Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries

Jeongwook Oh(mat@monkey.org,oh.jeongwook@gmail.com)

Jeongwook Oh works on eEye's flagship product called "Blink". He develops traffic analysis module that filters attacker's traffic. The analysis engine identifies protocol integrity violations by protocol parsing and lowers the chances of false positives and false negatives compared to traditional signature based IPS engines.

He's also interested in blocking ActiveX related attacks and made some special schemes to block ActiveX-based attacks without any false positives. The implementation was integrated to the company's product and used by the customers.

He runs Korean security mailing list called Bugtruck(not bugtraq).

Introduction

The Problem

Security patches are usually meant to fix security vulnerabilities. And thost are for fixing problems and protect computers and end users from risks. But how about releasing patch imposes new threats? We call the threat "1-day exploits". Just few minutes after the release of patches, binary diffing technique can be used to identify the vulnerabilities that the security patches are remedying. It's ironical situation but that's what is happening these days.
This binary diffing technique is especially useful for Microsoft's binaries. Not like other vendors they are releasing patch regularly and the patched vulnerabilities are relatively concentrated in small areas in the binary. Those facts make the patched area in the binary more visible and apparent to the patch analyzers.
We already developed "eEye Binary Diffing Suites" back in 2006 and it's widely used by security researchers to identify vulnerabilities. It's free and open-source and it's powerful enough to be used for 1-day vulnerabilities hunting purpose. So virtually, attackers have access to all the tools and targets they need to identify unknown vulnerabilities that is just patched. They can launch attack during the time frame users or corporates are applying patches. This process typically takes few minutes to few days.
From our observations during past few years, all the important security patches were binary diffed manually or automatically using tools. Sometimes the researchers claimed they finished analyzing patches in just 20-30 minutes. At most in a day, it's possible to identify the vulnerability itself and make working exploits. And binary diffing has now become too easy and cheap to the attackers. During patch applying time frame, the end users are more vulnerable and targeted using 1-day attack.

The Answer

So now it became crucial to make theses 1-day exploits more difficult and time-consuming so that the vendors can earn more time for the consumers to apply patches. Even though using severe code obfuscation is not an option for Microsoft's products, they can still follow some strategies and techniques to defeat the binary diffing processes without forsaking stability and usability. We are going to show the methods and tactics to make binary differs life harder. And will show the in-house tool that obfuscates the binaries in a way that especially binary differs confused. We call this process anti-binary diffing.

Binary Diffing

The History

Since being introduced 10 years ago by Bmat paper[BMAT], binary diffing is now so common and easily affordable technique. Aside from expensive commercial tools like "bindiff", there are already 2-3 free or opensource tools that can be used to identify exact patched points in the patch files.
Here we are presenting the brief history of binary diffing theories and tools.

BMAT(1999)
It severely depends on symbolic name matching. It was mainly used for Microsoft's binaries which symbol they have access to. With name-based matching, after matching procedure or function is identified, it conducts hashing-based comparison for the blocks inside each procedure. It's using 64bit hash value calculated from the instructions and the checksum is order dependent. Hashing is generated from instruction bytes. There are multiple level of abstractions with opcode and operands.
This paper is mainly focused on transfer of profile propagation and not in security patch analysis. The strength of this paper is that it suggested using checksum for block matches. Sometimes same block can be changed without code change by code optimizations. But usually the vendors doesn't change the optimization level and the registers and order of instructions in basic blocks usually remains same. So just doing checksumming basic blocks and compare them instead of whole instruction text will be acceptable in normal times. Also they showed 5 different levels of checksumming method for the basic blocks. They call these levels "matching fuzziness levels". As the level decreasess more and more information the hash has will use for calculation. The level 0 basically has all the information of the basic block including register allocation and block address operand and operands and opcodes. Level 5 has only opcode information.
This paper also presented CFG based basic block matches. So this match method is actually the basic method for binary diffing.

Automated Reverse Engineering(2004)
Halvar presented at Blackhat 2004 on reverse engineering techniques[ARE]. And he suggested using fingerprints of functions for binary diffing. Basically same idea is used for his following papers and bindiff tool. The idea is unique and simple. Using number of nodes and edges and number of calls as traits, construct a function signature. And compare them between binaries. It suggests isomorphic comparison between functions CG(call graph)s.

Comparing binaries with graph isomorphism(2004)

Todd Sabin suggested isomorphic analysis and matching two binaries[TODD]. It's based on instructions graph's isomorphic matching. It adopted divide and conquer strategy to diff whole structure. The interesting point is that it compares instructions not basic blocks. But the instruction comparison is merged as basic block matching and function matching. He claimed the binary diffing performance is acceptable and usable. But no POC ever released.

Structural Comparison of Executable Objects(2004)

Improved version of Halvar's Blackhat 2004 "Automated Reverse Engineering(2004)"[ARE] presentation[SCEO].

Graph-based comparison of Executable Objects(2005)

Halvar improved previous paper "Structural Comparison of Executable Objects(2004)". The basic idea is same.

The problem is that the algorithm is heavily dependent on CFG generation from the binaries. As the paper is saying in modern day binaries it's not simple work to extract CFG and CG. The complete CFG analysis of binary itself is a challenging work and any inconsistency in CFG can result in major analysis failure of whole binaries.

You can ignore CFG recognition failure, but it means a lot of false negatives. And there are some chances you miss important parts. Also this algorithm can't catch patches not incurring any structural changes like by using constant values or register changes.

The Tools

We are going to present all the major binary diffing tools out until now(commercial or freeware). We also compare their algorithms and show how effective in real world.

Sabre Security's bindiff(2004)

Halvar made a commercial binary diffing tool based on his graph based fingerprinting theory.

IDACompare(2005)

Based on the description of the tool, it looks like it's based on signature scanning. We didn't test the tool but it's mainly targeted for porting malware analysis data porting. It's also using mdb and the page says it's designed for around 500k file in size. For normal patch diffing the binary file can be more than 1M Bytes to 10 to 20 Mbytes.

eEye Binary Diffing Suite(2006)

We at eEeye, developed a opensource binary diffing suite back in 2006 summer. It was used internally for Microsoft's Patch Tuesday patches analysis. At that times, Microsoft didn't provide patch information to the vendors and patch analysis was the only way to obtain some secret information they don't release.

The "DarunGrim" is one of the tools included and performs the main binary diffing analysis. The tool used sqlite database to save differential analysis results. The main algorithm used in this tool is basically used in DarunGrim2, also. Because it was written in Python, it had low performance when the analysis target is bigger than around 10 mega bytes. And this is solved by using C++ instead of Python in DarunGrim2.

Patchdiff2(2008)

From the description of the tool, this tool is made specifically for security patch or hotfix analysis. This is not suitable tool for similar functions finding or malware similarities research. This is a freely distributed tool from Tenable Security. From the document, looks like they are using checksum of graph call for signaturing, which mean they have similar scheme as Halvar's.

DarunGrim2

This is the improved version of eEye Binary Diffing Suite. The major difference is that it's using C++ instead of Python. Aside from that it introduced a lot of improvements over the past version of DarunGrim.

DarunGrim2

Algorithms

We are going to present all known binary diffing algorithms, tactics that can be practically used. We will explain the algorithms that are used by diffing tools one by one with experimental data we collected. Some statistical method will be used to show the effectiveness of each algorithms. Mere simple looking algorithms can be surprisingly powerful in identifying serious vulnerabilities.
The engine is based on fingerprint hashing method to match procedures fast and in reliable way. The previous works in binary difference analysis were mainly concentrated on the graph structure analysis and graph isomorphism. But they involve intensive comparison of two graphs and also has a drawback like depending on the disassembler's CFG analysis capabilities. If the CFG is not complete whole procedure will remain unmatched. Some pattern like indirect call can't be analyzed using isomorphic analysis. Name matching is also widely used for procedure matching but if the symbol is not available, this is not an option.

Fingerprint hashing is the way to overcome this limitation and to improve analysis result drastically. It also has benefit of high performance. We are presenting the way to implement the fingerprint hashing and will show some real binary differences analysis examples.

But fingerprint hashing is not the only algorithms used in DarunGrim2, there are also some algorithms to match functions based on fingerprint matching. DarunGrim2 is also using traditional isomorphic algorithms and name matching.

Symbolic Names Matching

DarunGrim2 is using symbolic name matching as like other binary diffing tools. It's a basic starting points for binary matching procedure. The symbolic name can be exported name or names retrieved from vendor provided symbol . Usually the symbol file is not provided to the public, but Microsoft is kind enough to provide symbol files to anyone as soon as the patch is out. So it helps a lot with binary diffing process and actual tracing of the vulnerabilities.

Fingerprint Hash Map

What is a fingerprint?

We are presenting fingerprint hashing method as main algorithm of DarunGrim2. This method is very simple. Also it can be implemented without any intensive analysis of binaries. Fingerprint hashing is using a way to abstract the instruction sequences.

In general, fingerprint can have multiple meanings in computer science. In this article fingerprint is used to express a data bytes representing a basic block. For every basic blocks DarunGrim2 extract code bytes features and use that as a key to hash table. We call that fingerprint of the block. The size can be variable and relative to sum of the actual instructions bytes size. There can be many ways to make fingerprint.

Fingerprint matching is a method for matching basic blocks efficiently. Basic block is the basic element for differential analysis. It's a basic block that has one or more code referencing blocks. Building hash table takes O(n). DarunGrim2 build two fingerprint hash table for original binary and patched binary. For each unique fingerprints from original binary fingerprints, DarunGrim2 check if the patched binaries fingerprint hash table has matching entry. This procedure takes O(n) because searching hash table takes O(1).

Unlike flirt-like traditional function level fingerprint DarunGrim2 is using abstracted forms of fingerprint bytes as a hash data for basic blocks. So even though some of the basic blocks in a function is modified, other matching basic blocks support the function matching. For faster matching of enormous number of fingerprint hashes generated, we put all unique fingerprint strings to hash table and use them for matching fingerprints. There can be many ways to generate the fingerprint for a basic block. And as you switch the fingerprint generation method, it shows different behavior in the matching process.

Generating fingerprint for a basic block

The simplest way to generate fingerprint of a basic block is using opcodes and operand. Because fingerprint generation of the basic blocks are most crucial part in this method, there are few things to consider in generating fingerprints depending on the compiler and link options. We can determine to ignore memory or immediate type operands, because they tend to change with source code modification. We might choose to ignore any register differences and make them as one representation. There can be multiple way of fingerprint generation methods.

Using IDA

The implementation is using IDA for binary analysis. So the fingerprint generation is dependent on IDA's disassembly representation. And IDA is providing good structure to extract this information. DarunGrim2 plugin is using insn_t structure to retrieve information on instructions.

Overcoming Order Dependency

The drawback of simple sequential extraction of fingerprint is that it's order dependent. Actually this problem is common for any algorithms dependent on instructions order. So to solve this problem we also have option to perform instruction ordering normalization before generating any signatures. This is done by choosing instructions that is dependent each other. And sort them by instruction bytes ascending order. Any dependent instructions sequence should maintain their order anyway. This instruction ordering normalization is especially important for the binaries built with compilers doing aggressive optimization using instruction reordering.

Reducing Hash Collision

There can be short basic blocks and they tend to have duplicates in one binary. That is to say, there can be fingerprints that is not unique in the binary. This makes hash collision when used in hash table. Basically we ignore and don't use the entries that has conflicts for comparison. But there are better way to make the collision rate low. By associating one blocks fingerprint with next connected blocks in fingerprint CFG, we have some chances of making the collision blocks unique. In that way we can use fingerprint hashing for multiple linked blocks and reduce the number of collisions.

Determining functions matching pair

After hash table traverse and matching blocks, we go through every procedures in original binary to calculate the number of matches to the procedures in patched binary. If one procedure has basic blocks matching to different procedures in patched binary, we need to select one of them. We use simple method to accomplish this. We just count the number of matching entries for each matching couples and we select the match with highest number. If there are multiple entries with same highest number, we need to select one of them randomly.

So, the matching functions are determined by comparing the matching counts for each functions and selecting the function match pair that has biggest matching count. If the matching function is determined, any non-compliant matching pairs which has matching basic block in non-matching functions will be revoked.

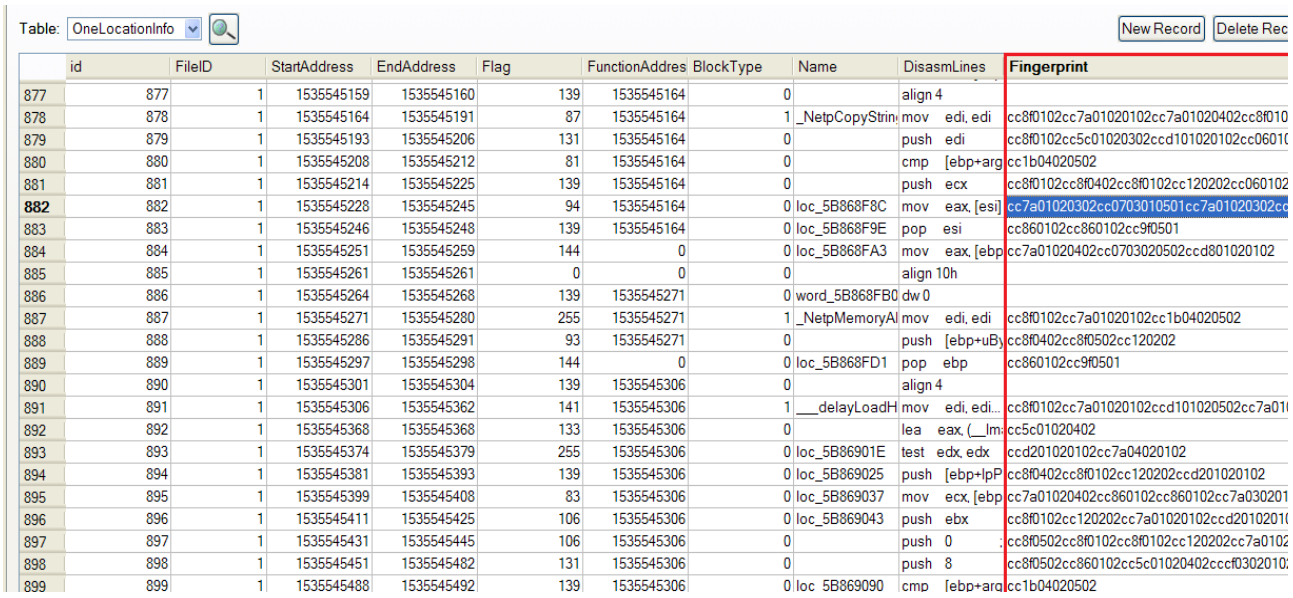
So the determining process for matching functions is mainly dependent on fingerprint matching. And for the real world patches, we found that fingerprinting basic blocks showed good performance and quality.

Matching blocks inside function

If matching functions are determined, the blocks inside each functions will have some kind of a locality. So we again construct fingerprint hash table for basic blocks inside each function and can perform fingerprint match on that. This is called "fingerprint match inside function". In this case the node number is relatively small. The collision probability is also very low compared to the global fingerprint match performed before.

How does it look like?

Here's the example of fingerprint representation inside database.



	id	FileID	StartAddress	EndAddress	Flag	FunctionAddres	BlockType	Name	DisasmLines	Fingerprint
877	877	1	1535545159	1535545160	139	1535545164	0		align 4	
878	878	1	1535545164	1535545191	87	1535545164	1	_NetpCopyStrin	mov edi, edi	cc8f0102cc7a01020102cc7a01020402cc8f010
879	879	1	1535545193	1535545206	131	1535545164	0		push edi	cc8f0102cc5c01020302ccd101020102cc0601
880	880	1	1535545208	1535545212	81	1535545164	0		cmp [ebp+arg	cc1b04020502
881	881	1	1535545214	1535545225	139	1535545164	0		push ecx	cc8f0102cc8f0402cc8f0102cc120202cc060102
882	882	1	1535545228	1535545245	94	1535545164	0	loc_5B868F8C	mov eax, [esi]	cc7a01020302cc0703010501cc7a01020302cc
883	883	1	1535545246	1535545248	139	1535545164	0	loc_5B868F9E	pop esi	cc860102cc860102cc9f0501
884	884	1	1535545251	1535545259	144	0	0	loc_5B868FA3	mov eax, [ebp	cc7a01020402cc0703020502ccd801020102
885	885	1	1535545261	1535545261	0	0	0		align 10h	
886	886	1	1535545264	1535545268	139	1535545271	0	word_5B868FB0	dw 0	
887	887	1	1535545271	1535545280	255	1535545271	1	_NetpMemoryAl	mov edi, edi	cc8f0102cc7a01020102cc1b04020502
888	888	1	1535545286	1535545291	93	1535545271	0		push [ebp+uBy	cc8f0402cc8f0502cc120202
889	889	1	1535545297	1535545298	144	0	0	loc_5B868FD1	pop ebp	cc860102cc9f0501
890	890	1	1535545301	1535545304	139	1535545306	0		align 4	
891	891	1	1535545306	1535545362	141	1535545306	1	___delayLoadH	mov edi, edi...	cc8f0102cc7a01020102ccd101020502cc7a01
892	892	1	1535545368	1535545368	133	1535545306	0		lea eax, (__Im	cc5c01020402
893	893	1	1535545374	1535545379	255	1535545306	0	loc_5B86901E	test edx, edx	ccd201020102cc7a04020102
894	894	1	1535545381	1535545393	139	1535545306	0	loc_5B869025	push [ebp+lpP	cc8f0402cc8f0102cc120202ccd201020102
895	895	1	1535545399	1535545408	83	1535545306	0	loc_5B869037	mov ecx, [ebp	cc7a01020402cc860102cc860102cc7a030201
896	896	1	1535545411	1535545425	106	1535545306	0	loc_5B869043	push ebx	cc8f0102cc120202cc7a01020102ccd2010201
897	897	1	1535545431	1535545445	106	1535545306	0		push 0 ;	cc8f0502cc8f0102cc8f0102cc120202cc7a0102
898	898	1	1535545451	1535545482	131	1535545306	0		push 8	cc8f0502cc860102cc5c01020402cccf0302010
899	899	1	1535545488	1535545492	139	1535545306	0	loc_5B869090	cmp [ebp+arg	cc1b04020502

Illustration 1: Query Results from Sqlite database showing Fingerprint entries.

Structure Based Analysis

Structure based analysis: basic block isomorphism

After determining matching functions and finishing fingerprinting match inside function, we perform structure based analysis. But, it's definitely not Halvar's method. Our method is just procedural and has a philosophy of divide and conquer. Because we are using basic blocks as graph node, it's different from Todd's isomorphic algorithm. As far as I know, this matching algorithm is similar to that is presented in BMAT tool [BMAT]. From the known matched nodes, it will try to match their children's nodes.

Control flow Inversion

In this process, we need to consider control flow inversion. Control flow inversion issue is already known from Todd's document [TODD]. DarunGrim2 just tweaks the CFG when we encounter any negative comparison type jumps(jnz,jle, etc).

Here's some C style pseudo code showing the code inversion removal process.

```
if(InstructionType==ja || InstructionType==jae || InstructionType==jc || InstructionType==jcxz || InstructionType==jecxz ||
InstructionType==jrcxz || InstructionType==je || InstructionType==jg || InstructionType==jge || InstructionType==jo ||
InstructionType==jp || InstructionType==jpe || InstructionType==js || InstructionType==jz || InstructionType==jmp ||
InstructionType==jmpfi || InstructionType==jmpni || InstructionType==jmpshort || InstructionType==jpo || InstructionType==jl ||
InstructionType==jle || InstructionType==jb || InstructionType==jbe || InstructionType==jna || InstructionType==jnae ||
InstructionType==jnb || InstructionType==jnbe || InstructionType==jnc || InstructionType==jne || InstructionType==jng ||
InstructionType==jnge || InstructionType==jnl || InstructionType==jnle || InstructionType==jno || InstructionType==jnp ||
InstructionType==jns || InstructionType==jnz
)
{
        if(InstructionType==ja || InstructionType==jae || InstructionType==jc || InstructionType==jcxz || InstructionType==jecxz ||
InstructionType==jrcxz || InstructionType==je || InstructionType==jg || InstructionType==jge || InstructionType==jo ||
InstructionType==jp || InstructionType==jpe || InstructionType==js || InstructionType==jz || InstructionType==jmp ||
InstructionType==jmpfi || InstructionType==jmpni || InstructionType==jmpshort)
        {
                is_positive_jmp=TRUE;
        }else{
                is_positive_jmp=FALSE;
        }
}
```

Calculating Match Rate

When performing procedural structural matching, we need to determine if two basic blocks are same or similar. We use fingerprint as a means for performing this operation. Because fingerprint is extracted from real code instructions, it shows the real implementation of the basic block and also has been abstracted and normalized for analysis. The fingerprint is stored as binary byte sequences in the memory when the fingerprint matching is performed, but it can be easily converted to hex ascii form representation. With converted form of ascii string we use well known string matching algorithms to calculate similarities of the basic blocks. This part is a little bit time-consuming and will have more false match rate if we used real instruction bytes or disassembly representation of the basic block.

So after all this process is finished we will repeat whole matching process again with unmatched code blocks that is not a member of any matching procedures. We repeat this process until there is no matching entries produced by any methods.

In real world examples, especially circumstances like Microsoft patch analysis, fingerprint matching and procedural structural matching is enough for identifying the patched part of the binary.

Real Life Issues

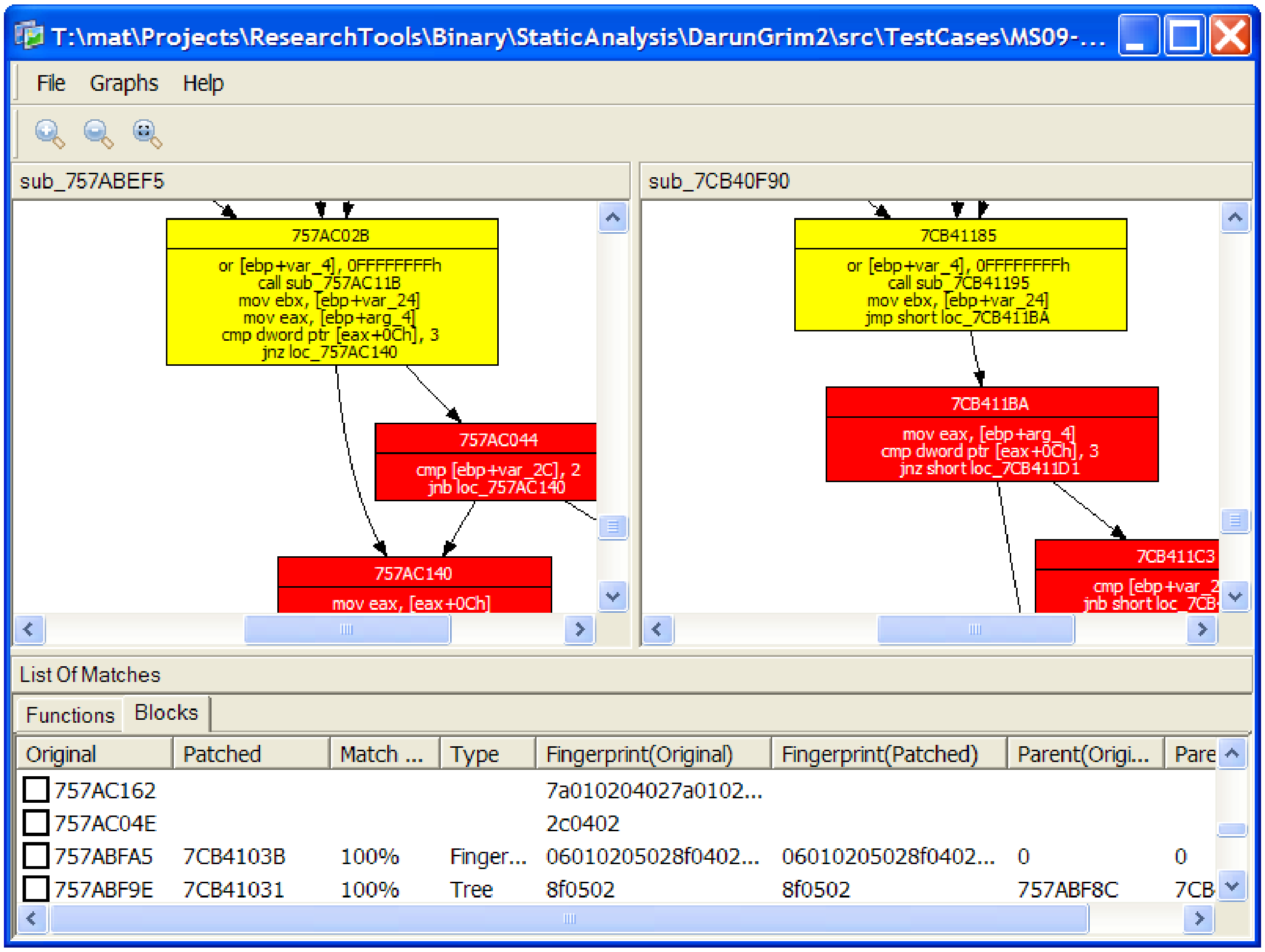
There are some issues with diffing real binaries because eash vendors are using different compilers and optimization methods. We are going to show some issues we encountered during binary diffing actual binaries.

Split Blocks

As a part of optimization, there are some blocks that is split in multiple location. We can define split block as like following.

"The block who has one child and the child of the block has only one parent in CFG."

The split blocks tend to make CFG broken and the matching process incomplete. Here's a example of the situation. 757AC02B block in left pane and 7CB411B5+7CB411BA blocks in right pane is same. But the blocks in right pane is split and all the children blocks is colored in red which means they are not recognized as matching blocks, even though they are matching.



T:\mat\Projects\ResearchTools\Binary\StaticAnalysis\DarunGrim2\src\TestCases\MS09-...
File
Graphs
Help
sub_757ABEF5
sub_7CB40F90
List Of Matches
Functions
Blocks

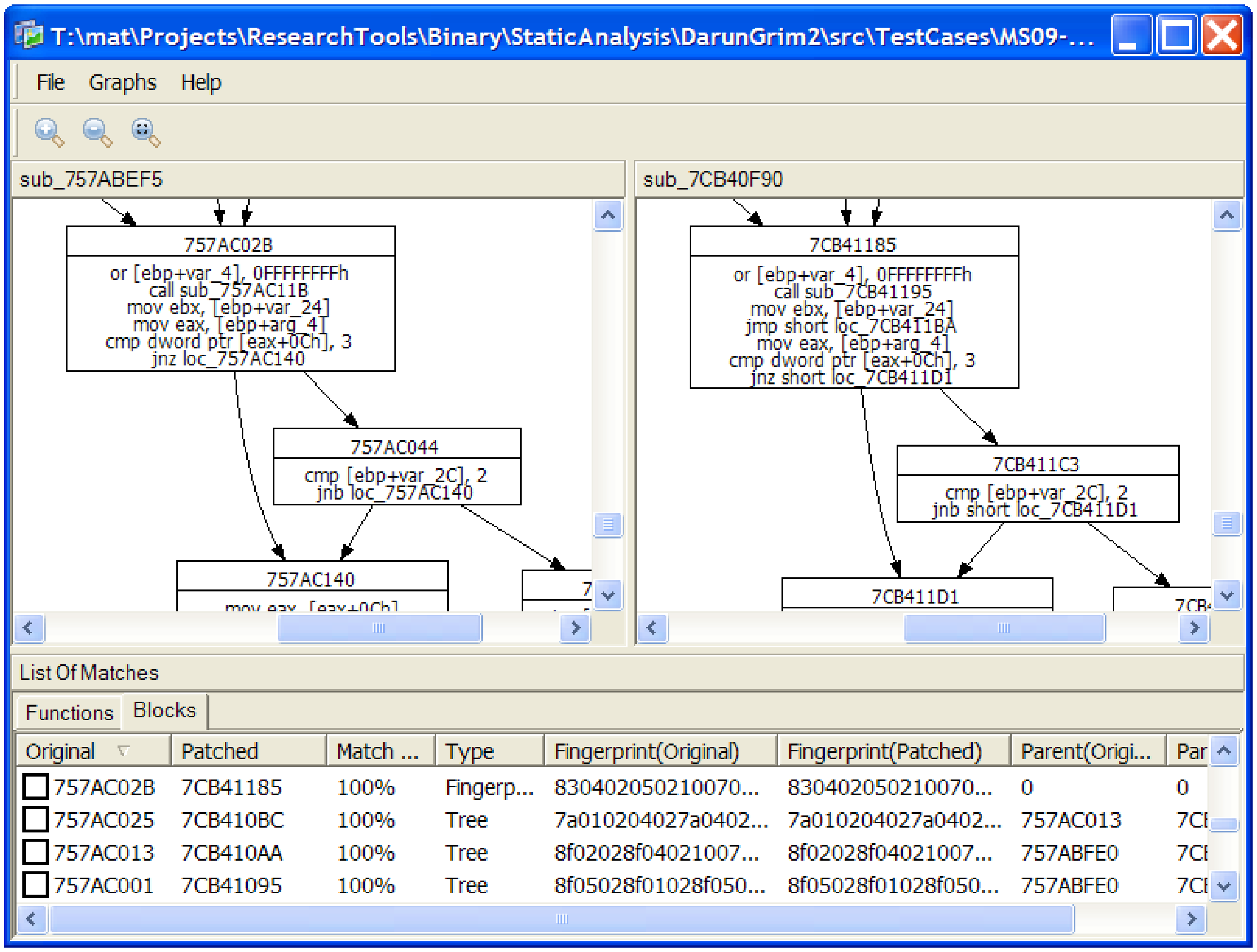
```
757AC02B
or [ebp+var_4], 0FFFFFFFFh
call sub_757AC11B
mov ebx, [ebp+var_24]
mov eax, [ebp+arg_4]
cmp dword ptr [eax+0Ch], 3
jnz loc_757AC140

757AC044
cmp [ebp+var_2C], 2
jnb loc_757AC140

757AC140
mov eax, [eax+0Ch]
```

```
7CB41185
or [ebp+var_4], 0FFFFFFFFh
call sub_7CB41195
mov ebx, [ebp+var_24]
jmp short loc_7CB411BA

7CB411BA
mov eax, [ebp+arg_4]
cmp dword ptr [eax+0Ch], 3
jnz short loc_7CB411D1

7CB411C3
cmp [ebp+var_2
jnb short loc_7CB
```

Original	Patched	Match ...	Type	Fingerprint(Original)	Fingerprint(Patched)	Parent(Origi...	Pare
☐ 757AC162				7a010204027a0102...			
☐ 757AC04E				2c0402			
☐ 757ABFA5	7CB4103B	100%	Finger...	06010205028f0402...	06010205028f0402...	0	0
☐ 757ABF9E	7CB41031	100%	Tree	8f0502	8f0502	757ABF8C	7CB

Illustration 2: Split Blocks(L:757AC02B=R:7CB41185+R:7CB411BA)

DarunGrim2 recognizes these split blocks and merges them in one block. After applying this merging process, here's the result of binary matching for same code blocks.


T:\mat\Projects\ResearchTools\Binary\StaticAnalysis\DarunGrim2\src\TestCases\MS09-...

File Graphs Help

sub_757ABEF5

```
757AC02B
or [ebp+var_4], 0FFFFFFFFh
call sub_757AC11B
mov ebx, [ebp+var_24]
mov eax, [ebp+arg_4]
cmp dword ptr [eax+0Ch], 3
jnz loc_757AC140

757AC044
cmp [ebp+var_2C], 2
jnb loc_757AC140

757AC140
mov eax, [eax+0Ch]
```

sub_7CB40F90

```
7CB41185
or [ebp+var_4], 0FFFFFFFFh
call sub_7CB41195
mov ebx, [ebp+var_24]
jmp short loc_7CB411BA
mov eax, [ebp+arg_4]
cmp dword ptr [eax+0Ch], 3
jnz short loc_7CB411D1

7CB411C3
cmp [ebp+var_2C], 2
jnb short loc_7CB411D1

7CB411D1
```

List Of Matches

Functions | Blocks

Original	Patched	Match ...	Type	Fingerprint(Original)	Fingerprint(Patched)	Parent(Origi...	Par
757AC02B	7CB41185	100%	Fingerp...	830402050210070...	830402050210070...	0	0
757AC025	7CB410BC	100%	Tree	7a010204027a0402...	7a010204027a0402...	757AC013	7CI
757AC013	7CB410AA	100%	Tree	8f02028f04021007...	8f02028f04021007...	757ABFE0	7CI
757AC001	7CB41095	100%	Tree	8f05028f01028f050...	8f05028f01028f050...	757ABFE0	7CI

Illustration 3: Now the blocks are merged in right(7CB41185)

You can see that all the unmatched blocks are now matched.

Hot Patching

Microsoft's binaries sometimes has hot patch instructions just before the real function starts. Here's an example of a function with hot patch preamble.

```
W32TimeGetNetlogonServiceBits
patched has this:
.text:765D1E9C ; int __stdcall sub_765D1E9C(unsigned __int8 *NetworkAddr,int)
.text:765D1E9C sub_765D1E9C    proc near              ; CODE XREF: sub_765A909A+53E#p
.text:765D1E9C
.text:765D1E9C Binding        = dword ptr -24h
.text:765D1E9C String        = dword ptr -1Ch
.text:765D1E9C var_18         = dword ptr -18h
.text:765D1E9C var_10         = dword ptr -10h
.text:765D1E9C var_4          = dword ptr -4
.text:765D1E9C NetworkAddr     = dword ptr  8
```

```
.text:765D1E9C arg_4          = dword ptr  0Ch
.text:765D1E9C
.text:765D1E9C                mov     eax, eax
.text:765D1E9E
.text:765D1E9E ; __stdcall W32TimeGetNetlogonServiceBits(x, x)
.text:765D1E9E _W32TimeGetNetlogonServiceBits@8:
.text:765D1E9E                push    ebp
.text:765D1E9F                mov     ebp, esp
.text:765D1EA1                push    0FFFFFFFFh
.text:765D1EA3                push    offset dword_765D1F80
```

The red part is the hot patching instruction, it's usually "mov eax,eax" which has no effect but to reserve space for hot patch installation. It's 2 bytes instruction which can make the hot patching process race-condition free.

Sometimes original binary doesn't have it and patched binary has it. That makes the basic block looks modified. So DarunGrim2 ignores any hot patching instructions. When a instruction that is doing "mov" operation on same register at the start of a function, DarunGrim2 ignore them when generating basic block fingerprint.

Basic Blocks in Multiple Functions

Usually one basic block belongs to one function. But for optimization reason, there are some cases that one basic block can be part of multiple functions.
Here's a basic block at 41fbc2 from Windows kernel disassembly. The IDA disassembler reports it part of "MmCopyToCachedPage" function.

```
.text:0041FBC2 loc_41FBC2:                      ; CODE XREF: MmCopyToCachedPage(x,x,x,x,x)+DF#j
.text:0041FBC2                mov     eax, [ebp-44h]
.text:0041FBC5                test    byte ptr [eax], 1
.text:0041FBC8                jz      loc_44CA3F
```

But from code flow analysis using custom tool, we found that the specific basic block belongs to more than one function.

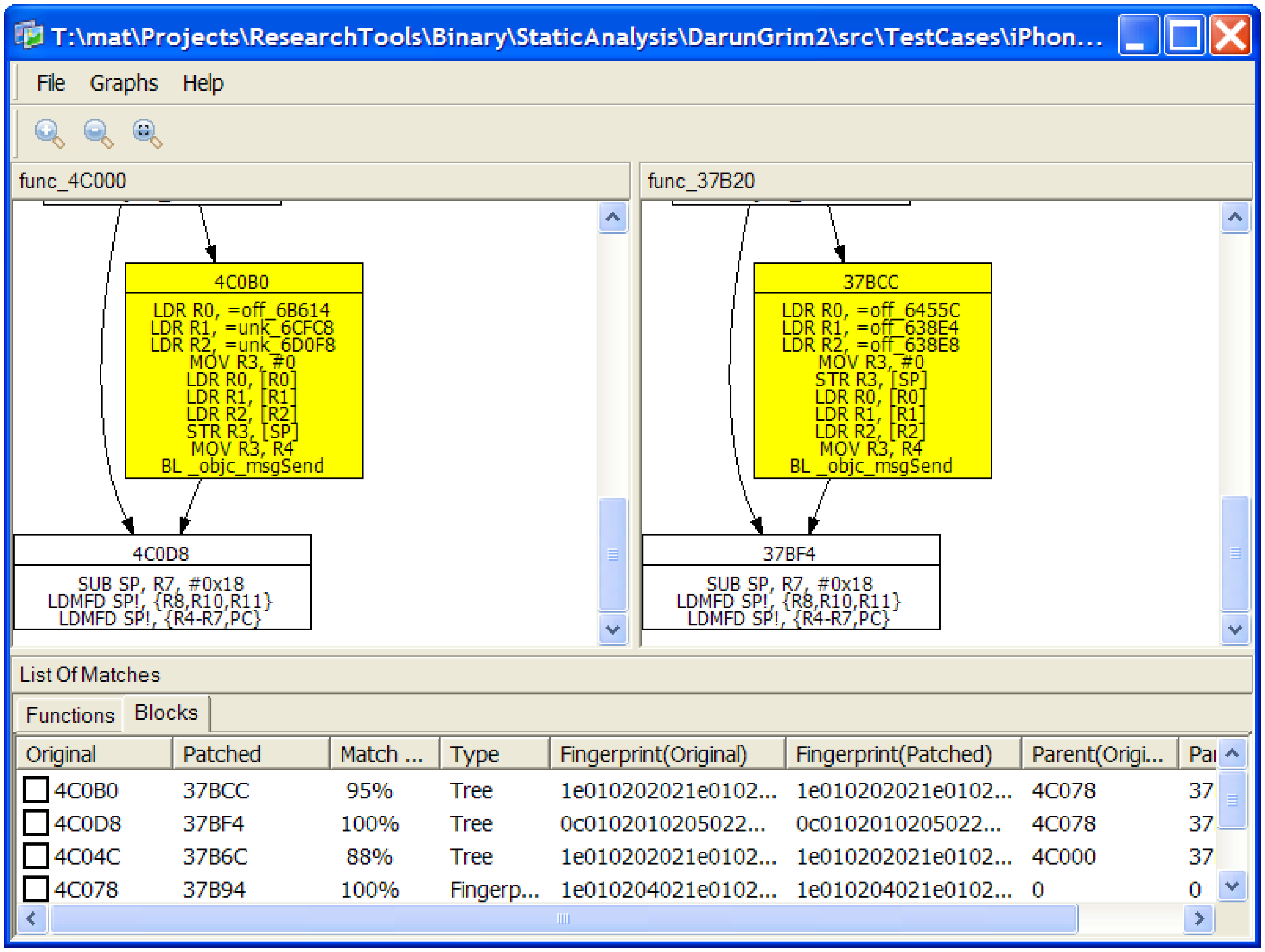
```
Function 41d3c9(MmUnmapViewInSystemCache): 41d3c9 41d431 41d43b 41d2c4 41d488 41d440 41d5bc 41d2cd 41d492 41d7b4
41d445 41d5e0 41d339 41d312 41d4aa 41c856 41d469 41d349 41d355 41d662 41d31d 41d5e8 41d4b4 41c86b 41c94e 41d474 427991
41d36c 44c7d4 41d66d 41d7d7 41d32b 41d5f2 41d872 41c870 41d482 44c7b8 42975c 41d378 44c80c 41d686 41d649 41d640 41c878
44c7ed 44c7c0 41d37c 44c818 41d691 44c7fe 41c87e 41c888 41d395 44c7de 44c83d 41d69a 44c83b 41c8e4 41c883 41c89a 41d3a1
42931c 44c821 44c842 41d6a4 429739 41c8a4 421515 41d3a9 44c84b 440761 41c8bd 440843 4407b2 4407a7 41c8c8 44c831 440852
44084a 435352 4407c0 4352d6 41c8d6 41c8cf 440854 440747 44c8a2 4407cf 4408a1 4352f5 4352dd 41c8d3 41c8fc 44c9fe 440861
4408c0 440752 44c8b1 44c8cd 4407e7 4352fd 44c874 4352e7 44ca07 440875 435360 44c8f6 4408cd 44c8b9 44080a 44c8e8 435316
44c8ff 44082b 4408d6 43536e 440816 44ca25 43531f 435378 4408fa 44081a 44ca3f 43532c 44c920 440905 440825 41fbce 435337
44c931 44c92a 41fbdb 44ca7c 43533f 44c936 41fbe4 41fe6c 44ca90 44ca81 43534a 44c894 44c947 44c93a 41fbf2 41fbc2 41fe75
44ca9a 43f4e2 44c963 44c94d 41fbfc 43f19f 41fe7e 44cad8 41fcac 41fdd9 44c985 44c96d 44c987 41fc06 41f006 44ca74 43f1b3 41fe89
41fcaf 41fddf 41fe3d 44c971 44c98b 44c989 41fc1e 43f1d1 43f1dd 41fe93 41fcca 43f41d 41fe27 41fe40 44c9f7 44c98f 429e27 41fc4a
43f1d7 43f1eb 43f4c3 41fe9c 41fcf4 44ccb4 44caa2 43f434 44c995 44c9b0 41fc54 43f203 41fea9 41fd09 43f4fe 44cd24 44cd0d 44caad
43f452 44cabe 44c9ca 429d99 41fc5c 43f23f 43f245 41fca7 41fd10 41fd1c 41fd1e 44cd32 44cd2c 44cd12 44cac6 44c9cf 44ca50 429dc0
44cb59 43f296 44cc70 41fd45 44cd3a 44c9d8 429df1 43f344 43f2c3 44cafb 41fd61 44cd3d 44c9df 44ca5f 429e06 43f35e 44cb70
43f2db 44cb15 41fdcf 41fd83 44c9e4 429e10 43f4f0 43f36c 44cb83 44cb97 43f319 43f3f9 41fd89 44c9f4 44c9eb 429e19 43f371 43f377
44cb88 44cb8e 44cba6 44cb9c 43f40b 44cd44 41fd91 41efe7 43f3be 44cd60 41fdaa 43f3ca 44cbc6 41fdc6 44cca7 43f3db 44cbe1 43f3e0
43f3e6 44cbf0 44cbe6 43f3ea 44cc0e 44cc2b 44cc3e 44cc32 44cc69 44cc58 44cc5c

Function 41fb11(MmCopyToCachedPage): 41fb11 41fb53 41fdd9 41fbc2 41fddf 41fe3d 44ca3f 41fbce 41fe27 41fe40 41fbdb 44ca7c
41fbe4 41fe6c 44ca90 44ca81 41fbf2 41fe75 44ca9a 43f4e2 41fbfc 43f19f 41fe7e 44cad8 41fcac 41fc06 41f006 44ca74 43f1b3 41fe89
41fcaf 41fc1e 43f1d1 43f1dd 41fe93 41fcca 43f41d 429e27 41fc4a 43f1d7 43f1eb 43f4c3 41fe9c 41fcf4 44ccb4 44caa2 43f434 41fc54
43f203 41fea9 41fd09 43f4fe 44cd24 44cd0d 44caad 43f452 44cabe 429d99 41fc5c 43f23f 43f245 41fca7 41fd10 41fd1c 41fd1e 44cd32
```

```
44cd2c 44cd12 44cac6 44ca50 429dc0 44cb59 43f296 44cc70 41fd45 44cd3a 429df1 43f344 43f2c3 44cafb 41fd61 44cd3d 44ca5f
429e06 43f35e 44cb70 43f2db 44cb15 41fdcf 41fd83 429e10 43f4f0 43f36c 44cb83 44cb97 43f319 43f3f9 41fd89 429e19 43f371 43f377
44cb88 44cb8e 44cba6 44cb9c 43f40b 44cd44 41fd91 41efe7 43f3be 44cd60 41fdaa 43f3ca 44cbc6 41fdc6 44cca7 43f3db 44cbe1 43f3e0
43f3e6 44cbf0 44cbe6 43f3ea 44cc0e 44cc2b 44cc3e 44cc32 44cc69 44cc58 44cc5c

Function 440702(MiRemoveMappedPtes): 440702 440854 440745 44c9fe 440861 440747 44ca07 440875 435360 4408c0 440752
44c8f6 4408cd 44c842 440761 44c8ff 44082b 4408d6 44c84b 4407b2 4407a7 440843 435378 4408fa 435352 4407c0 4352d6 440852
44084a 44c920 440905 44c8a2 4407cf 4408a1 4352f5 4352dd 44c931 44c92a 44c8b1 44c8cd 4407e7 4352fd 44c874 4352e7 44c936
44c8b9 44080a 44c8e8 435316 44c947 44c93a 43536e 440816 44ca25 43531f 44c963 44c94d 44081a 44ca3f 43532c 44c985 44c96d
44c987 440825 41fbce 435337 44c971 44c98b 44c989 41fbdb 44ca7c 43533f 44c9f7 44c98f 41fbe4 41fe6c 44ca90 44ca81 43534a
44c894 44c995 44c9b0 41fbf2 41fbc2 41fe75 44ca9a 43f4e2 44c9ca 41fbfc 43f19f 41fe7e 44cad8 41fcac 41fdd9 44c9cf 41fc06 41f006
44ca74 43f1b3 41fe89 41fcaf 41fddf 41fe3d 44c9d8 41fc1e 43f1d1 43f1dd 41fe93 41fcca 43f41d 41fe27 41fe40 44c9df 429e27 41fc4a
43f1d7 43f1eb 43f4c3 41fe9c 41fcf4 44ccb4 44caa2 43f434 44c9e4 41fc54 43f203 41fea9 41fd09 43f4fe 44cd24 44cd0d 44caad 43f452
44cabe 44c9f4 44c9eb 429d99 41fc5c 43f23f 43f245 41fca7 41fd10 41fd1c 41fd1e 44cd32 44cd2c 44cd12 44cac6 44ca50 429dc0
44cb59 43f296 44cc70 41fd45 44cd3a 429df1 43f344 43f2c3 44cafb 41fd61 44cd3d 44ca5f 429e06 43f35e 44cb70 43f2db 44cb15 41fdcf
41fd83 429e10 43f4f0 43f36c 44cb83 44cb97 43f319 43f3f9 41fd89 429e19 43f371 43f377 44cb88 44cb8e 44cba6 44cb9c 43f40b
44cd44 41fd91 41efe7 43f3be 44cd60 41fdaa 43f3ca 44cbc6 41fdc6 44cca7 43f3db 44cbe1 43f3e0 43f3e6 44cbf0 44cbe6 43f3ea 44cc0e
44cc2b 44cc3e 44cc32 44cc69 44cc58 44cc5c
```

From the above analysis result, you can see that basic block at 41fbc2(in yellow color) is parts of function "MmUnmapViewInSystemCache", "MmCopyToCachedPage" and "MiRemoveMappedPtes". This kind of multiple function ownership of a basic block can prevent proper binary diffing process. The limitation with IDA is that it only supports one function match for one basic block. DarunGrim2 solves this problem by doing custom CFG analysis and make it possible for a basic block belong multiple functions.

Instruction Reordering

In real world, instruction reordering is not happening a lot. Especially with Microsoft's binaries, we didn't see that much of instruction reordering cases. But during ARM binaries diffing experiments, we found that there are a lot of instruction reordering happen over each releases. "Illustration 4: Instruction Reordering in ARM Binaries" is a diffing result from iPhone 2.2. vs 3.0 binary. The function in question is matched by name matching, but the root node is colored yellow, which means the basic block is different.


T:\mat\Projects\ResearchTools\Binary\StaticAnalysis\DarunGrim2\src\TestCases\iPhon...
File
Graphs
Help
func_4C000
func_37B20
List Of Matches
Functions
Blocks

```
4C0B0
LDR R0, =off_6B614
LDR R1, =unk_6CFC8
LDR R2, =unk_6D0F8
MOV R3, #0
LDR R0, [R0]
LDR R1, [R1]
LDR R2, [R2]
STR R3, [SP]
MOV R3, R4
BL _objc_msgSend

4C0D8
SUB SP, R7, #0x18
LDMFD SP!, {R8,R10,R11}
LDMFD SP!, {R4-R7,PC}
```

```
37BCC
LDR R0, =off_6455C
LDR R1, =off_638E4
LDR R2, =off_638E8
MOV R3, #0
STR R3, [SP]
LDR R0, [R0]
LDR R1, [R1]
LDR R2, [R2]
MOV R3, R4
BL _objc_msgSend

37BF4
SUB SP, R7, #0x18
LDMFD SP!, {R8,R10,R11}
LDMFD SP!, {R4-R7,PC}
```

Original	Patched	Match ...	Type	Fingerprint(Original)	Fingerprint(Patched)	Parent(Origi...	Pa
☐ 4C0B0	37BCC	95%	Tree	1e010202021e0102...	1e010202021e0102...	4C078	37
☐ 4C0D8	37BF4	100%	Tree	0c0102010205022...	0c0102010205022...	4C078	37
☐ 4C04C	37B6C	88%	Tree	1e010202021e0102...	1e010202021e0102...	4C000	37
☐ 4C078	37B94	100%	Fingerp...	1e010204021e0102...	1e010204021e0102...	0	0

Illustration 4: Instruction Reordering in ARM Binaries

So we checked the disassembly of two basic blocks.

Original	*Patched*
STMFD SP!, {R4-R7,LR}	STMFD SP!, {R4-R7,LR}
ADD R7, SP, #0x14+var_8	ADD R7, SP, #0x14+var_8
LDR R3, =(off_3AFD9AAC - 0x32FF9A80)	SUB SP, SP, #0xC
SUB SP, SP, #0xC	LDR R3, =(off_3B2CF6C8 - 0x33328E08)
LDR R1, =(off_3AFD86B8 - 0x32FF9A88)	LDR R1, =(off_3B2CDE70 - 0x33328E10)
LDR R3, [PC,R3]	STR R0, [SP,#0x20+var_20]
STR R0, [SP,#0x20+var_20]	LDR R3, [PC,R3]
LDR R1, [PC,R1] ; "initWithPath:"	MOV R0, SP
MOV R0, SP	LDR R1, [PC,R1] ; "initWithPath:"
MOV R6, R2	MOV R6, R2
STR R3, [SP,#0x20+var_1C]	STR R3, [SP,#0x20+var_1C]
BL _objc_msgSendSuper2	BL _objc_msgSendSuper2
SUBS R5, R0, #0	SUBS R5, R0, #0
BEQ loc_32FF9B84	BEQ loc_33328F08

Table 1: The original disassembly(same colors for same instructions)

The original and the patched basic block is basically same. The only difference is the order of each instructions many of the instructions are swapped.

So to solve this instruction reordering problem, we use grouping using variable tracing and sorting method. We can follow each register or displacement variable's change and usage paths. For example "Illustration 5: Part of register,displacement variable and call arguments trace of the patched basic block" shows some part of patched binaries basic block's variable traces. DarunGrim2 currently support register, displacement variable and call arguments tracing.

By using variable tracing we can group each instruction nodes. There can be multiple groups and they are independent each other and the order of instruction between them will not affect each other. We calculate hash value from each groups. The hash will use instruction type and operand's type and value. We intentionally ignores any immediate value or memory references. They tend to be random across each builds and will prevent proper calculation. With the hash value calculated, we sort each block and list them in sorted order.

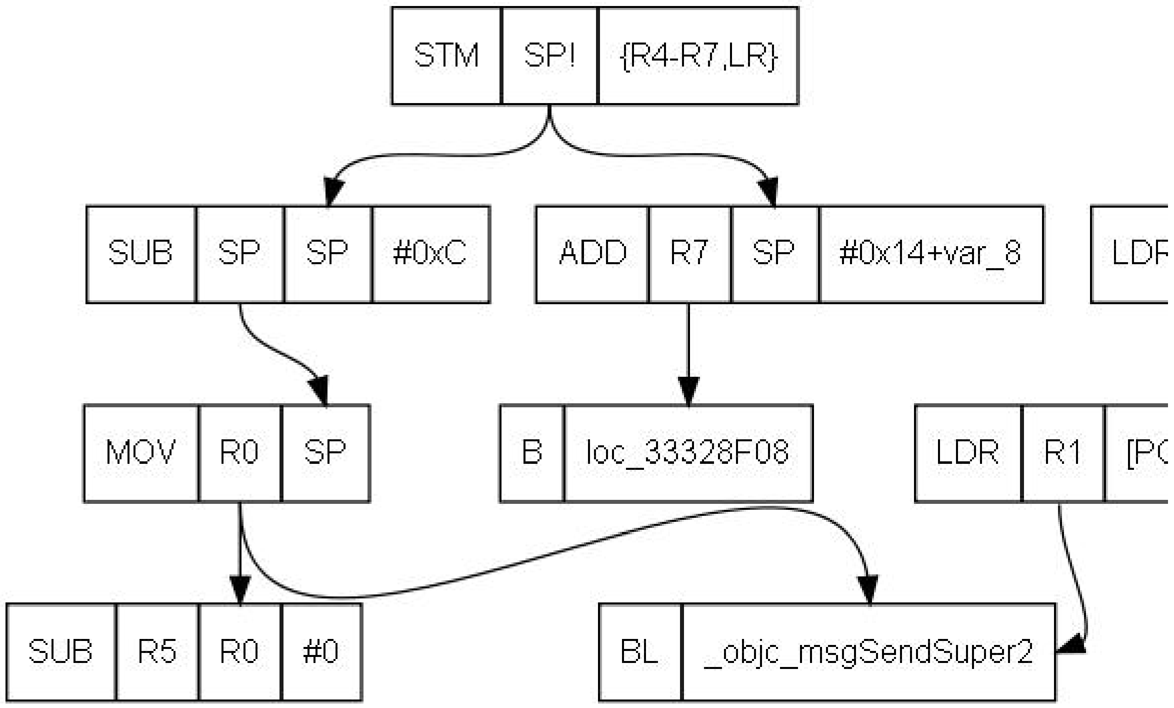


Illustration 5: Part of register,displacement variable and call arguments trace of the patched basic block

"Table 2: After Grouping And Sorting" shows the result of grouping and sorting. They show same sequence of instructions except immediate and memory reference values. This instruction re-reordering process consume more CPU than original way of gathering sequential fingerprint. So this feature can be optionally used when necessary.

Original	***Patched***
STMFD SP!, {R4-R7,LR}	STMFD SP!, {R4-R7,LR}
ADD R7, SP, #0x14+var_8	ADD R7, SP, #0x14+var_8
SUB SP, SP, #0xC	SUB SP, SP, #0xC
BEQ loc_32FF9B84	BEQ loc_33328F08
MOV R0, SP	MOV R0, SP
SUBS R5, R0, #0	SUBS R5, R0, #0
STR R0, [SP,#0x20+var_20]	STR R0, [SP,#0x20+var_20]
LDR R3, =(off_3AFD9AAC - 0x32FF9A80)	LDR R3, =(off_3B2CF6C8 - 0x33328E08)
LDR R3, [PC,R3]	LDR R3, [PC,R3]
STR R3, [SP,#0x20+var_1C]	STR R3, [SP,#0x20+var_1C]
LDR R1, =(off_3AFD86B8 - 0x32FF9A88)	LDR R1, =(off_3B2CDE70 - 0x33328E10)
LDR R1, [PC,R1] ; "initWithPath:"	LDR R1, [PC,R1] ; "initWithPath:"
BL _objc_msgSendSuper2	BL _objc_msgSendSuper2
MOV R6, R2	MOV R6, R2

Table 2: After Grouping And Sorting

Examples

We will show the result of binary diffing of real Microsoft patches to show the effectiveness of identifying patched points. We will use 3-4 critical-rated patches as examples. The critical and hard-to-find vulnerabilities doesn't need to be hard to pinpoint in the patches. We will use DarunGrim2 to demonstrate this.

Microsoft's Binaries

Why are Microsoft's binaries easy targets?

They ship security patches in every second Tuesday of the Month. So the patches tend to contain only security fixing codes. Usually there is no feature enhancements or other types of patches are included in the patch. The feature enhancements are in service pack, not in monthly patch. So you can get pure security related patched parts with binary diffing.

They also provide symbols. They usually provide symbols for system dlls and drivers and kernel images. They don't provide symbol for non core OS products like IIS or Active Directory or Microsoft Office. But many critical vulnerabilities are found in system DLLs, drivers and kernel, the attacker can use the symbols Microsoft provides to get more confidence in binary diffing. And it also saves a lot of time understanding the codes that modified.

They don't do any code obfuscation. Aside from code optimization, they usually never do code obfuscation. Actually if they do code obfuscation, it will make a lot of problems. For example, it can conflict other products and also make debugging process very hard for them. So I think they will never be able to use code obfuscation technique for their shipping products.

This makes the process of binary diffing more easy and cost effective. Amount of modified codes in patch is relatively small compared to other vendors. This means Microsoft is doing well in their code maintenance. In the point of binary differs' view, it's a low hanging fruit.

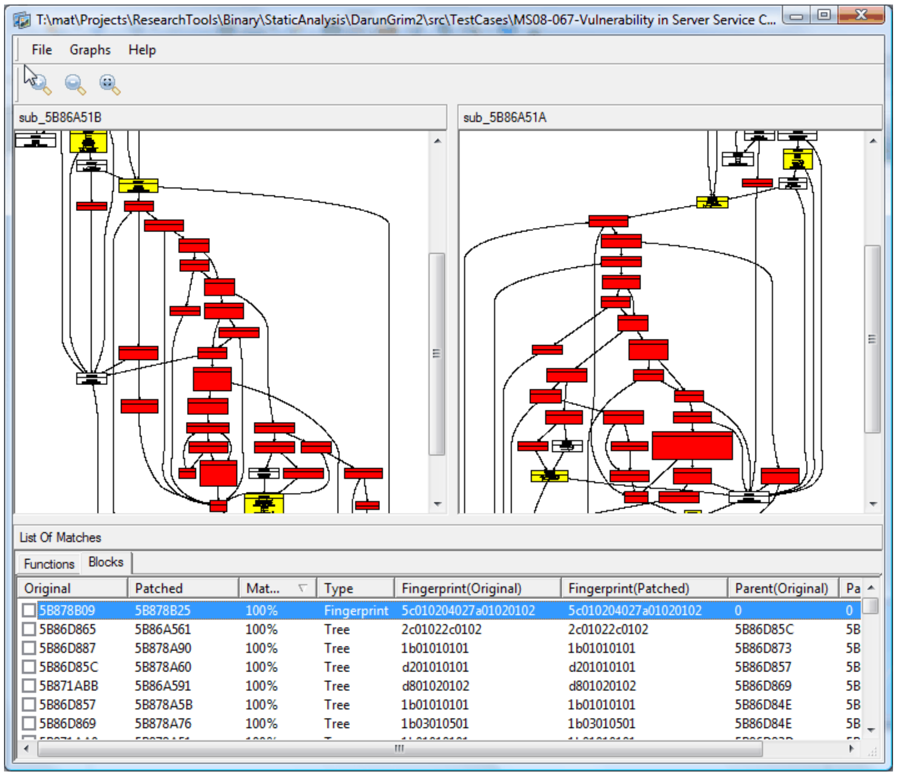
Gathering Binaries

It's very easy to gather Microsoft's binaries. You can just visit each patches page. For example for MS08-067 issue, you can visit http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx and you can download binaries that matching your operating system. Then you can extract the binaries inside the package using "/x" option. You can find older version of the binaries from your harddisk and open IDA and load the files and just start the diffing. This whole process usually takes little than 30 minutes.

The infamous MS08-067(which was exploited by Conficker)

MS08-067 is a very interesting patch. It's patching stack buffer overflow problem inside netapi32.dll. And the real problem is that the stack overflow can be reached through anonymous pipe, which means this exploit can be exploited remotely without any credentials. This vulnerability was used with Conficker worm to propagate through internal network after it infects any machine in the network.

From the point of binary diffing, MS08-067 is a very easy target. Identifying the modified block just takes few minutes. And there is only 2 functions changed. And one is a change in calling convention. So there is only one function changed.


T:\mat\Projects\ResearchTools\Binary\StaticAnalysis\DarunGrim2\src\TestCases\MS08-067-Vulnerability in Server Service C...
File
Graphs
Help
sub_5B86A51B
sub_5B86A51A
List Of Matches
Functions
Blocks

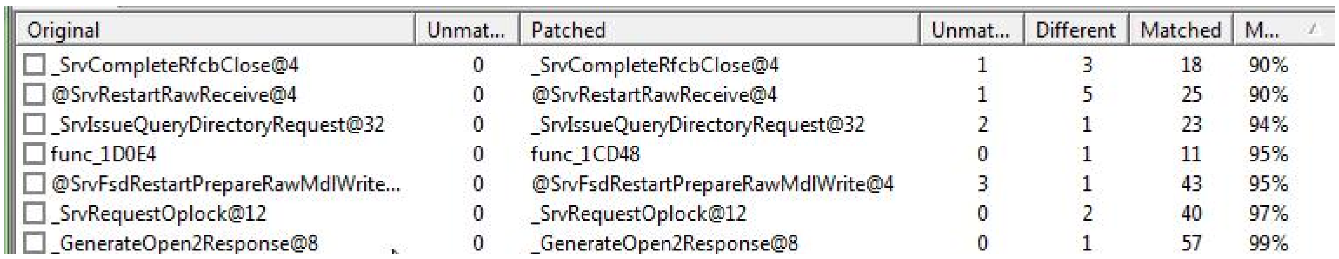
Original	Patched	Mat...	Type	Fingerprint(Original)	Fingerprint(Patched)	Parent(Original)	Pa
5B878B09	5B878B25	100%	Fingerprint	5c010204027a01020102	5c010204027a01020102	0	0
5B86D865	5B86A561	100%	Tree	2c01022c0102	2c01022c0102	5B86D85C	5B
5B86D887	5B878A90	100%	Tree	1b01010101	1b01010101	5B86D873	5B
5B86D85C	5B878A60	100%	Tree	d201010101	d201010101	5B86D857	5B
5B871ABB	5B86A591	100%	Tree	d801020102	d801020102	5B86D869	5B
5B86D857	5B878A5B	100%	Tree	1b01010101	1b01010101	5B86D84E	5B
5B86D869	5B878A76	100%	Tree	1b03010501	1b03010501	5B86D84E	5B

Illustration 6: Differences between two matching functions are massive

Of course, you need more research and analysis of the patched function to get what was really causing the vulnerabilities happening. It's not in the range of binary diffing art itself. Binary diffing is a starting point of whole patch analysis. Especially for MS08-067 patch, the modified codes are enormous, we can easily guess that it might be a huge mistake in processing some data. More analysis will reveal the actual vulnerabilities.

MS08-063: DarunGrim2 vs bindiff

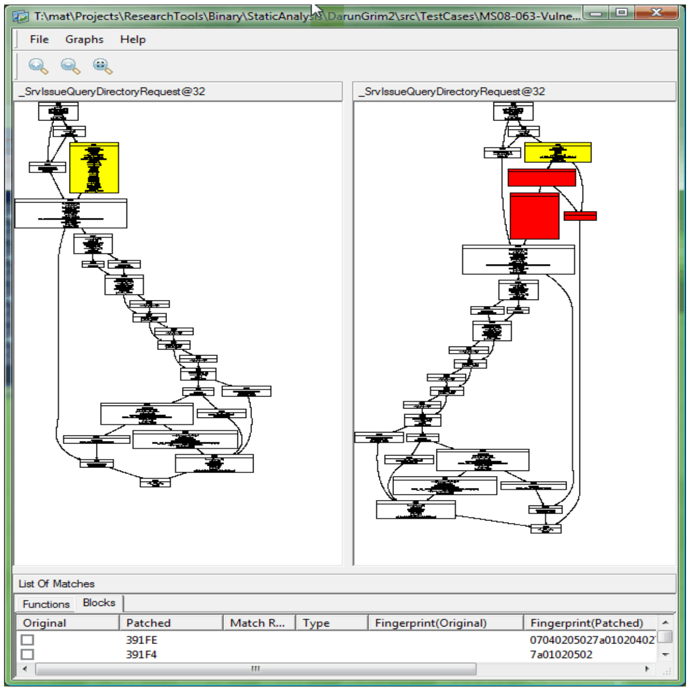
Binary diffing with DarunGrim2 will show following functions are modified. You can check them one by one to find any interesting basic block additions. The "Illustration 11: Screenshot from bindiff manual" shows the list of modified functions.



Original	Unmat...	Patched	Unmat...	Different	Matched	M...
☐ _SrvCompleteRfcbClose@4	0	_SrvCompleteRfcbClose@4	1	3	18	90%
☐ @SrvRestartRawReceive@4	0	@SrvRestartRawReceive@4	1	5	25	90%
☐ _SrvIssueQueryDirectoryRequest@32	0	_SrvIssueQueryDirectoryRequest@32	2	1	23	94%
☐ func_1D0E4	0	func_1CD48	0	1	11	95%
☐ @SrvFsdRestartPrepareRawMdlWrite...	0	@SrvFsdRestartPrepareRawMdlWrite@4	3	1	43	95%
☐ _SrvRequestOplock@12	0	_SrvRequestOplock@12	0	2	40	97%
☐ _GenerateOpen2Response@8	0	_GenerateOpen2Response@8	0	1	57	99%

Illustration 7: Modified Functions

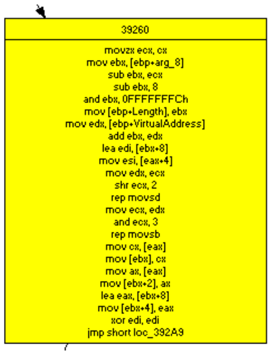
The "_SrvIssueQueryDirectoryRequest@32" has the patch that is supposed to fix the security issue. The zoomed-out call graph of the function looks like following.


T:\mat\Projects\ResearchTools\Binary\StaticAnaly...\DarunGrim2\src\TestCases\MS08-063-Vulne...
File
Graphs
Help
_SrvIssueQueryDirectoryRequest@32
_SrvIssueQueryDirectoryRequest@32
List Of Matches
Functions
Blocks

Original	Patched	Match R...	Type	Fingerprint(Original)	Fingerprint(Patched)
☐	391FE				07040205027a01020402
☐	391F4				7a01020502

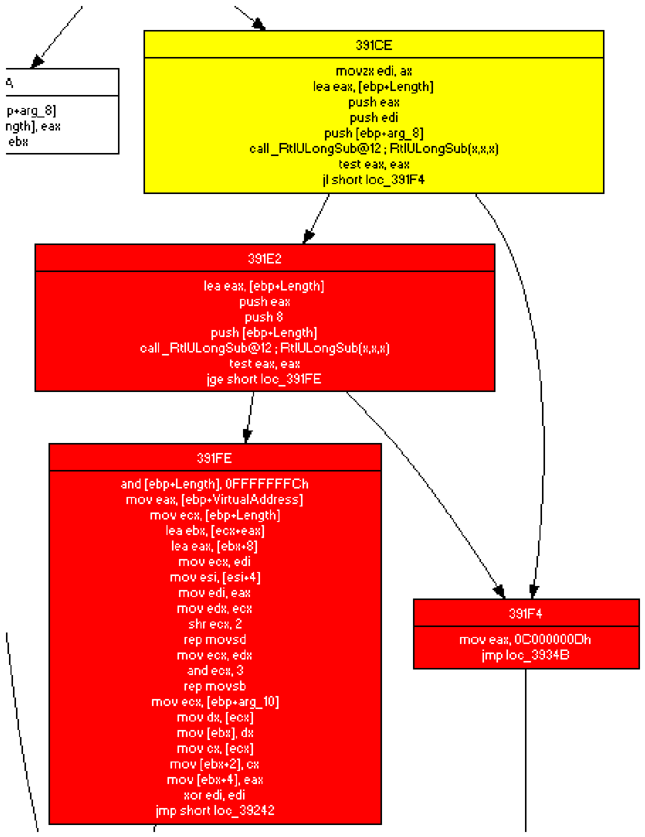
Illustration 8: Zoomed out view of the modified function(_SrvIssueQueryDirectoryRequest@32)

The original basic block that is modified is like following.



```
39260
movzx ecx, cx
mov ebx, [ebp+arg_8]
sub ebx, ecx
sub ebx, 8
and ebx, 0FFFFFFFCh
mov [ebp+Length], ebx
mov edx, [ebp+VirtualAddress]
add ebx, edx
lea edi, [ebx+8]
mov esi, [eax+4]
mov edx, ecx
shr ecx, 2
rep movsd
mov ecx, edx
and ecx, 3
rep movsb
mov cx, [eax]
mov [ebx], cx
mov ax, [eax]
mov [ebx+2], ax
lea eax, [ebx+8]
mov [ebx+4], eax
xor edi, edi
jmp short loc_392A9
```

Illustration 9: Original basic block that is modified



```
4,
p+arg_8]
ngth], eax
ebx
```

```
391CE
movzx edi, ax
lea eax, [ebp+Length]
push eax
push edi
push [ebp+arg_8]
call _RtlULongSub@12 ; RtlULongSub(x,x,x)
test eax, eax
jl short loc_391F4
```

```
391E2
lea eax, [ebp+Length]
push eax
push 8
push [ebp+Length]
call _RtlULongSub@12 ; RtlULongSub(x,x,x)
test eax, eax
jge short loc_391FE
```

```
391FE
and [ebp+Length], 0FFFFFFFCh
mov eax, [ebp+VirtualAddress]
mov ecx, [ebp+Length]
lea ebx, [ecx+eax]
lea eax, [ebx+8]
mov ecx, edi
mov esi, [esi+4]
mov edi, eax
mov edx, ecx
shr ecx, 2
rep movsd
mov ecx, edx
and ecx, 3
rep movsb
mov ecx, [ebp+arg_10]
mov dx, [ecx]
mov [ebx], dx
mov cx, [ecx]
mov [ebx+2], cx
mov [ebx+4], eax
xor edi, edi
jmp short loc_39242
```

```
391F4
mov eax, 0C000000Dh
jmp loc_3934B
```

Illustration 10: The patched parts

In "Illustration 8: Zoomed out view of the modified function(_SrvIssueQueryDirectoryRequest@32)" The red blocks below are the additional basic blocks and they are doing sanity checks for the data.

From the patch you can see it's calling "RtlUlongSub" function and returns C000000Dh return code according to the result. With further tracing of what data it's checking sanity, you can see what kind of vulnerabilities it's fixing.

So DarunGrim2 finds 7 modified functions. And if you look into each changes most of them have some kind of sanity checks or something meaningful changes added.

If you look into bindiff's manual page, you can see binary diffing example using this exact same binaries. But, the screenshot from the bindiff help file shows only 3 function changes.

```
SrvFsdRestartPrepareRawMdlWrite
SrvIssueQueryDirectoryRequest
SrvRestartRawReceive
```

So as Halvar mentioned in his paper, function fingerprinting using nodes, edges and call count usually have few false negatives. And sometimes they are critical if the patch is done without changing the CFG structure.

Here's the graph screenshot from bindiff help file showing "_SrvIssueQueryDirectoryRequest@32" . Primary is patched file and secondary is original file. And you can see that the diffing result is same as DarunGrim2.

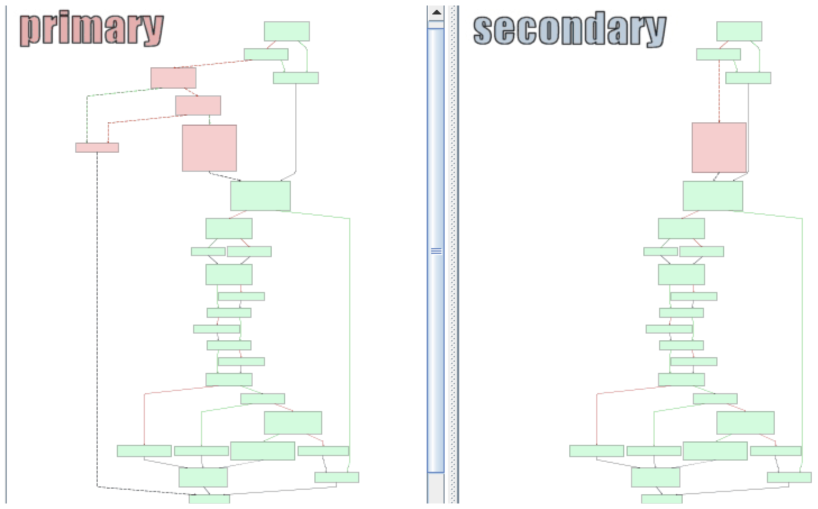
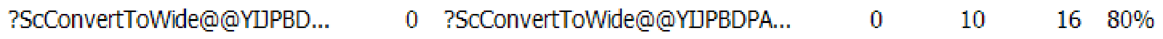


Illustration 11: Screenshot from bindiff manual

So this diffing example shows that DarunGrim2 diffing scheme is useful and practical as much as bindiff. And sometimes if you want more detailed depth of diff analysis DarunGrim2 might show better results. But this doesn't mean that which tool or algorithm is superior or inferior. Both of them has their own strength and weakness, so you need to choose the tool and algorithms according to your specific diffing needs.

MS09-020: WebDav case

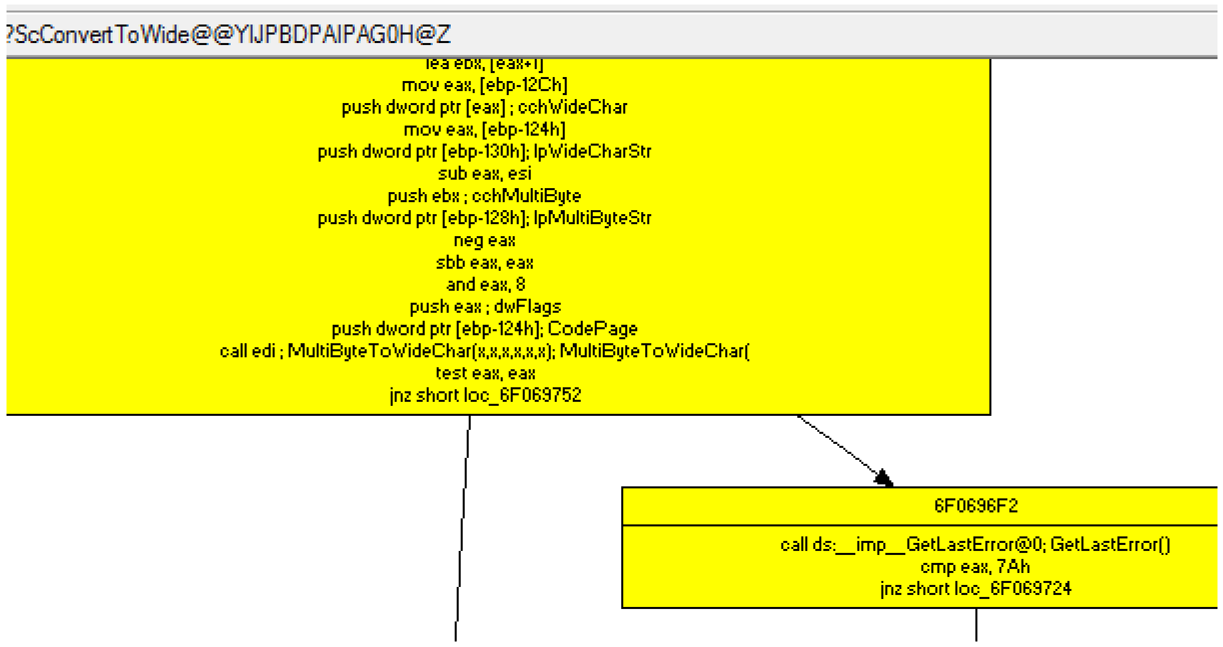
By diffing the patches, the following functions are shown as matching and modified. The interesting thing on this issue is that there is no unidentified blocks in original or patched binaries. Only 9 basic blocks are modified.



```
?ScConvertToWide@@YIJPBD...    0   ?ScConvertToWide@@YIJPBDPA...    0     10     16   80%
```

Illustration 12: The Function Match Line

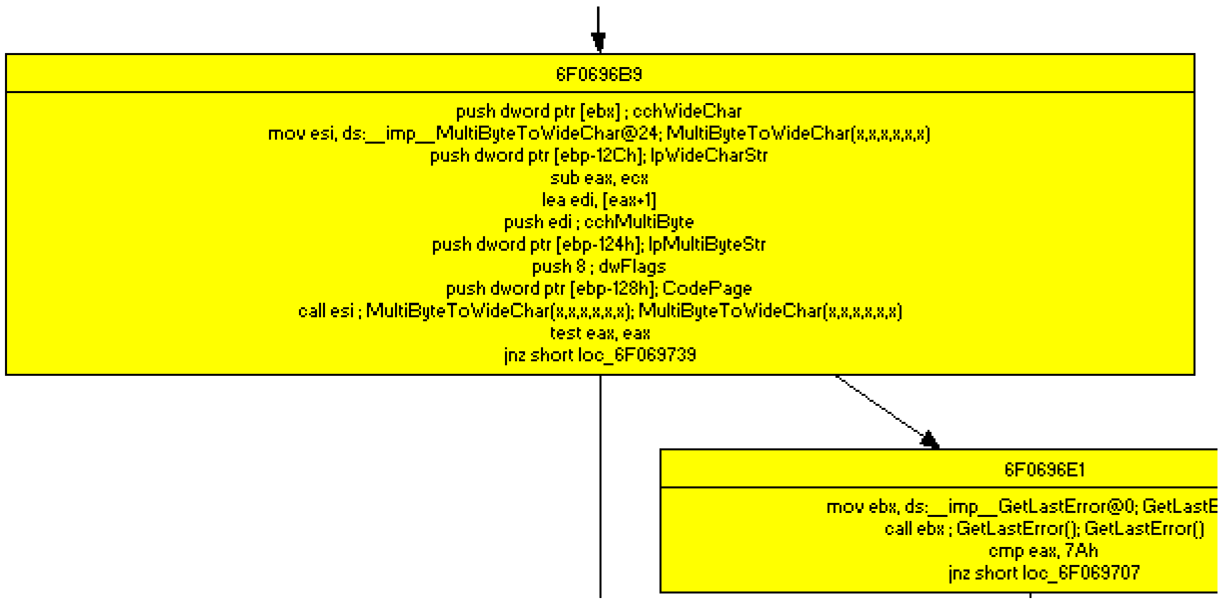
So the most significant change happened in following basic blocks. The following is showing the blocks from original binary.



```
?ScConvertToWide@@YIJPBDPAIPAG0H@Z

lea ebx, [eax+1]
mov eax, [ebp-12Ch]
push dword ptr [eax] ; cchWideChar
mov eax, [ebp-124h]
push dword ptr [ebp-130h]; lpWideCharStr
sub eax, esi
push ebx ; cchMultiByte
push dword ptr [ebp-128h]; lpMultiByteStr
neg eax
sbb eax, eax
and eax, 8
push eax ; dwFlags
push dword ptr [ebp-124h]; CodePage
call edi ; MultiByteToWideChar(x,x,x,x,x,x); MultiByteToWideChar(
test eax, eax
jnz short loc_6F069752

6F0696F2
call ds:__imp__GetLastError@0; GetLastError()
cmp eax, 7Ah
jnz short loc_6F069724
```

Illustration 13: Original binary

And the following is the basic blocks from patched binary.



```
6F0696B9
push dword ptr [ebx] ; cchWideChar
mov esi, ds:__imp__MultiByteToWideChar@24; MultiByteToWideChar(x,x,x,x,x,x)
push dword ptr [ebp-12Ch]; lpWideCharStr
sub eax, ecx
lea edi, [eax+1]
push edi ; cchMultiByte
push dword ptr [ebp-124h]; lpMultiByteStr
push 8 ; dwFlags
push dword ptr [ebp-128h]; CodePage
call esi ; MultiByteToWideChar(x,x,x,x,x,x); MultiByteToWideChar(x,x,x,x,x,x)
test eax, eax
jnz short loc_6F069739

6F0696E1
mov ebx, ds:__imp__GetLastError@0; GetLastE
call ebx ; GetLastError(); GetLastError()
cmp eax, 7Ah
jnz short loc_6F069707
```

Illustration 14: Patched Binary

The difference is unnoticeable at first. But you can find that the 2nd argument for "MultiByteToWideChar" call is changed. The original binary is getting it using following assembly sequences.

```
6F0695EA mov     esi, 0FDE9h
,,,,
6F069641 call    ?FIsUTF8Url@@YIHPBD@Z ; FIsUTF8Url(char const *)
6F069646 test    eax, eax
if(!eax)
{
            6F0695C3 xor     edi, edi
            6F06964A mov     [ebp-124h], edi
}else
{
            6F069650 cmp     [ebp-124h], esi
}
,,,
6F0696C9 mov     eax, [ebp-124h]
6F0696D5 sub     eax, esi
6F0696DE neg     eax
6F0696E0 sbb     eax, eax
6F0696E2 and     eax, 8
```

It's doing some arithmetics using "eax" and the result is used for the flag of the call.But if at address 6F069641, the call for "FisUTF8Url" function returns TRUE, "esi" and "[ebp-124h]" will have same value. So the flag will be 0.

MSDN(http://msdn.microsoft.com/en-us/library/dd319072(VS.85).aspx) declares 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.

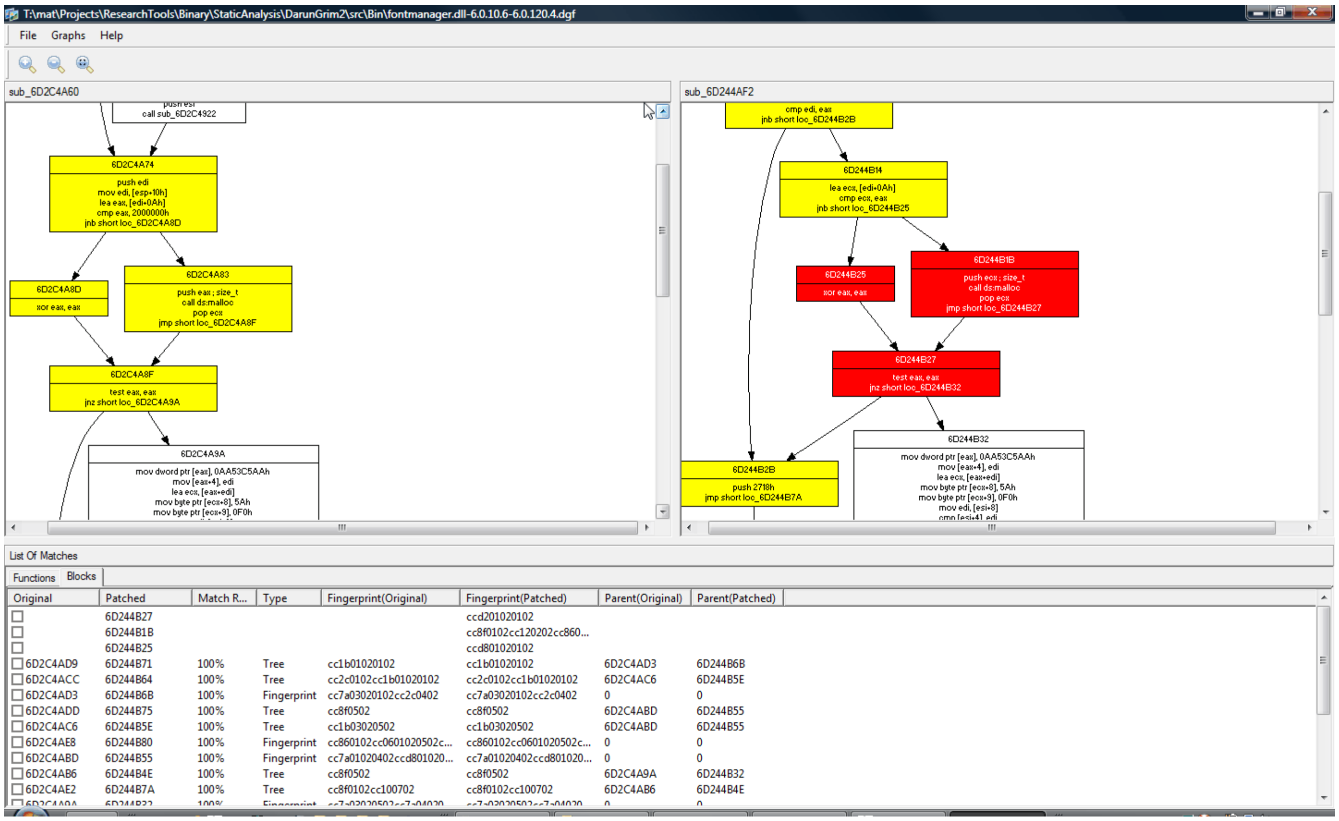
The "FIsUTF8Url" is not complete UTF8 recognizing routine, but it's doing some heuristic character range checks to see if the string looks like UTF8. Even though the string looks like UTF8, it still can be invalid UTF8 string. And you don't specify the "MB_ERR_INVALID_CHARS" flag, the function will convert the string ignoring the errors.

This vulnerability is special, because the fix doesn't incur any CFG structure change. The only change is some instructions inside blocks. CFG based matching tools like "bindiff" have possibilities to miss these kinds of patches.

Non-MS Binaries

JRE Font Manager Buffer Overflow(Sun Alert 254571)

It's also possible binary diff binaries without symbols. Sun Microsystems doesn't provide symbols for their binaries. But with binary diffing of the target binaries, you can get the following result.


T:\mat\Projects\ResearchTools\Binary\StaticAnalysis\DarunGrim2\src\Bin\fontmanager.dll-6.0.10.6-6.0.120.4.dgf
File
Graphs
Help
sub_6D2C4A60
sub_6D244AF2
List Of Matches
Functions
Blocks

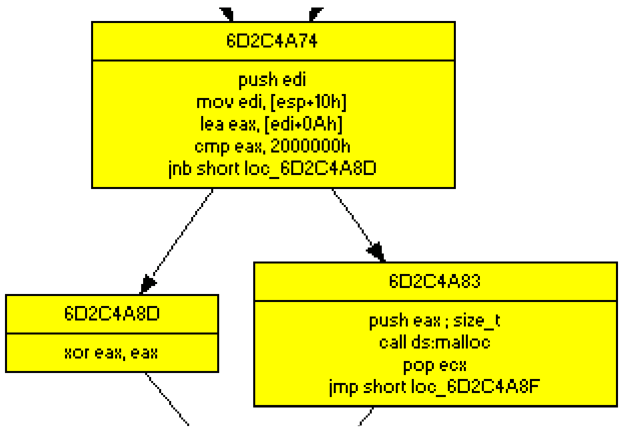
```
call sub_6D2C4922

6D2C4A74
push edi
mov edi, [esp+10h]
lea eax, [edi+0Ah]
cmp eax, 2000000h
jnb short loc_6D2C4A8D

6D2C4A8D
xor eax, eax

6D2C4A83
push eax ; size_t
call ds:malloc
pop ecx
jmp short loc_6D2C4A8F

6D2C4A8F
test eax, eax
jnz short loc_6D2C4A9A

6D2C4A9A
mov dword ptr [eax], 0AA53C5AAh
mov [eax+4], edi
lea ecx, [eax+edi]
mov byte ptr [ecx+8], 5Ah
mov byte ptr [ecx+9], 0F0h
```

```
cmp edi, eax
jnb short loc_6D244B2B

6D244B14
lea ecx, [edi+0Ah]
cmp ecx, eax
jnb short loc_6D244B25

6D244B25
xor eax, eax

6D244B1B
push ecx ; size_t
call ds:malloc
pop ecx
jmp short loc_6D244B27

6D244B27
test eax, eax
jnz short loc_6D244B32

6D244B2B
push 2718h
jmp short loc_6D244B7A

6D244B32
mov dword ptr [eax], 0AA53C5AAh
mov [eax+4], edi
lea ecx, [eax+edi]
mov byte ptr [ecx+8], 5Ah
mov byte ptr [ecx+9], 0F0h
mov edi, [esi+8]
```

Original	Patched	Match R...	Type	Fingerprint(Original)	Fingerprint(Patched)	Parent(Original)	Parent(Patched)
☐	6D244B27				ccd201020102		
☐	6D244B1B				cc8f0102cc120202cc860...		
☐	6D244B25				ccd801020102		
☐ 6D2C4AD9	6D244B71	100%	Tree	cc1b01020102	cc1b01020102	6D2C4AD3	6D244B6B
☐ 6D2C4ACC	6D244B64	100%	Tree	cc2c0102cc1b01020102	cc2c0102cc1b01020102	6D2C4AC6	6D244B5E
☐ 6D2C4AD3	6D244B6B	100%	Fingerprint	cc7a03020102cc2c0402	cc7a03020102cc2c0402	0	0
☐ 6D2C4ADD	6D244B75	100%	Tree	cc8f0502	cc8f0502	6D2C4ABD	6D244B55
☐ 6D2C4AC6	6D244B5E	100%	Tree	cc1b03020502	cc1b03020502	6D2C4ABD	6D244B55
☐ 6D2C4AE8	6D244B80	100%	Fingerprint	cc860102cc0601020502c...	cc860102cc0601020502c...	0	0
☐ 6D2C4ABD	6D244B55	100%	Fingerprint	cc7a01020402ccd801020...	cc7a01020402ccd801020...	0	0
☐ 6D2C4AB6	6D244B4E	100%	Tree	cc8f0502	cc8f0502	6D2C4A9A	6D244B32
☐ 6D2C4AE2	6D244B7A	100%	Tree	cc8f0102cc100702	cc8f0102cc100702	6D2C4AB6	6D244B4E

Illustration 15: Overview of the Difference

The unpatched binary looks like following.



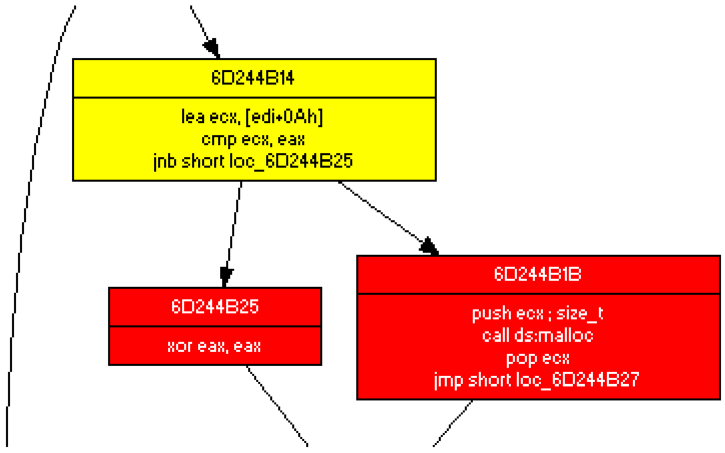
```
6D2C4A74
push edi
mov edi, [esp+10h]
lea eax, [edi+0Ah]
cmp eax, 2000000h
jnb short loc_6D2C4A8D
```

```
6D2C4A8D
xor eax, eax
```

```
6D2C4A83
push eax ; size_t
call ds:malloc
pop ecx
jmp short loc_6D2C4A8F
```

Illustration 16: The Original Parts

The patched binary has following.



```
6D244B14
lea ecx, [edi+0Ah]
cmp ecx, eax
jnb short loc_6D244B25
```

```
6D244B25
xor eax, eax
```

```
6D244B1B
push ecx ; size_t
call ds:malloc
pop ecx
jmp short loc_6D244B27
```

Illustration 17: The patched parts

From IDA, I retrieved more parts of disassembly. The original and patched parts looks like following. The patched binary has additional lines in red.

Original

```
.text:6D2C4A75          mov    edi, [esp+10h]
.text:6D2C4A79          lea    eax, [edi+0Ah]
.text:6D2C4A7C          cmp    eax, 2000000h
.text:6D2C4A81          jnb    short loc_6D2C4A8D
.text:6D2C4A83          push   eax            ; size_t
.text:6D2C4A84          call   ds:malloc
```

Patched

```
.text:6D244B06          push   edi
.text:6D244B07          mov    edi, [esp+10h]
.text:6D244B0B          mov    eax, 2000000h
.text:6D244B10          cmp    edi, eax
.text:6D244B12          jnb    short loc_6D244B2B
.text:6D244B14          lea    ecx, [edi+0Ah]
.text:6D244B17          cmp    ecx, eax
.text:6D244B19          jnb    short loc_6D244B25
.text:6D244B1B          push   ecx            ; size_t
.text:6D244B1C          call   ds:malloc
```

If you look into the red part, you can see that the part is checking edi register's value is over 2000000h.

Orignal check only ecx(which is edi+0ah) value is over 2000000h(6D2C4A7C). The additional edi value check actually prevents integer overflow. For example edi is 0xffffffff, then ecx will be

0xffffffff+0xa=0x9. In Orignal binary ecx will pass the test because it's smaller than 2000000h. But in the patched binary the cmp edi,eax in address 6D244B10 will filter big edi value. This can help prevent integer overflow.

So we can know that there's a integer overflow is happening in this part. And with further investigation we can find where the data is coming in. It must be somewhere the attacker can control.

Actually the original advisory here(iDefense Security Advisory 03.26.09: Sun Java Runtine Environment (JRE) Type1 Font Parsing Integer Signedness Vulnerability) describe the vulnerability as following

The vulnerability occurs when parsing glyph description instructions in the font file. When parsing the glyph descriptions, a 16bit signed counter is used as the index to store the next glyph point value. This counter is compared to a 32bit value that represents the maximum size of the heap buffer. Under certain conditions, the 16bit counter will be interpreted as a negative value, which allows the attacker to store data before the allocated buffer.

So we see the patch we just found just matches the vulnerability description.

Anti-Binary Diffing

So now it became crucial to make these practices more difficult and time-consuming so that earn more time for the consumers to apply patches. Even though using severe code obfuscation is not an option for Microsoft's products, they can still follow some strategies and techniques to defeat the binary diffing processes without forsaking stability and usability. We are going to show the methods and tactics to make binary differs life harder.
We will give some ideas and informations on how to prevent effective binary diffing. We are going to show tactics and algorithms that can be used to defeat the binary diffing methods. Some are related to practices and processes that the vendors are doing and others are totally technical.

We can think of some measures to prevent this kind of binary difference analysis.

Symbol Mangling

We can think about changing symbol names for the procedures. Many implementations are basically dependent on symbolic name of procedures and variables. So if we use different names for original and patched binaries then we can make name based procedure matching useless. BMAT is actually heavily dependent on name matching, and they are also doing similarities based name matching. So we need to mangle the names to avoid the heuristic name matching.

Reordering and replacing instructions

Some binary diffing tools are using code checksum for matching basic blocks. And the checksum is order dependent sometimes. In that case just reordering or replacing existing instructions to alternates will make the matching fail.

CFG Altering

Some tools like bindiff is dependent on CFG signatures of each functions. How about if ruin this signature by adding fake nodes and edges and calls. So the function signature will be modified and the fixed points identifying process will fail. By putting a branch that will never be taken, you can break CFG and make diffing algorithms based on CFG break.

By altering the structure of procedure's CFG or program's CG without changing functionality, it's possible to make binary difference analysis tools based on structural analysis fail. This method also affects isomorphic analysis based binary matching. Because the graph structure must be matched for each binaries, just putting few meaningless blocks or instructions will confuse function signaturing process. Something to note is that we need to put the CFG mangling blocks to enough number of functions. If it's concentrated around the patched function, it only makes the patched point more visible.

Use proxy call

Prevent CG Recognition
Replace call <XXX> to call <YYY> and the function YYY will look like following

```
proc YYY:
        call XXX
        ret
```

Function YYY is basically null function without any functionality other than call proxying. In case disassembler of binary differ recognizes this null functions, you can put some non-functioning garbage instruction to deceive disassembler or binary differ.

Call that never returns

Actually this idea came with some real world case where DarunGrim2 was deluded where RpcRaiseException broke CFG and prevented matching of functions. This will break CFG seriously. If you put a call that is never returning and put a function block just next to that "call XXX" instruction.

Change instructions like following:

```
jz A
```

To something like following.

```
1: call B
2: proc B:
3:      add esp,4
4:      jz A
```

The call will never return and if the code part that is next to "jz A" line will be recognized as belonging to their functions. If you place a basic block from other functions there, it will totally break the function 's block ownership recognition.

Sharing Basic Blocks

Sharing basic blocks by multiple functions can break CFG and CG. Sometimes compiler perform optimization sharing same basic blocks between multiple functions. The basic blocks shared must be contain returning basic block. So it's not easy to find this kind of basic blocks that is same for any given functions pair.

Use multiple heads for a function

If you make multiple entry point to a function, you can effectively confuse disassembler from recognizing real function entry point. To make the fake function head(entry point) more realistic, you can make a fake basic blocks that is calling the fake head. This will convince the disassembler to believe that the fake head is also valid one. So this will prevent proper function-block matching and will affect binary diffing using graph isomorphism.

Anti Binary Diffing Tool: Hondon

We will implemented in-house tool called "Hondon"(in Korean means chaos) to show that it's possible to defeat major commercial and freeware binary diffing tools. It can be used to obfuscated binaries so that the patched points are buried under other meaningless differences. The obfuscated code parts should not affect the performance of the module nor it should not make the debugging difficult. As to say, it should not have any serious side effects other than preventing binary diffing. It will just make the patched code parts invisible and buried among obfuscated fake patched parts.
Hondon is implementing most of the methods presented here. And you can run it as a IDA plugin. It will be dependent on IDA's disassembling ability. That's better than a tool has full blown disassembler because you can manually correct what IDA's disassembler mis interpreted. If we provided our own in-house disassembler, it would make the feedback process almost impossible. So you run this as IDA's plugin, it will rewrite the target binary as binary-diffing-proof binary.

Conclusion

So we looked into the history and current state of binary diffing art. The 1-day exploit threat is real and currently happening every patch days. Sometimes some people diff different version of product, for example, IE6 and IE7 binaries and find some vulnerabilities fixed silently. And the points where vulnerabilities are fixed are good starting points for further vulnerabilities hunting. Bugs tend to aggregate and many times around where bugs were found, another bugs reside. Or some fixes are incomplete and someone can find those facts and can exploit the conditions.

So as the attacking technology improves, the protection techniques need to evolve accordingly. Some major vendors are reluctant to use severe form of anti debugging, because it can break things. So they need some lightweight, non-aggressive and effective way for defeat binary differs. We presented "Hondon" which exploit binary-differs weak points. As anti-binary diffing technology evolves, binary differ will evolve, too. They will process the broken CFGs and also do strong form of instruction reordering or use other techniques. So the story will continue...

Acknowledgements

Without patience and help from my wife and kids, this paper would never been possible. They supported me without knowing they are doing that and their presence itself was great help to me. Also I thank to Byoungyoung Lee for his advice, help and all the times spent listening to me and talking on the messenger.

References

[BMAT] Z. Wang, K. Pierce and S. McFarling, BMAT - A Binary Matching Tool for Stale Profile Propagation, The Journal of Instruction-Level Parallelism (JILP), Vol. 2, May 2000.

[ARE] Automated Reverse Engineering

[TODD] Comparing binaries with graph isomorphism((http://web.archive.org/web/20061026170045/www.bindview.com/Services/Razor/Papers/2004/comparing_binaries.cfm)

[SCEO] Structural Comparison of Executable Objects

[IDACompare] IDACompare(http://labs.idefense.com/files/labs/releases/previews/IDACompare/)

[PatchDiff2] PatchDiff2(http://cgi.tenablesecurity.com/tenable/patchdiff.php)

[BinDiff] http://www.zynamics.com/bindiff.html

[DG2] http://www.darungrim.org