二进制比较与反二进制比较
——对抗 1day 和 0day

作者：

Jeong Wook Oh，韩国人，中文名：吴政昱，就职于 Eeye 美国，超过十年的工作经验，韩国 bugtruck 网络安全邮件列表运营者，Eeye 产品 Blink 的开发者，主要负责攻击过滤分析。其分析引擎使得 Blink 与传统的基于 IPS 的签名相比，提高了识别的准确率。

陈琛，昵称 Eong， 国内安全研究人员，就职于启明星辰，Ph4ntOm 成员，超过七年网络安全研究经验，主要研究方向为 Linux/Mac 安全研究、基于二进制的漏洞挖掘、Fuzz 技术、手持终端安全等等。

简介

起因

众所周知，安全补丁是用来修复安全漏洞的并保护最终用户不受安全威胁的。但是公开补丁其实也会带来安全风险。这就是我们常说的“1day”。在安全补丁公开的数小时内，就会有人用二进制比较技术来寻找安全补丁所修补的漏洞。这很讽刺，但是这却是事实。

二进制比较技术最早应用于微软的补丁。不像其它厂商，微软会较为有规律地发布安全补丁，而且补丁通常都只修补一个应用程序的一个或几个漏洞。所以补丁较为容易分析。

我们在 2006 年的时候开发了一套叫做"eEye Binary Diffing Suites"的工具包，世界上有很多安全研究人员都使用过这个工具。它是免费而且开源的，很容易被用于“1day”挖掘。实际上，攻击者有这些工具并在补丁公布和用户打补丁这个时间差上进行攻击。这个过程有可能持续几分钟，也可能持续几天。

从我们对过去几年的监测中可以知道，所有重要的安全补丁都被二进制比较过，不管是自动化的还是手动的。快的时候几十分钟就有人跳出来说他已经完成补丁的分析了。大概一天的时间，就会有可利用的漏洞被确认并写出可工作的 exploit。现在二进制比较工具很容易获取，所以在这个补丁还没完全部署到每一台机器上的时间，用户的确正在遭受 1-day 的攻击。

另外，通过二进制比较，攻击者可能找到一些厂商没有发现、没有公开或者没有正确修补的漏洞，从而引发更为危险的 0day 攻击。

解决方案

实际上，使补丁分析变得更困难能够缓解这一问题并使得用户能有更多的时间来部署补丁。尽管微软并不会对他们的产品进行代码混淆，他们依然可以使用一些技巧和策略来增加二进制比较的难度。我们会讨论一些使得二进制比较更为困难的方法并且会演示一个用来混淆二进制文件的工具，这就是反二进制比较。

二进制比较

历史

从最早的 BMAT 那篇文章至今已经有十年之久，直到最近两年二进制比较才被人们广泛使用。除了昂贵的“bindiff”，我们目前有 2～3 个免费或者开源的工具可以用于补丁分析。这里我们大概介绍一下历来有哪些二进制比较的理论和工具。

BMAT(1999)

这种方法严重地依赖于符号文件，它主要是用于微软的二进制文件，因为这些文件都是有符号文件下载的。在基于名称匹配的基础上，当所有的函数都已经被匹配以后，它会对函数内的块进行基于哈希的比较。它从汇编指令生成 64bit 的哈希值，并对指令和操作数进行了抽象分级。这篇文章并没有集中于安全补丁的分析。它主要是介绍如何通过校检值来比较块。如果代码优化参数被改变，那么生成的块也会被改变，但是一般厂商并不会改变这些优化参数，所以块都会保持原来的指令顺序不变。所以对块计算校检值并用于比较比比较整个所有的指令要好一些。文章中还提到了五种不同级别的块校检值计算方法。他们将这五种级别称成为“matching fuzziness”级别。级别越低，用于计算校检值的信息就越多，0 级基本上将所有信息都用于计算校检值，包括寄存器、块地址、操作数、opcode 指令等等，而第五级只使用 opcode 指令生成校检值。

这篇文章还阐述了如何基于块比较来获得 CFG，这种方法成为目前所有二进制比较所采用的基本方法。

自动化逆向引擎(2004)

Halvar 在 Blackhat 2004 就这个议题进行了演讲。他提出使用函数的指纹（fingerprints）来进行比较。而且他基于这个独特而简单的思路开发了著名的 bindiff。Bindiff 使用节点的个数、边界数以及调用数作为特征，生成一个函数的签名，然后使用它进行比较。该工具还使用了基于函数 CG（call graph）的同构比较。

基于同构分析的二进制比较 (2004)

Todd Sabin 提出了基于同构分析的二进制比较方法。它基于指令图形的同形匹配。它不再对函数进行分割，而是直接将整个函数结构进行同构匹配。有意思的是它比较指令而不是基本块。他声称性能还不错可以用于使用，但是从未公布工具。

可执行文件的比较(2004)

Halvar 基于 2004 年演讲的改进版本。

基于图像的可执行文件比较(2005)

Halvar 改进了他 2004 的理论，基本方法是一致的。 问题是该算法严重依赖于二进制文件的 CFG 生成，就如他所说的，从二进制文件生成 CFG 不是件容易的事。生成 CFG 本身就是一个富有挑战性的工作。任何矛盾的 CFG 都会导致整个分析失败。

你可以忽略 CFG 识别错误，但是这就意味着许多错误的结果。而且你可能会错过非常重要的部分。而且这种算法无法分析没有结构性变化的补丁，例如仅仅是改变了某个常量或者换了寄存器。

工具

我们将要回顾一下至今为止所有的二进制比较工具（包括商业和免费的）我们同样会比较他们所使用的算法以及效率。

Sabre Security bindiff(2004)

Halvar 基于他的图像和指纹理论所开发的商业二进制比较工具。

IDACompare(2005)

根据这个工具的描述，它应该是基于签名扫描的。我们并没有测试过这个工具，它主要是用于恶意软件分析的。根据网站页面的描述，它使用了 mdb 并且适用于 500k 左右的文件。所以不太适合安全补丁分析，因为补丁很容易超过这个数值。

eEye Binary Diffing Suite(2006)

Jeong Wook Oh 在 2006 年夏天开发了一套开源的二进制比较工具。它主要用户微软的补丁分析。那个时候，微软还不给其它厂商提供补丁的相关信息。所以通过工具分析补丁是唯一的途径。

"DarunGrim" 是当时包括在内的一个主要工具。这个工具使用 sqlite 数据库来存储分析结果。该工具的算法经过逐步改进也成为了 DarunGrim2 的算法。这个工具是用 python 写的，所以性能和兼容性上存在一定问题。不过这些都在 DarunGrim2 上得到了解决。

Patchdiff2(2008)

从工具描述看来，它主要是用于安全补丁分析和恶意软件研究。这是 Tenable Security 免费发布的工具，不开源。根据相关文档，它应该使用了基于 CG 的算法，和 Halvar 的方法类似。

DarunGrim2

这是 eEye Binary Diffing Suite 的 C++改进版本. 主要的区别在于用 C++完全重写了并且修正了很多性能问题。

DarunGrim2

算法

我们将结合目前所有被用于二进制比较的算法和技巧。每种算法都有各自的优缺点，简单的算法在验证某些漏洞时也会有非常好的效果。该引擎基于指纹哈希对比来快速稳定地匹配过程。之前主要的二进制比较工具都集中于图形结构分析和同构比较。但是基于两个图像的精确比较也是有缺点的，例如过分依赖于反编译器生成的 CFG。如果 CFG 不完整，那么整个过程都会成为未匹配的。所以类似间接调用是不能使用同构分析的。名称匹配也是被广泛使用的，但是如果没有符号文件，那就完全无法匹配。

指纹哈希能够克服这些局限性并很好地改进分析结果。与此同时，它还有很不错的性能。我们在程序中使用了这种方式，你可以在后面的例子中看到区别。

但是指纹哈希匹配并不是我们所使用的唯一方式。我们也采用了其他算法作为辅助，例如

同构分析和符号名称匹配。

符号名称匹配

DarunGrim2 也像其他程序一样使用符号名称匹配，这是二进制过程匹配的最基本的方法。符号名称可能是导出的名称也可能是符号文件提供的名称。一般来说符号文件是不公开提供的，但是微软一直为所有补丁提供符号文件。这对于二进制比较和漏洞挖掘有非常重要的作用。

指纹哈希映射

什么是指纹?

我们采用指纹哈希的方法作为 DarunGrim2 的主要算法. 这种方法非常简单，不需要对二进制文件非常精确地分析就可以实现。指纹哈希采用指令序列作为特征值。

一般说来，指纹可能存在多重含义。这里指纹是用于表示一个基本块的数据。对于每一个基本块，DarunGrim2 都从中读取所有的字节并作为 key 存储在哈希表中。我们可以称之为块的指纹。指纹的大小随着字节的不同而不同。有很多其他方法也可以生成指纹。

指纹匹配可以非常有效地匹配基本块。基本块是二进制比较的最基本的分析元素，一个基本块可能含有多个引用。DarunGrim2 为比较的两个二进制文件分别建立指纹哈希表，然后将每一项都进行匹配。不像 flirt 那种传统的函数指纹，DarunGrim2 用得是抽象的字节作为基本块的指纹。所以即使函数中有些基本块被修改了，根据其它匹配的块依然可以正确地将函数匹配。为了快速地匹配海量的指纹哈希，我们把生成的指纹串存在哈希表中然后进行匹配。有很多方式可以生成基本块的指纹，你也可以尝试其他的指纹生成方式，匹配的时候略有不同。

从基本块生成指纹

最简单的生成指纹的方式就是使用指令和操作数。因为基本块的指纹生成是至关重要的部分，所以还有一些问题需要考虑，例如编译和链接选项产生的差异。我们会忽略内存和立即数，因为改变源代码的时候很容易导致这种差异。我们也会选择性地忽略一些寄存器差异。

使用 IDA

DarunGrim2 会从 IDA 读取二进制分析结果。所以指纹的分析将会依赖于 IDA 的逆向结果。IDA 能提供很好的信息，所以 DarunGrim2 只是从 IDA 中读取 insn_t 结构来获取信息。

克服顺序依赖的缺陷

连续字串的指纹最大的缺陷就是顺序依赖。实际上这个问题普遍存在于各种算法当中。所以为了解决这个问题我们提供了选项，在生成指纹之前对指令顺序进行整理。这个过程只要判断指令的互相依赖关系即可。然后将指令按照升序排列。互相依赖的指令必须维持他们原有的顺序。优化指令序列对于编译器指令顺序优化过后的代码尤为重要。

减少哈希碰撞

一个二进制文件中会有不少短的基本块，这些基本块很容易重复。这就会导致哈希重复，我们可以直接抛弃这些重复的基本块。但是我们有更好的处理方法，就是将一个基本块和连接它的下一个基本块进行关联。这样重复的哈希值就很少了。

检测函数匹配对

在基于哈希表的匹配完成以后，我们必须对函数中匹配的基本块进行统计。如果一个过程匹配多个过程，那我们必须进行选择。我们采用的方法很简单，就是利用匹配的基本块计数，数值最高的作为匹配项，如果有多个同样数值的，那么会随机选择一项。一旦某一项被选择，那么未被选择的块中的基本块关联将被取消。

其实这个抉择过程很大程度上依赖于哈希的配对。在实际使用中，这样的哈希配对方法有很好的性能和不错的效果。

函数中的匹配块

如果一个过程被选择配对了。那么其中的块也会重新计算哈希表进行配对。这就是函数中的二次匹配。和之前的方法一样，通过关联上下块，重复的哈希值依然会很少。

哈希表是什么样的?

这是一个生成的哈希表数据库。

Table: OneLocationInfo

New Record | Delete Rec

	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	cc8f0102cc120202cc7a01020102ccd20102010
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

图 1：哈希表数据库查询

基于结构的分析

基于结构的分析：基本块同构

在进行完函数配对并将其中的块一一配对以后，我们将会进行基于结构的分析。当然，这不是 Halvar 的那种方法。我们的方法是更为可靠的。因为我们使用基本块作为图形节点，这和 Todd

的同构算法不同。这种方法和 BMAT 的方法比较类似。从已经匹配的节点，去匹配他们的子节点。

流程追踪

接下来我们必须要考虑流程追踪，Todd 已经在他的文章中阐述过这个问题，DarunGrim2 只是在他的方法上进行了改进。

这是一段 C 风格的代码，用于流程追踪。

```
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;

        }

}
```

计算匹配指数

当进行程序化的结构匹配时，我们必须对两个基本块进行判断，看他们是否一样或者相似。我们依然采用哈希来实现这个操作。因为指纹哈希依然是当前最可靠的值。哈希值是作为字节序列存储在内存中的，所以很容易转换成 ascii，转换以后就可以很方便地计算两个基本块的相似程度。这部分会比较花时间而且会有比较多的错误计算。

在这个过程完成以后，我们将会将没有被匹配的代码块重新匹配，直到没有任何匹配项产生为止。

实际上，在用于微软的补丁比较时，指纹匹配和结构化匹配已经完全足够了。

真实案例

在二进制比较过程中会存在一些问题，因为不同厂商往往使用不同的编译器和优化参数。下面是我们在实际应用中遇到的一些问题。

分割块

作为优化的一部分，有些块会被分割开存储在不同位置，我们可以这样定义分割块：

"CFG 中当一个块只有一个字块，而这个字块只有一个父块的时候，他们就是分割块"

这些分割块会使 CFG 被破坏，匹配过程也会被影响。这是个真实案例。左边的 757AC02B 和右边的 7CB411B5+7CB411BA 其实是一个块。但是却被标记为红色了（不匹配），其实他们是匹配的。

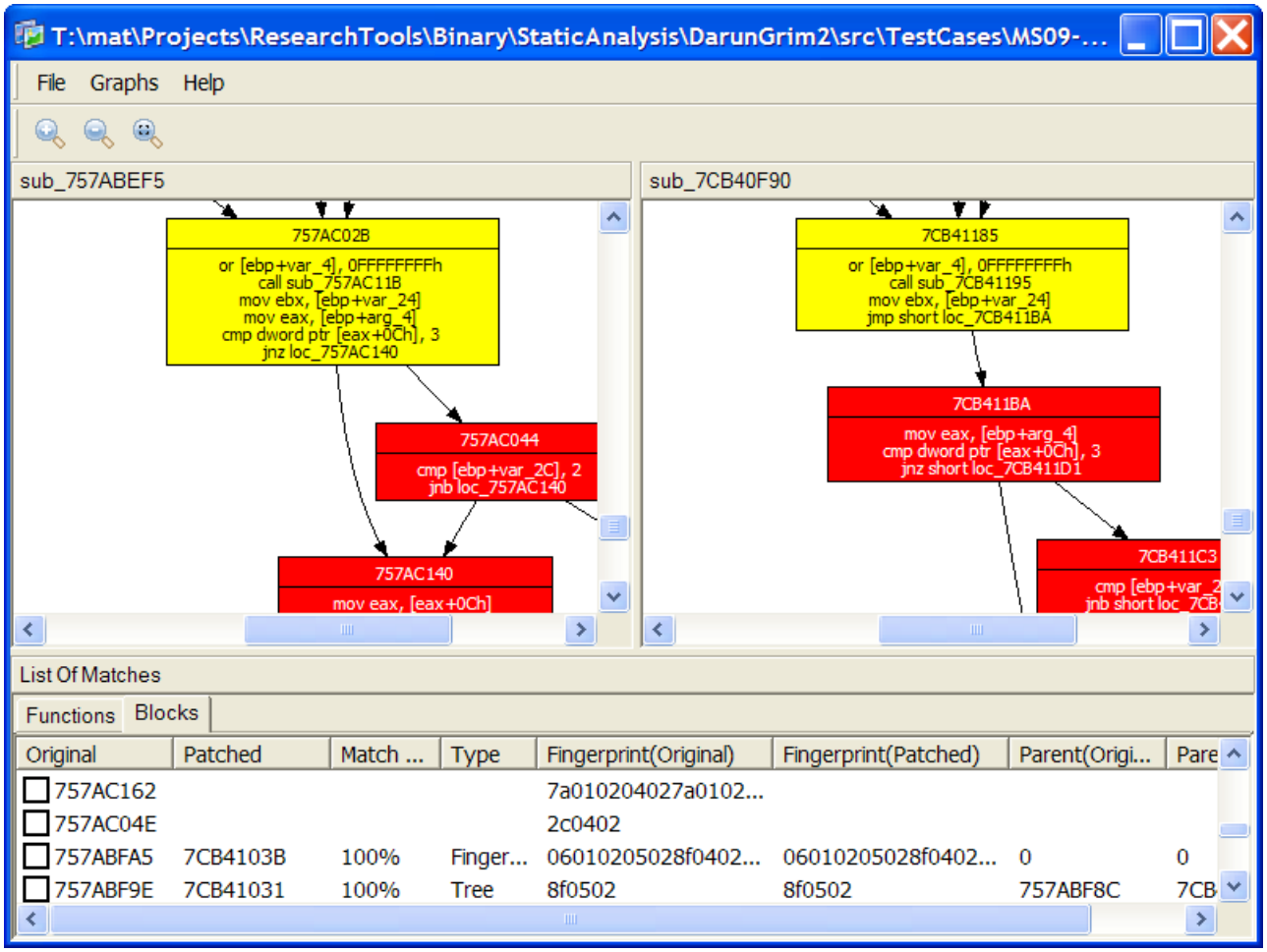


图 2: 分割块(L:757AC02B=R:7CB41185+R:7CB411BA)

DarunGrim2 能够识别分割块并将它们合并在一起。在合并以后，看起来会像是下面这样子。

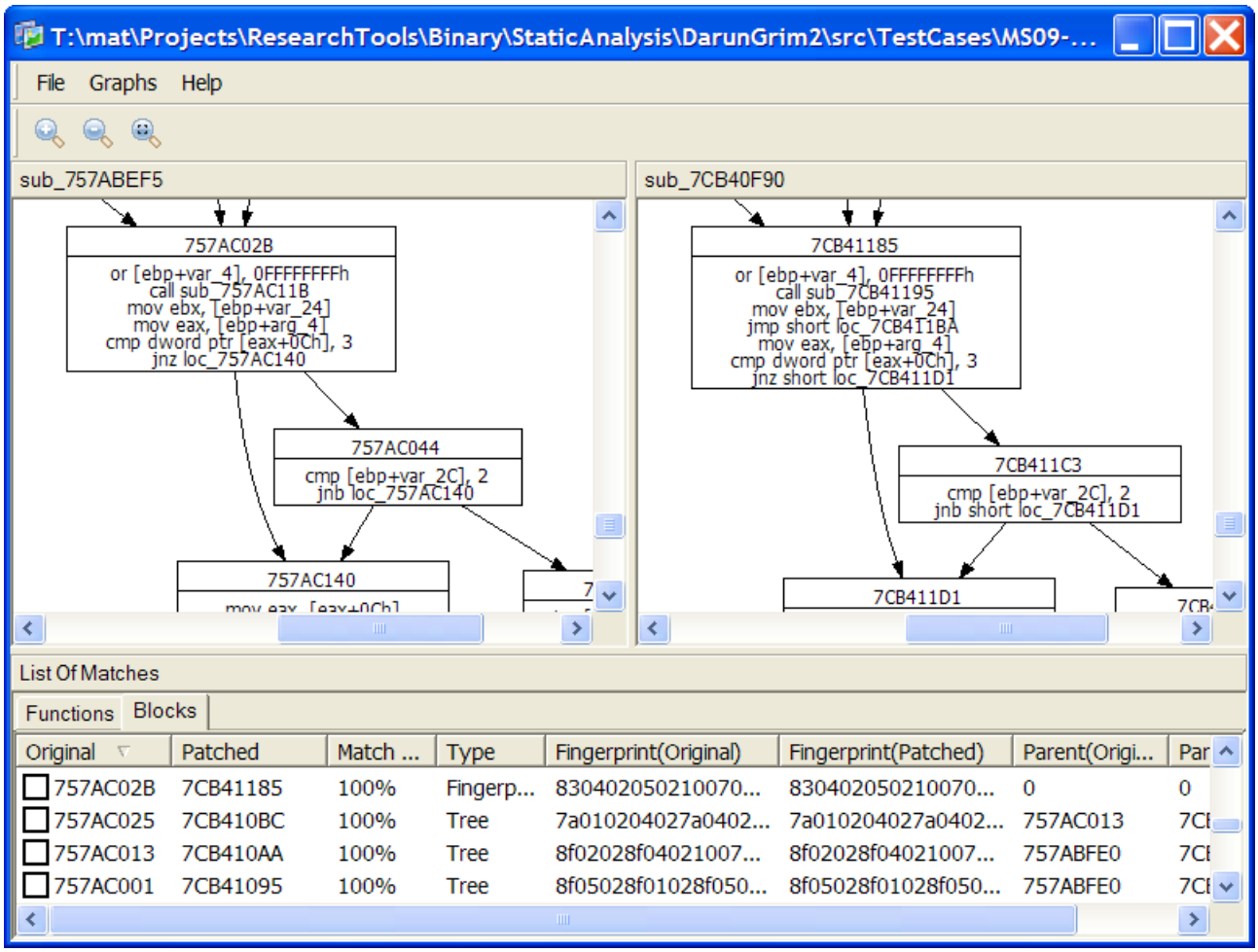


图 3: 现在右边的分割块合并了(7CB41185)

现在左右两边能匹配了。

Hot Patching

微软的某些二进制文件偶尔会使用 hot patch，那么在函数的开始会存在一些 hot patch 的指令这里是一个 hot patch 的案例。

```
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
```

红色部分是 hot patch 的指令，一般都是"mov eax,eax" ，看起来没有任何意义，但是却为 hot patch 保留了空间。

有时候原来的二进制没有 hot patch 指令，但是补丁文件有。这会导致这个基本块像是被改动过。所以 DarunGrim2 会忽略 hot patch 指令。当一个指令是“mov”并且在操作同一个寄存器时，它就会被忽略。

多个函数包含同一个基本块

一般说来一个基本块只属于一个函数。但是由于优化的原因，有时候一个基本块可能会属于多个函数。这是一个基本块在 41fbc2 位置，来自于 Windows kernel。但是 IDA 却认为它属于 "MmCopyToCachedPage" 函数。

```
.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
```

从代码流程分析来看，我们可以看到这个块属于多个函数。

```
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

从上面的分析结果可以看出。黄色的 41fbc2 同时出现在三个函数中，他们分别是 "MmUnmapViewInSystemCache", "MmCopyToCachedPage" 和 "MiRemoveMappedPtes"。这种情况可以在二进制比较的分析过程中避免。IDA 只支持一个基本块匹配一个行数。DarunGrim2 通过特定的 CFG 分析使得一个块能属于多个函数，从而解决了这个问题。

指令顺序优化

在实际使用中，指令顺序优化其实用得不多，尤其是微软的补丁，我们并没有遇到需要指令顺序优化的例子。但是在 ARM 的二进制文件中，几乎每次都用到了指令顺序优化。图 4 是 iphone2.2 和 iphone3.3 的固件比较。两边的黄色块毫无疑问是匹配的，但是却标记为黄色（部分匹配）。

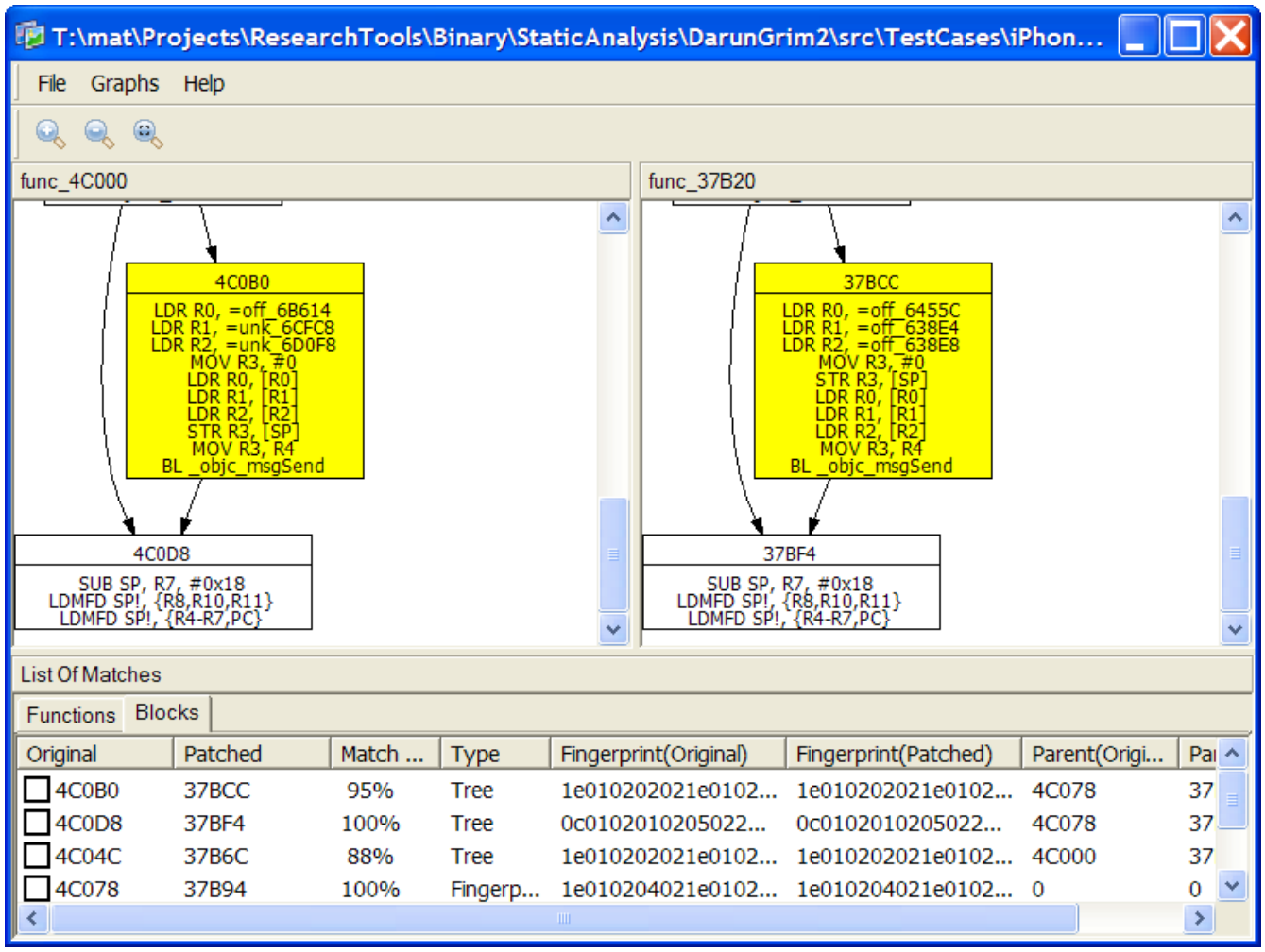


图 4: ARM 架构中的指令顺序优化

我们看看两边到底有什么差别

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     R6, R2
STR     R3, [SP,#0x20+var_1C]
BL      _objc_msgSendSuper2
SUBS    R5, R0, #0
BEQ     loc_32FF9B84
```

```
MOV     R0, SP
LDR     R1, [PC,R1]     ; "initWithPath:"
MOV     R6, R2
STR     R3, [SP,#0x20+var_1C]
BL      _objc_msgSendSuper2
SUBS    R5, R0, #0
BEQ     loc_33328F08
```

表 1: 原始的反编译代码

这两个基本块其实是一致的。唯一的差别就是指令顺序略有不同。

为了解决这个问题，我们采用了分组变量跟踪和排序的方法。我们可以跟踪每一个寄存器或者变量的变化。例如图 2 所示。DarunGrim2 现在支持寄存器、可替换变量和调用参数跟踪。

当使用变量跟踪时，我们可以把每一个节点都分组。于是我们得到很多独立的组，组内的指令顺序是可以任意替换而不会影响的。我们对每个组进行哈希计算。我们依然护绿立即数和内存引用，因为每次编译都会产生不同的值。当这些哈希值计算完以后，我们会将这些块重新排序。

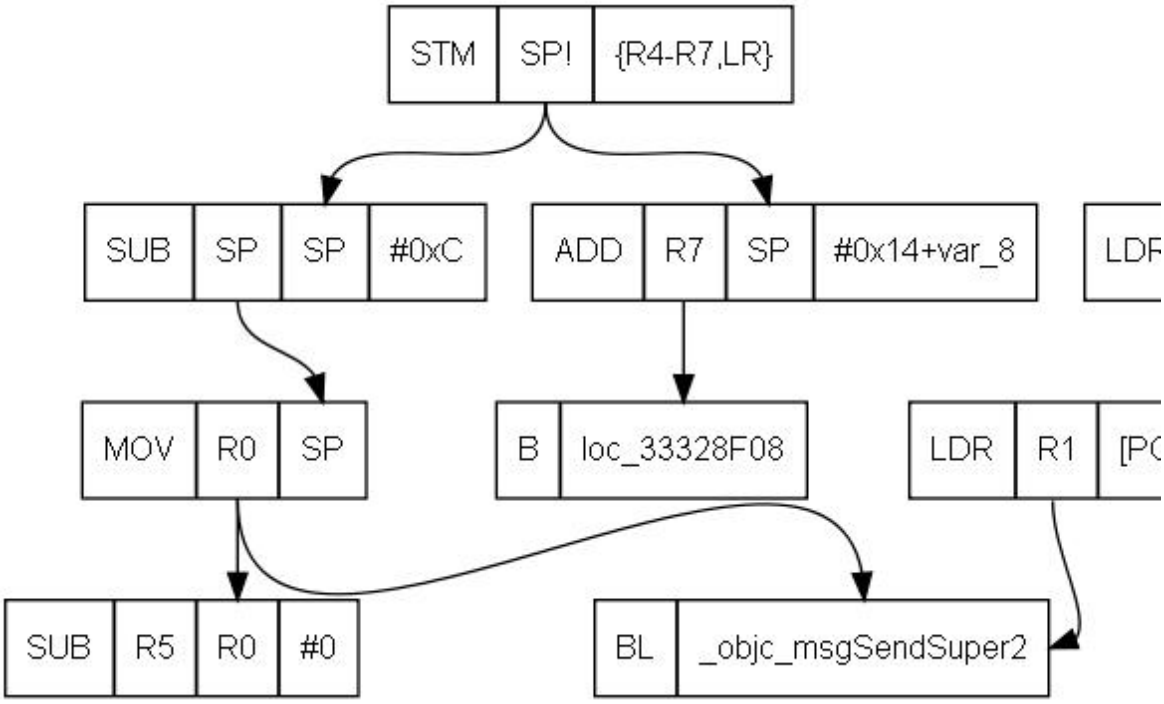


图 5: 寄存器，可替换变量和调用参数跟踪

"表 2: 分组并排序以后" 是排序以后的结果。现在两边是基本一致的了。这种优化排序的方法需要消耗更多的 CPU 进行计算。所以有需要时才开启。

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

表 2: 分组并排序以后

范例

我们将会演示几个微软的补丁分析来说明利用 DarunGrim2 来分析的便利性。我们将会举三到四个严重级别的例子。

微软补丁

为什么微软补丁是容易分析的?

微软每个月的第二个周二都会发布安全补丁。所以这些补丁只包含安全漏洞的修复代码。一般来说是不会有性能增强之类的代码出现的。性能增强一般都随着 SP 发布，而不是每个月的安全补丁。所以二进制比较的时候不用担心会有其他代码干扰。

微软还提供符号。一般系统 DLL 和驱动以及内核都有符号，但是其他产品不一定有，例如 IIS 和 Active Directory 以及 Office。但是大部分补丁都属于前者，都是有符号的，所以分析的时候会比较节约时间。

微软从来不做任何代码混淆，除了代码优化。实际上如果他们做代码混淆的话，可能会导致更多的问题产生，包括调试上的不方便。所以微软以后也应该不会做代码混淆的。

这些使得微软的补丁分析非常容易，需要分析的代码比较而言也很小。

获取补丁

要收集微软的补丁非常容易，你只需要去访问补丁页面就行了。例如 MS08-067，你只要去访问 http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx 就可以下载到符合你系统的补丁。然后你可以用/x 选项释放补丁中的文件。你可以在你机器上找到老的文件，然后用 IDA 对他们进行分析比较就行。整个过程不超过 30 分钟。

臭名昭著的 MS08-067(Conficker 蠕虫)

MS08-067 是一个非常有意思的补丁。它修补了 netapi32.dll 中的栈溢出。而且这个栈溢出可以通过匿名管道来触发，这就意味着这个漏洞可以远程利用。这个漏洞被 Conficker 蠕虫利用并在互联网上大规模传播。

从二进制比较的角度讲，MS08-067 是个很不错的对象。只需要几分钟就能找到发生改变的函数。

实际上就一个函数发生改变

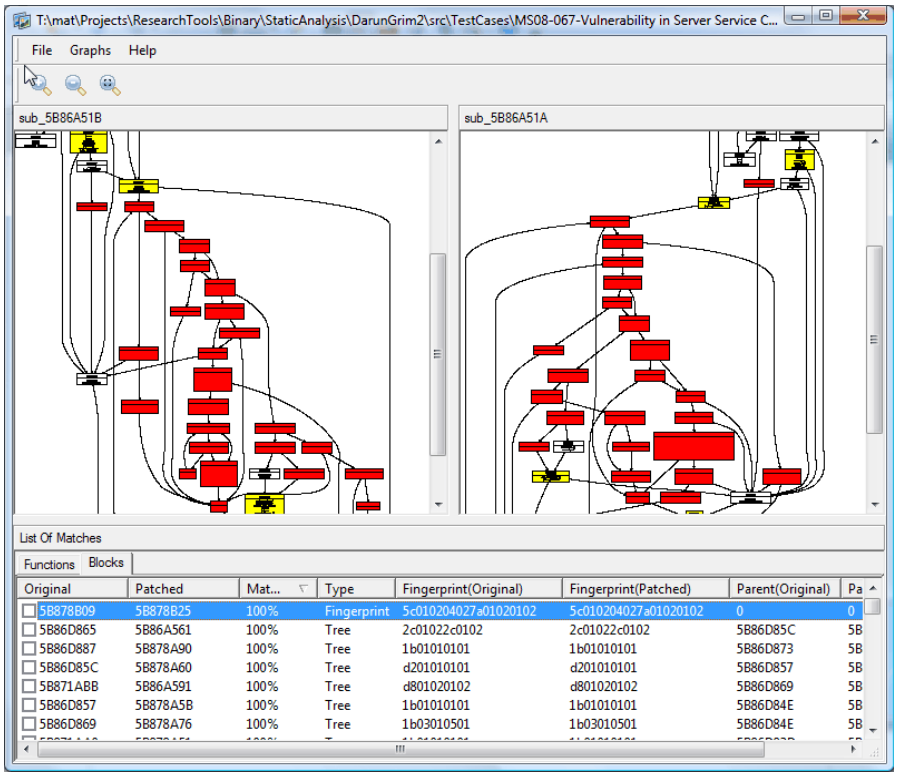


图 6: 左右两个函数几乎完全不同

当然，如果你需要深入研究这个漏洞，你需要花更多的时间。不过这不属于二进制比较的内容了。二进制比较只是一个开始，从图 6 可以看到，修改的代码是非常多的。要利用这个漏洞进行攻击需要进一步深入的分析。

MS08-063: DarunGrim2 vs bindiff

DarunGrim2 的分析结果会显示以下的函数被改变。你可以一个一个查看。

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%

图 7: 改变的函数

"_SrvIssueQueryDirectoryRequest@32" 是漏洞所在的函数。如图 8 所示。

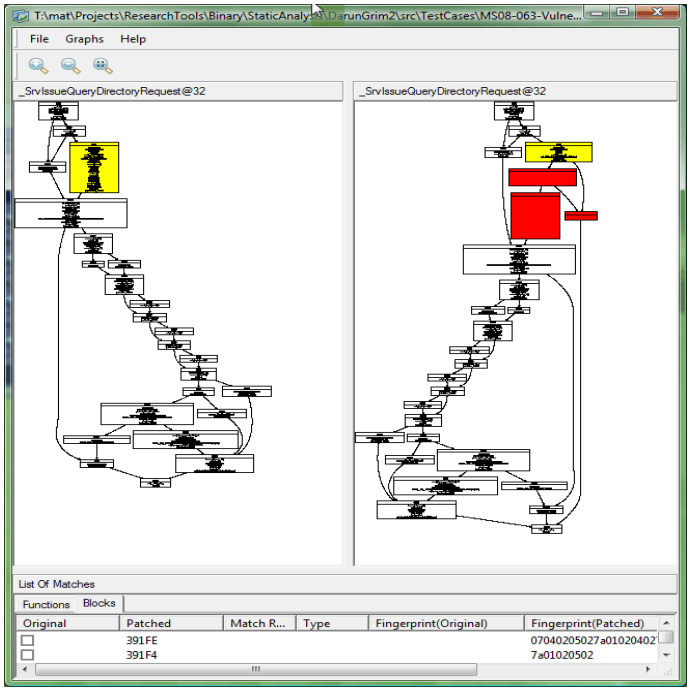


图 8: 函数(_SrvIssueQueryDirectoryRequest@32)

详细的修改如下：

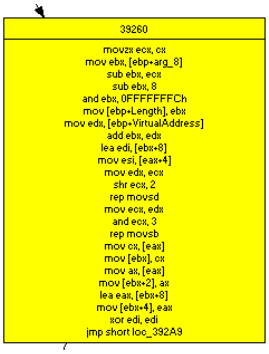


图 9: 补丁前

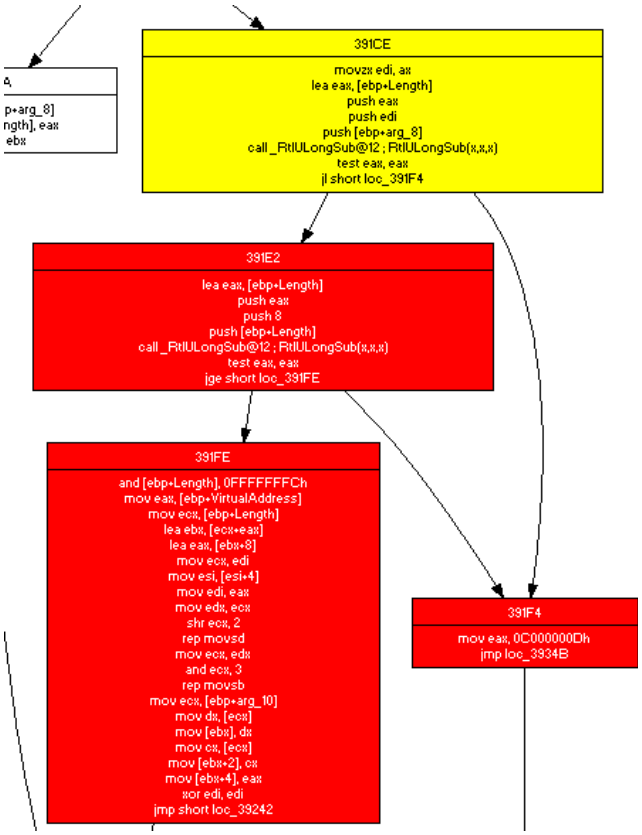


图 10: 补丁后

在图 10 中，红色块是增加的块，用于对数据进行检查。

从补丁中你可以看到，增加了对 RtlUlongSub 函数的调用来对数据进行检查，跟进这个函数，你能看到它具体做了什么。

所以实际上 DarunGrim2 找到了 7 个有修改的函数，而且都是存在修改的。

如果你用 bindiff 的话，你可以看到 bindiff 只是显示三个函数改变。

```
SrvFsdRestartPrepareRawMdlWrite
SrvIssueQueryDirectoryRequest
SrvRestartRawReceive
```

所以就如 Halvar 所说的，基于节点、边界和调用计数的指纹方法会比较少出现误报，但是如果 CFG 结构没有发生变化，那么它会漏掉一些。

这是 bindiff 帮助文档中的截图，"_SrvIssueQueryDirectoryRequest@32"所得到的结果和 DarunGrim2 是一致的。

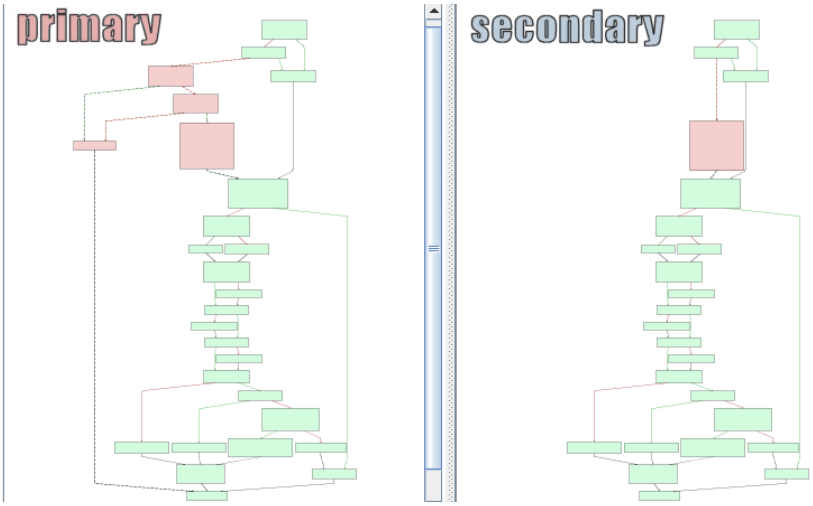


图 11: bindiff 说明书截图

所以实际的比较结果证明 DarunGrim2 是和 bindiff 一样有效的。而且在某些时候，DarunGrim2 能够更加准确地分析出结果。但是这不意味着哪个工具或者算法更出色。两者都各有所长，也各有所短。你需要根据你的需求选择工具。

通过比较这个补丁，下面的函数显示为被改变的。有意思的是根本没有增加任何基本块，只是原来的块被改变了。

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

图 12: 函数比较结果

所以漏洞修补必然是在这个函数之类。下图显示的是未补丁文件。

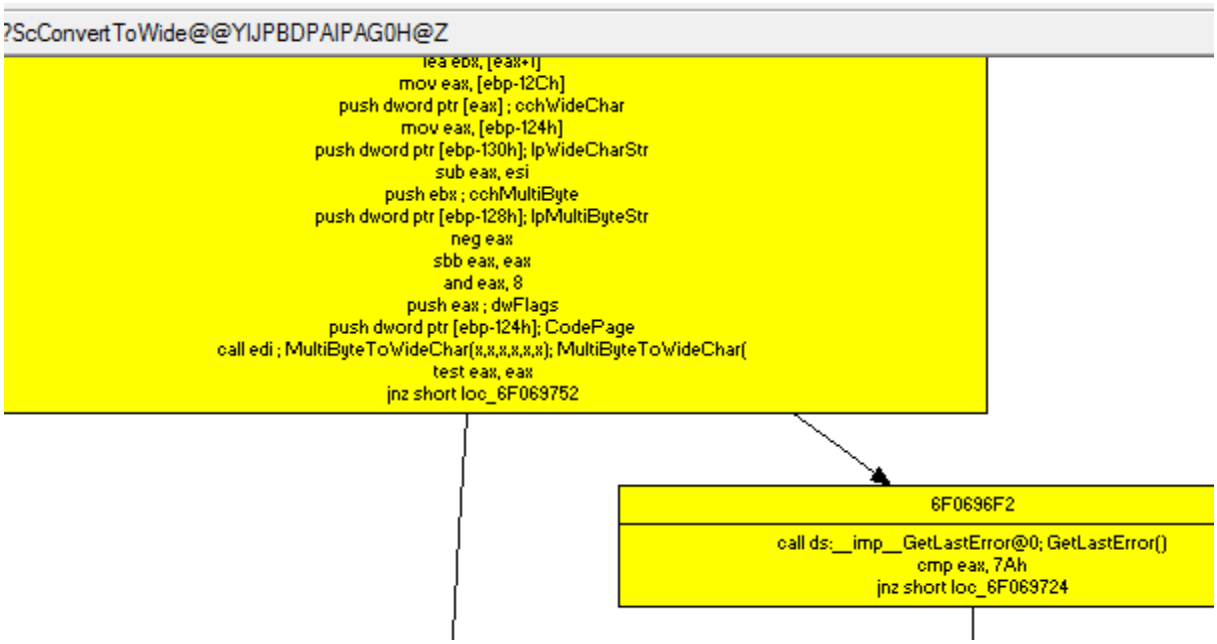


图 13: 未补丁文件

下面是补丁后文件

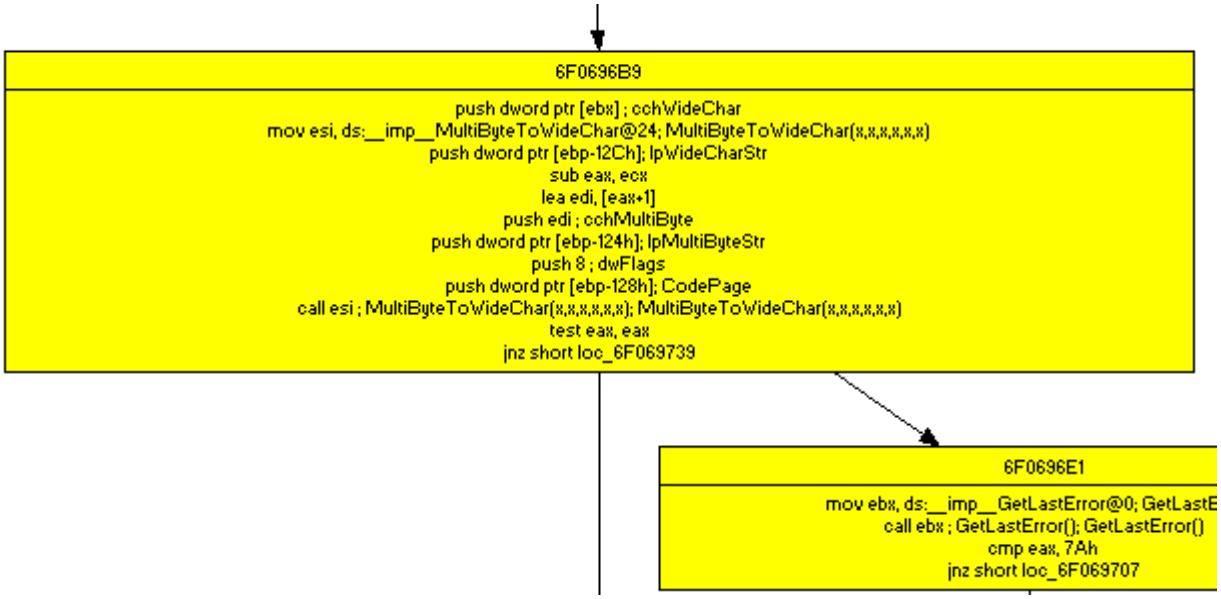


图 14: 补丁后文件

他们的区别不容易被发现，仔细看你会发现"MultiByteToWideChar"的第二个参数发生了变化。未补丁之前它是这样的：

```
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
```

“eax”经过了一系列计算，其中关键点在于 FisUTF8Url，如果返回值为真，那么"esi"和"[ebp-124h]"值相等，eax 为 0.

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

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.

"FIsUTF8Url"并不是一个完整的 UTF8 处理函数，如果传入的参数中包含非法的 UTF8 值，而且没有指定"MB_ERR_INVALID_CHARS"参数，那么函数就不会丢弃非法的 UTF8，从而导致问题。

这是个特殊的漏洞，这个漏洞没有对 CFG 进行改变，所有的改变都是在原有的块上进行的。那么像 bindiff 这样的工具是无法发现的。

非微软补丁

JRE Font Manager Buffer Overflow (Sun Alert 254571)

没有符号也是可以进行二进制比较的。Sun 并不为他们的二进制文件提供任何符号。例如 DarunGrim2，你可以得到以下结果：

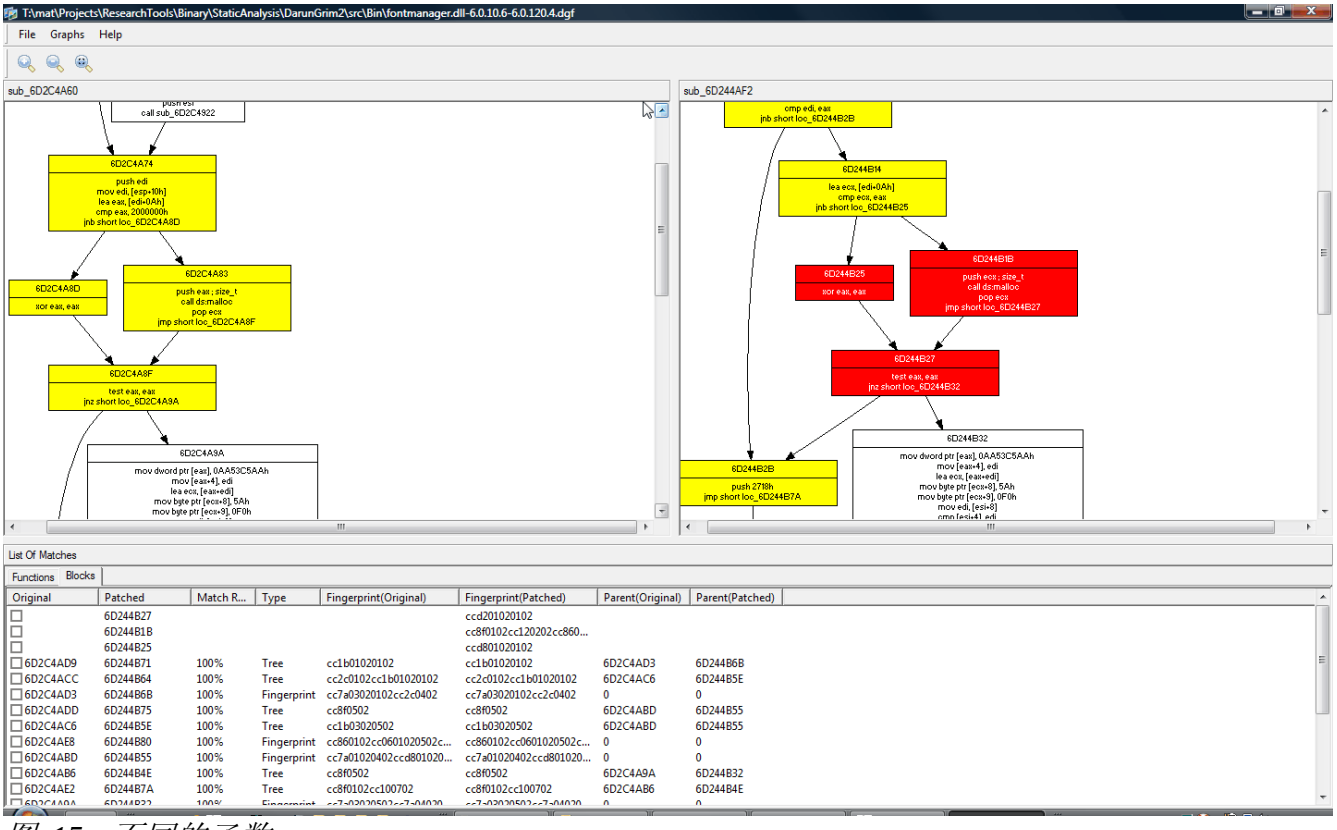


图 15：不同的函数

未补丁前如下：

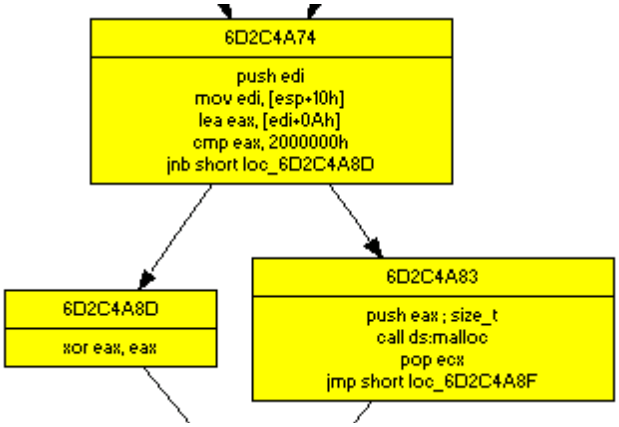


图 16: 未补丁前

补丁后：

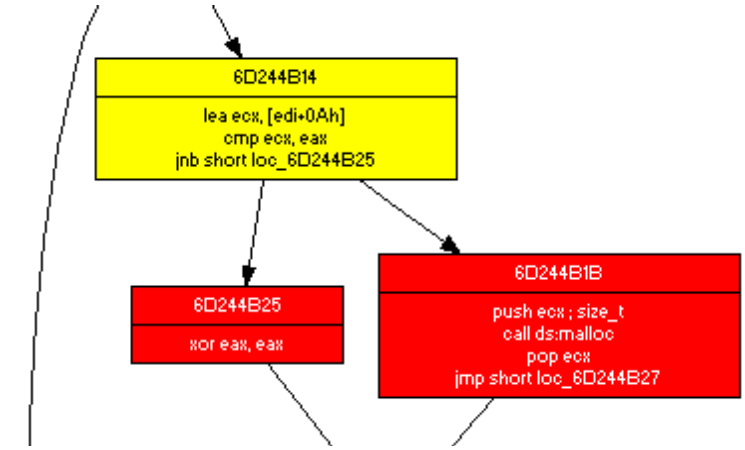


图 17: 补丁后

从 IDA 截取了一下汇编代码。补丁后只是加了两行。

Original			**Patched**		
.text:6D2C4A75	mov	edi, [esp+10h]	.text:6D244B06	push	edi
.text:6D2C4A79	lea	eax, [edi+0Ah]	.text:6D244B07	mov	edi, [esp+10h]
.text:6D2C4A7C	cmp	eax, 2000000h	.text:6D244B0B	mov	eax, 2000000h
.text:6D2C4A81	jnb	short loc_6D2C4A8D	.text:6D244B10	cmp	edi, eax
.text:6D2C4A83	push	eax ; size_t	.text:6D244B12	jnb	short loc_6D244B2B
.text:6D2C4A84	call	ds:malloc	.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

红色部分代码对 edi 进行检查，判断 edi 是否大于 2000000h。

未补丁前只对 ecx 进行是否大于 0x2000000h 的检查，增加的这个检查是为了修补整数溢出。如

果 edi 是 0xffffff，那么 ecx 就是会等于 0x9。那么 ecx 依然可以通过 0x2000000 的那个判断。增加了 cmp 那句以后，就不会产生整数溢出了。

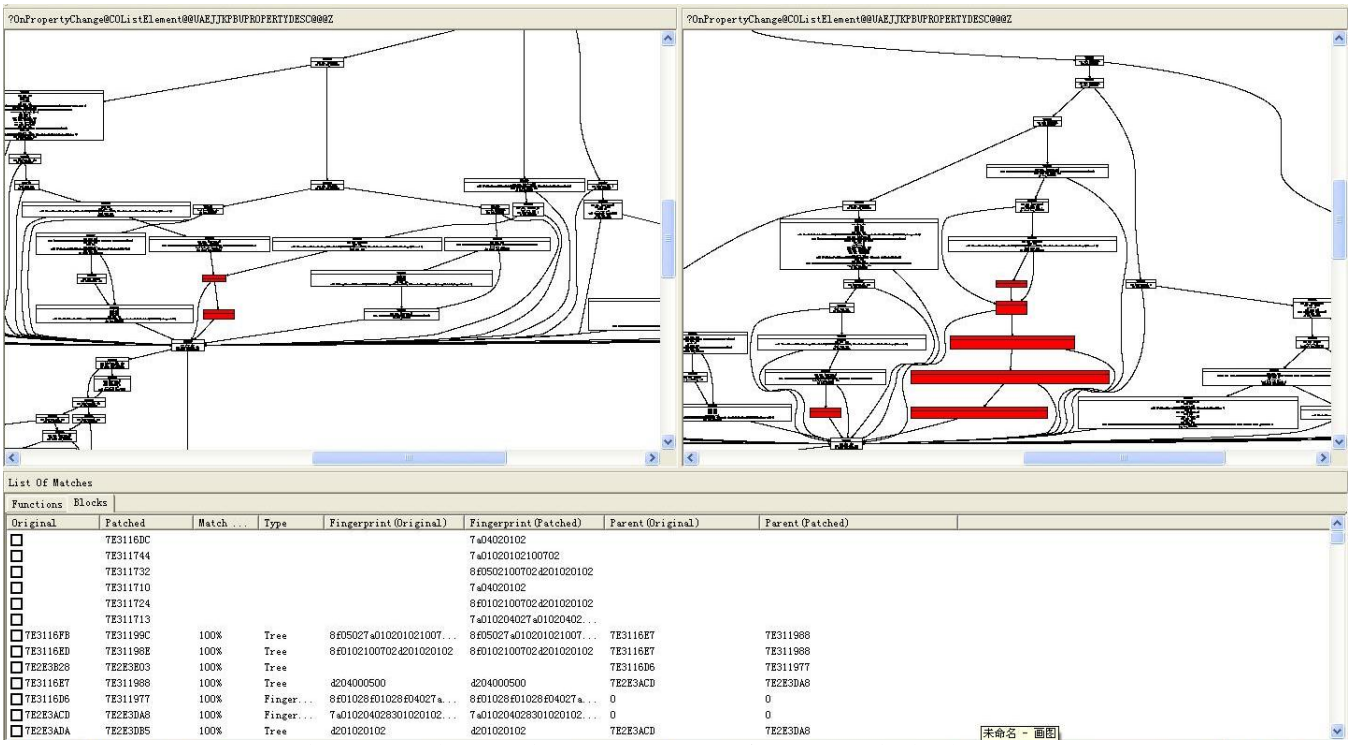
更深入地研究你会发现这个数据是可以被用户控制的。

原始公告在这里(iDefense Security Advisory 03.26.09: Sun Java Runtine Environment (JRE) Type1 Font Parsing Integer Signedness Vulnerability) ，公告是这样描述的：

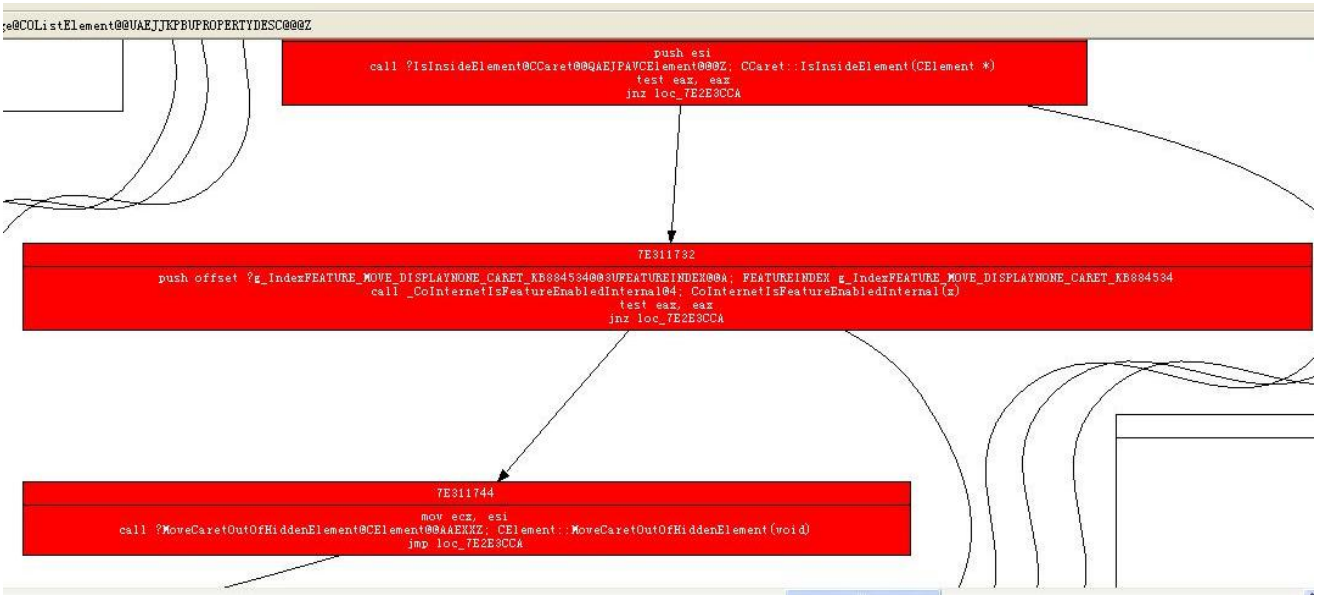
> The vulnerability occurs when parsing glyph description instructions in the font file. When parsing the glyph descriptions, a 16bit signed counter is used as the index to store the next glyph point value. This counter is compared to a 32bit value that represents the maximum size of the heap buffer.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

IE6 VS IE7

有这么一个关于 IE6 和 IE7 的故事，2008 年某个时候，微软按照常规在著名的周二公布了一堆补丁，其中包括一个 IE 补丁 MS08-058，其中包括了 7 个漏洞，没错，微软就是这样让你痛苦的。我和往常一样用二进制比较工具进行了分析，花了一些时间，找到了其中一个漏洞修补点。



左边是补丁前，右边是补丁后，红色部分是存在差异的部分



我们可以明显地看到这里增加了三个函数调用，分别是：

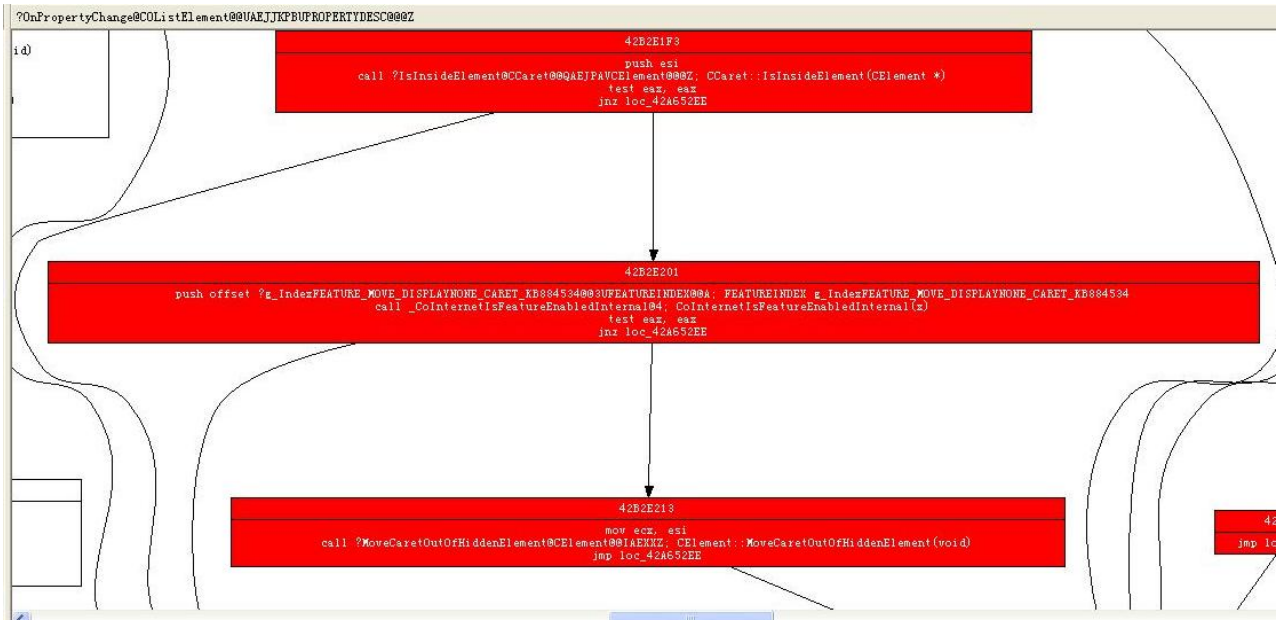
Call ?IsInsideElement@CCaret

Call _CoInternetIsFeatureEnabledInTernal

Call MoveCaretOutOfHiddenElement

我又回头看了看公告，关注到一个细节，我发现 IE7 并不存在问题。这是唯一的一次，IE7 不受影响。

我很好奇为什么，于是我突发奇想地找来了 IE7 的文件（未打 MS08-058 补丁的），并开始比较。



IE7 中该部分代码

有趣的事情发生了，IE6 中的漏洞修补代码和 IE7 几乎完全一致，我当时甚至怀疑是不是有人

在比较这两者找到漏洞的？

于是我也开始比较 IE6 和 IE7 的文件，你可以发现一些类似这样的片段:

IE6:	IE7:
mov eax, [ebp+var_C]	mov eax, [ebp+var_C]
mov ecx, [eax]	cmp eax, ebx
push esi	jz short loc_4299xxxx
push eax	
call dword ptr [ecx+0Ch]	mov ecx, [eax]
	push edi
	push eax
	call dword ptr [ecx+0Ch]

这两段很明显了。当然，这样并不足以证明问题，是否有溢出需要进一步地研究并构造 poc 进行验证。

思考：比较不同版本的程序是可以用来找漏洞的？

在闭源的情况下，新版本的程序往往在旧版程序基础上增加或改变功能。通过比较不同版本的程序，我们有可能发现发生改变的代码的问题。当然这些问题可能不像补丁前后那么明显，但是这已经足够了，它让你知道你该关注哪里，在这样一个过程中，你甚至可能发现 IE6 和 IE7 都存在的问题。或许大家可以深入研究一下去年的 MS08-078。

除了 IE，我们现在还有很多产品可以进行不同版本比较，例如 Adobe Flash 9 和 10，Adobe reader，以及 Windows 系统 DLL 文件。

当然我们还可以比较一些有趣的东西，例如之前提到的 ARM 构架以及 MIPS 构架的代码——我们手头很流行的 iphone 和 psp 不同版本的固件。

反二进制比较

现在我们该考虑如果保护用户远离这种威胁了。既然代码混淆对于开发来说并不怎么好，那么厂商可以考虑使用一些技巧来使得二进制比较更加困难，为用户部署补丁争取更多时间。我们将阐述一些方法。

这些方法和技巧，有些正被使用，有些是理论性质的。

改变符号

改变函数过程符号名称。这对于依赖于名称比较的工具来说是非常有用的。如果补丁的时候对函数名称进行修改，那么符号名称对比将会失效，类似 BMAT 这种完全基于符号名称的工具自然就失效了。

扰乱指令顺序

一些二进制比较工具使用基本块的校检值来进行匹配，校检值对于指令顺序的依赖度很高，如果指令顺序发生变化，那么这些工具将不能正常工作。

扰乱 CFG

类似 Bindiff 的工具都依赖于每个函数的 CFG 签名。如果通过增加假的节点、边界或者调用来改变 CFG，那么可以完全破坏 CFG 签名，使得整个分析过程错误。例如通过增加一个从来都不会被执行的分支，你就可以使得 CFG 发生错误。
如果修改过程的 CFG 或者程序的 CG 而不改变函数，那么有可能使得这类工具无法正确地进行结构分析。这种方法也会影响同构分析。增加一些无意义的块或者指令也可以扰乱函数签名。需要注意的是我们需要将扰乱块放到足够多的函数中去，如果只是集中在被补丁的函数周围，那只会此地无银三百两。

*使用 **proxy** 调用*

破坏 CG
将 call <XXX> 替换成 call <YYY> ，而 YYY 只是一个 proxy 调用。

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

函数 YYY 其实没有任何作用，只是继续调用 XXX。为了避免反编译工具认出这个空函数，你可以在这个函数里面增加一些无用的代码。

调用不返回

实际上这个思路来自于一个真实的案例。这种方法会破坏 DarunGrim2 的分析，也会破坏 CFG。如果你增加一个像下面“call B”那样的调用，而从来不返回的话。

将这一行改成下面的样子：

```
jz A
```

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

proc B 会从 jz A 跳出，从来都不会返回。这样会破坏基本块的分析。

共享基本块

在多个函数中共享基本块同样可以破坏 CFG 和 CG。有些时候编译器在优化代码时会这么做。特别是当这个基本块有返回指令时，将会使得匹配非常困难。

同一个函数使用多个头

如果你为一个函数增加多个入口，那么你可以有效的迷惑反编译工具，使他们认不到真正的入口。要使假的入口更具迷惑性，你可以构造一个基本块去调用哪个假的入口。这样编译器会认为那个假的入口也是一个入口。这样可以破坏韩素匹配，也会影响 bindiff 的图形化同构。

反二进制比较工具: Hondon

我们将会实现一个内部工具叫做"Hondon"(中文的混沌) ，这个工具将可以阻止现有的商业或者免费二进制比较工具。它可以被用来混淆二进制文件以便隐藏修补点。而且这些混淆代码并不会造成性能下降或者调试困难。换句话说，它不会有任何负面影响除了阻止二进制比较。“Hondon”将会应用上面提到的几项技术。你可以当作一个 IDA 插件运行。它会依赖于 IDA 的反汇编，你在 IDA 中运行它，就可以从 IDA 打开的文件得到一个混淆过的二进制文件。

总结

我们回顾了二进制比较的历史，也对现状进行了分析。1day 是真实存在的，甚至天天都在我们身边发生。有时候人们比较不同版本的文件，例如 IE，或者 adobe flash。他们可以发现一些被偷偷修补的漏洞或者 0day。而且通过二进制比较，人们可能从已有的漏洞中学习，并发现一些类似的漏洞，或者发现没有补全的漏洞。

随着攻击手段的不断提高，防御技术也需要随着提高。很多厂商出于自身利益考虑，不愿意使用反调试技术，他们或许可以用上类似 Hondon 一样的轻量级反二进制比较技术，这些技术并没有什么副作用，仅仅是组织攻击者去比较不同版本的文件。或许二进制比较技术会随之得到更大的进步，而故事也会继续上演。

参考文献

[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