIANT *,short *)+152	call ?ReplacePtr@CTreeNode@@SGJPAPAV1@PAV1@@Z; CTreeNode::ReplacePtr(C

Picture 36: Patched file's call reference to CTreeNode::ReplacePtr

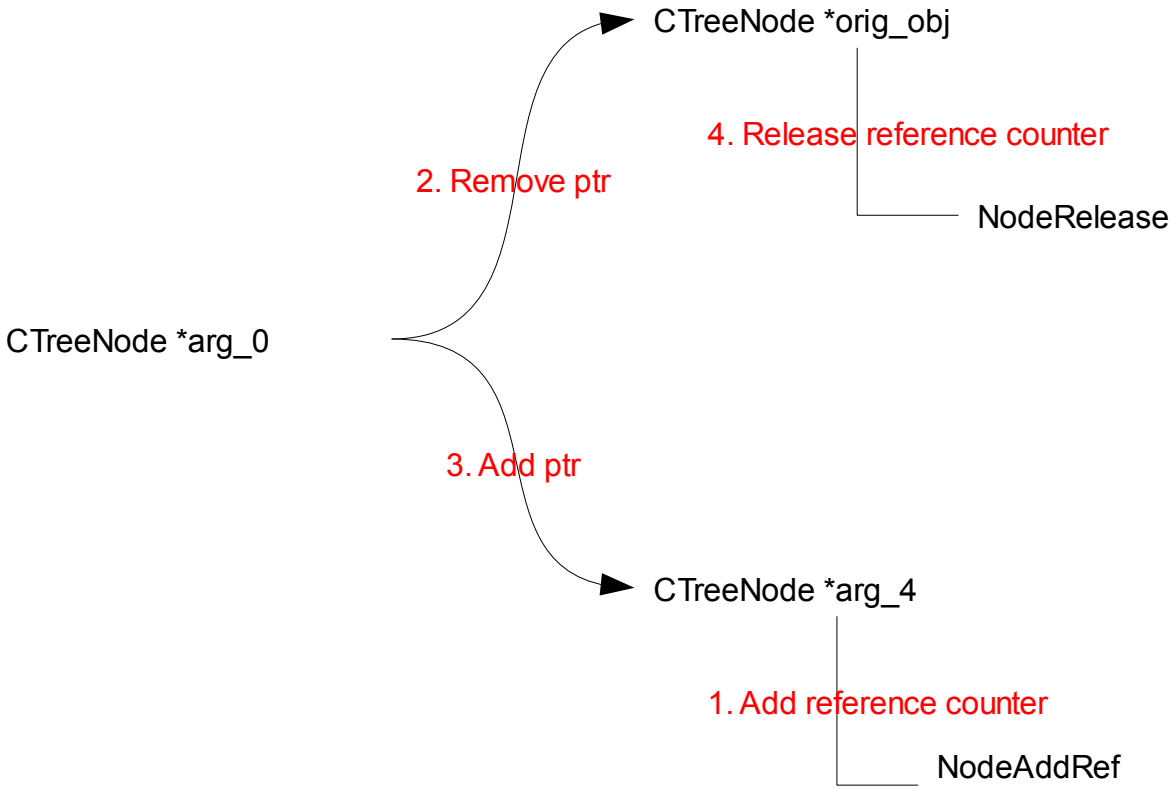
While the patched function has the call references all over the places as seen in Picture 36. And the function name implies that they are all related to event processing.

If you look into the “CTreeNode::ReplacePtr” method. It looks like Picture 37. It's calling “CTreeNode::NodeAddRef” and “CTreeNode::NodeRelease” methods.



Picture 37: Main part of CTreeNode::ReplacePtr method

So with further analysis, we can see that the function releases reference to original memory pointer and add new reference to the new CTreeNode object that has been passed as first argument to the method.



Picture 38: Flow of logic in CTreeNode::ReplacePtr

So conclusion here is original binary was failing to replace pointer for the tree node. And freed node was used accidentally later in the code. “ReplacePtr” methods in adequate places fixed the problem. We might use “ReplacePtr” pattern for use-after-free bug in IE. But the effectiveness is fairly limited to this same type of bug in Internet Explorer. Anyway, adding the pattern will help to find same issue later binary diffing if they missed any function replace this time.

CVE-2010-0806: Use-after-free vulnerability in the Peer Objects component

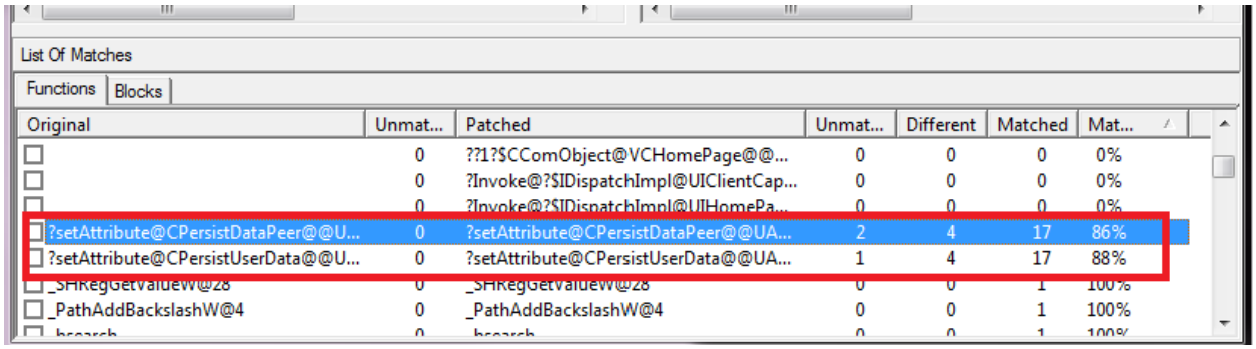
In the wild exploit looked like this.

```
function blkjbdkjb()
{
 eejeefe();
 var sdfsfsdf = document.createElement("BODY");
 sdfsfsdf.addBehavior("#default#userData");
 document.appendChild(sdfsfsdf);
 try {
  for (i=0; i<10; i++)
  {
   sdfsfsdf.setAttribute('s',window);
  }
```

```
}catch(e)
{
}
```

This is about setting invalid object as an attribute to “#default#userData” object. The object might be expecting string variant type of object and will convert the original object to BSTR variant type leaving some members of the structure pointing to invalid memory.

Here's the diffing results using DarunGrim.

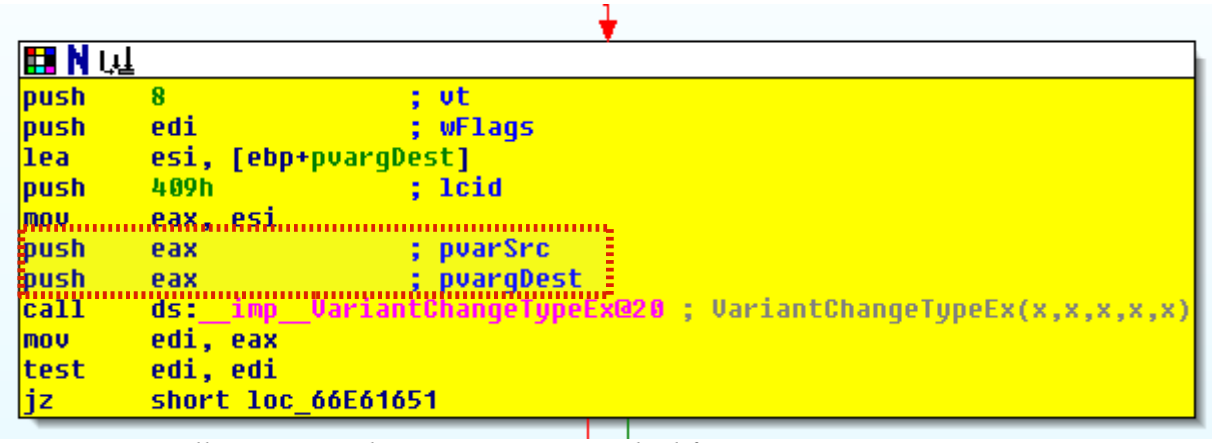


Picture 39: Patched functions from CVE-2010-0806

If you look into “CPersistUserData::setAttribute” function for function level diffing results, you can find interesting change in the call to “VariantChangeTypeEx” function.

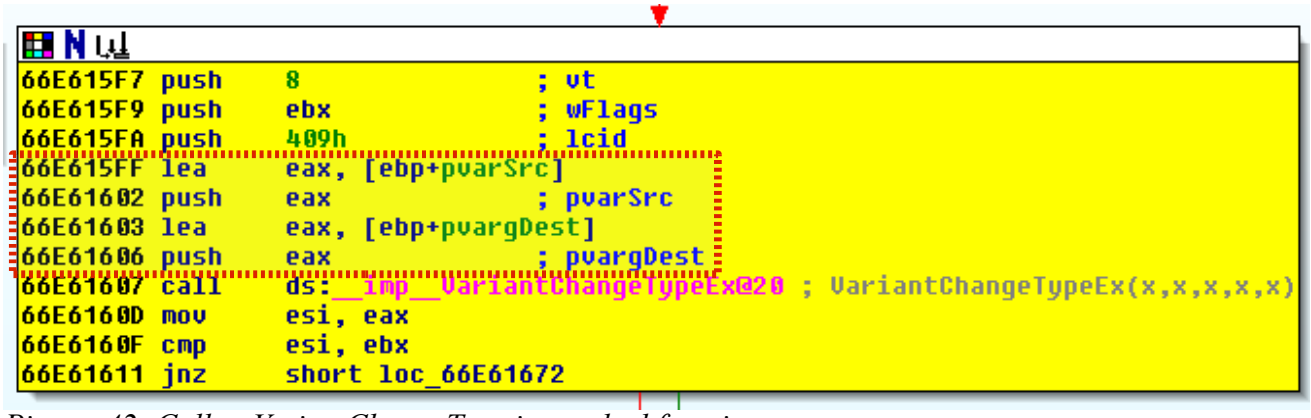
Picture 40: Diffing result

In unpatched function the 1st and 2nd arguments to function “VariantChangeTypeEx” are same as shown in Picture 41



Picture 41: Call to VariantChangeType in unpatched function

In patched function the 1st and 2nd argument to function VariantChangeType is different as shown in Picture 42.



Picture 42: Call to VariantChangeType in patched function

Looks like this vulnerability happens during converting Variant type to VT_BSTR when the type is not VT_BSTR type. During the time it converts the data passed as the 3rd argument for the method. The source and destination of the function were same and it could be a problem for the original Variant data because the function will overwrite original object and the object will be used in other parts of the code .

The fix is separating the source and destination of the call to preserve source argument data. So we can use the "VariantChangeTypeEx" calls appearing in modified blocks as a signature for identifying these kinds of vulnerabilities involving Variant type variable conversion issues.

Finding 1-day Exploit

If you look at the diffing results, you can see that actually two methods are modified. One was "CPersistUserData::setAttribute" that is related to "#default#userData", the other is "CPersistDataPeer:: setAttribute" which we don't have any idea. If you look into function level patching, "CPersistDataPeer:: setAttribute" has same type of modification to " VariantChangeTypeEx" function call as in "CPersistUserData::setAttribute". So we can see that the issue is not limited to

"#default#userData" persistent data. If you trace “CPersistDataPeer:: setAttribute” usage, you can easily guess that the method is used by “savehistory” and “savesnapshot” behavior. So the problem is not limited to "#default#userData", but also “#default#savehistory” and “#default#savesnapshot” behaviors are affected.

With this new information you can write a simple 1-day exploit like following. This is just a POC and it will only crash IE that has not been patched for CVE-2010-0806, but if you put heap-spraying code to here that will make it a valid and full functional 1-day exploit that can perform remote code execution.

```
<HTML>
<HEAD>
<META NAME="save" CONTENT="history">
<STYLE>
  .sHistory {behavior:url(#default#savehistory);}
</STYLE>

<SCRIPT>
function test()
{
        i1.setAttribute( "x", document );
        i2.setAttribute( "x", document );
        i3.setAttribute( "x", document );
        i4.setAttribute( "x", document );
}
</SCRIPT>
</HEAD>
<BODY>
<INPUT class=sHistory type=text id=i1 onload="test()">
<INPUT class=sHistory type=text id=i2 onload="test()">
<INPUT class=sHistory type=text id=i3 onload="test()">
<INPUT class=sHistory type=text id=i4 onload="test()">
</BODY>
</HTML>
```

No available documents on the Internet ever mentioned this issue. If you use binary diffing technology, it's a matter of time before you can find something interesting like 1-day exploits that hasn't been disclosed in any other places.

The problem with 1-day exploit is that many security appliances and software products are still heavily dependent on signatures and they are using simple pattern matching as their main defense measure. If an attacker figured out 1-day exploits and using them for their own benefits, then the original signature will not cover the issue. So IPS vendors need to pay more attention to patch analysis and need to get most out of the vendor patches to write their signatures. Even though the vendors are releasing more

and more information to the security vendors, looks like it's still more of the security vendor's work to figure out how to protect users from the undisclosed vulnerabilities.

Signatures

We can make custom signatures incurred from each incidents of use-after-free bugs. Also if it involves Variant related routines, we can use Variant data type handling APIs as the signatures. These kinds of vulnerabilities are also different case by case, so further research is need to establish a stable way to catch them.

Conclusion

Binary diffing can benefit IPS rule writers and security researchers. Locating security vulnerabilities from binary can help further binary auditing.

There are typical patterns in patches according to their bug class. Security implication score by DarunGrim3 helps finding security patches out from feature updates. The security implication score logic is written in Python and customizable on-demand. Also we presented typical patterns for each major vulnerability classes. The signature system needs for us to update the signatures as new patterns are found. And data flow analysis using IDATracker will help the researchers to expedite the analysis process.

This automatic security patch locating ability will be beneficial to the IPS rule writers. They can spend more time in concentrating on what really matters instead of spending time to find the actual parts to analyze. To achieve all these, I upgraded the current implementation of DarunGrim(http://www.darungrim.org) binary diffing system to support pattern matching. And also I'm going to release IDATracker as an open-source project soon.