

使用32位64位交叉编码混淆来打败静态和动态分析工具

wildsator (/author/wildsator) · 2015/12/09 11:43

Ke Sun(wildsator@gmail.com)

原文:<http://en.wooyun.io/2015/12/08/33.html>



0x00 摘要

混淆是一种能增加二进制分析和逆向工程难度与成本的常用技术。主流的混淆技术都是着眼于使用与目标CPU相同的机器代码，在相同的处理器模式下，隐藏代码并进行控制。本文中引入了一种新的混淆方法，这一方法利用了64Windows系统中的32位/64位交叉模式编码。针对静态和动态分析工具的案例研究证明了这种混淆技术虽然简单，但是十分有效。

0x01 64位Windows中的模式转换

所有的64位Windows操作系统都能向下兼容未修改过的 32 位应用，这是通过WoW64（64位Windows上的32位Windows）子系统实现的。WoW64使用了3个动态链接库（Wow64.dll，Wow64win.dll 和 Wow64cpu.dll），提供了一个兼容层，用作64位内核与32位应用{1}之间的接口，借此，处理器模式就可以在32位和64位之间动态地切换。

在任何时间中，处理器模式都是由当前代码段（CS）描述符中的控制位决定的。图1就是一个段描述符的示意图。L 位（21位）是“64位代码段”控制标记：“值为1则说明这个代码段中的指令会在64位模式下执行，值为0则说明在这个代码段中的指令会在兼容（32位）模式下执行。” {2}

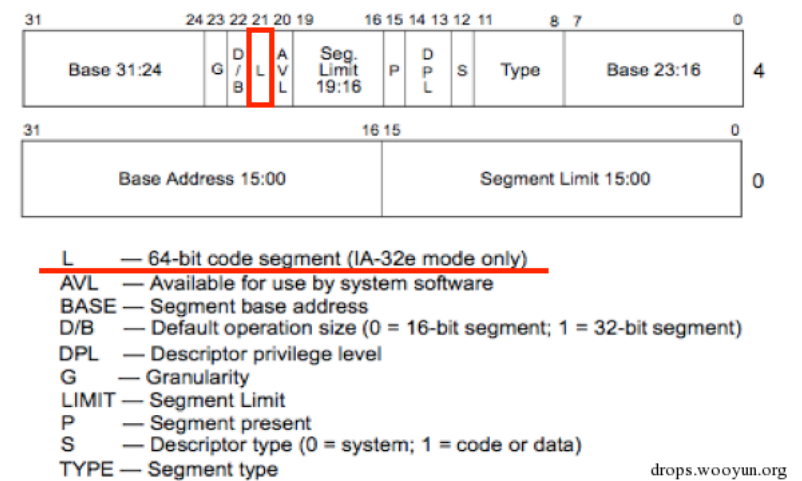


图1-段描述符的示意图 {2}

64位Windows下的任何应用，无论是32位还是64位，在载入到内存中时，都至少有两个代码段。这两个代码段的32位地址空间映射是完全重合的，但是段选择器是不同的：其中一个的选择器是0x23，L标记为0（32位代码段）；另一个的选择器是0x33，L标记为1（64位代码段）。在一个应用的PE（可移植可执行）标头中，“Machine”字段的标签是32位还是64位PE决定了这个应用的默认代码段。

当远分支指令把控制流从一个代码段上转移到另一个有着不同L标记的代码段上时，处理器模式交换就会被触发。例如，如果一个32位应用在64位Windows中执行，默认的CS选择器就会是0x23，处理器会是32位模式。如果，使用了远转移来跳转到一个选择器是0x33的代码段上，那么在执行完这条指令后，处理器模式就会切换到64位，在从32位向64位切换时也是一样的。

有很多远分支指令都可以用于触发模式转换，包括（1）远转移（2）远调用（3）远返回和（4）中断返回。这些指令就像是不同维度的平行世界之间的传送门，在需要时可以通过这些门穿梭于不同的世界。因为这种方便的可控模式交换，混淆技术才有了机会来使用交叉模式编码，这种方法既可以用在32位应用中，也可以用在64位应用上。

唯一需要注意的是64位应用，在64位模式中使用的内存地址需要限制在低于2G的内存空间中，否则，在切换到32位模式时，会出现内存地址冲突。

0x02 交叉模式混淆的概念

所有的IA指令都可以分为两类：兼容交叉模式的指令和不兼容交叉模式的指令。兼容交叉模式指的是指令的机器代码在32位和64位模式下都是有效的，并且含义也都是完全相同的，否则指令就会被视作不兼容交叉模式。

英特尔的软件开发人员手册中列出了每条指令及其可应用模式的详细信息。在这个指令表中，“64-bit Mode”字段和“Compat/Leg Mode”（32位）字段都显示是“有效的”（类似于图2中的“MOV r32, imm32”），这就是一条兼容指令。如果其中任何一个字段不是“有效的”，那么这就是一条不兼容指令。这两类指令都可以用于交叉模式混淆。

MOV—Move

Opcode	Instruction	Op/	64-Bit	Compat/	Description
B0+ rb ib	MOV r8, imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8 ^{***} , imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16, imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32, imm32	OI	Valid	Valid	Move imm32 to r32.
REXW + B8+ rd id	MOV r64, imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /0 ib	MOV r/m8, imm8	MI	Valid	Valid	Move imm8 to r/m8.

compatible instruction

incompatible instruction

图2-兼容与不兼容指令示例

2.1 兼容指令

虽然兼容指令的执行在32位和64位模式下都是相同的，但是，考虑到两种模式的差异，得到的结果也有可能是不同的。图3中就是一个简单的例子。从图中可以看到，无论是在32位还是64位模式下，在运行时反汇编同一组指令会得到完全相同的结果。

32-bit mode	64-bit mode
test!Callback64bit 001615c3 8bc4 mov eax, esp 001615c5 e800000000 call test!Callback64bit+0x7 001615ca 8bdc mov ebx, esp 001615cc 2bc3 sub eax, ebx	test!Callback64bit 00000000 00b215c3 8bc4 mov eax, esp 00000000 00b215c5 e800000000 call test!Callback64bit+0x7 00000000 00b215ca 8bdc mov ebx, esp 00000000 00b215cc 2bc3 sub eax, ebx
after code execution eax = 4	after code execution eax = 8

图3-兼容指令在不同模式下的执行示例

“call”只是简单地调用了下一条指令“mov ebx, esp”，并且这个代码块仅仅会检查栈指针在“call”指令的执行前后，修改了多少字节。由于栈框架大小的区别，在32位模式下，eax中的结果是4；在64位模式下，eax中的结果是8。这就证明了，同一组指令的结果要取决于模式，也就是说可以利用混淆数据或控制流来对抗某种单一模式的分析工具。

2.2 不兼容指令

利用不兼容指令进行混淆更加直接，因为，这类指令只在一种模式下是有效的。一条64位的不兼容指令要么会被视作有效指令，要么就在32位模式下解释成完全不同的指令，反之亦然（如图4），不支持处理动态模式转换的分析工具在这时就会遇到问题。

64-bit mode				32-bit mode			
00000000 012a15a6	50	push	rax	011a15a7	50	push	eax
00000000 012a15a7	53	push	rbx	011a15a8	53	push	ebx
00000000 012a15a8	4152	push	r10	011a15a9	41	inc	ecx
00000000 012a15aa	488bc3	mov	rax,rbx	011a15aa	52	push	edx
00000000 012a15ad	4c2bd0	sub	r10, rax	011a15ab	48	dec	eax
00000000 012a15b0	415a	pop	r10	011a15ac	8bc3	mov	eax,ebx
00000000 012a15b2	5b	pop	rbx	011a15ae	4c	dec	esp
00000000 012a15b3	58	pop	rax	011a15af	2bd0	sub	edx, eax
				011a15b1	41	inc	ecx
				011a15b2	5a	pop	edx
				011a15b3	5b	pop	ebx
				011a15b4	58	pop	eax

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + 8B /r	MOV r64, r/m64	RM	Valid	N.E.	Move r/m64 to r64.
REX.W + 2B /r	SUB r64, r/m64	RM	Valid	N.E.	Subtract r/m64 from r64.

图4-不兼容指令在不同模式下的执行示例

0x03 案例研究交叉模式混淆

一个很简单的例子就能证明交叉模式混淆的有效性。图5是交叉模式代码的示意图，这些代码首先会作为32位原生指令来执行。然后，通过使用远转移，跳转到CS 0x33，把处理器转换成64位模式，并继续执行64位原生指令（在32位模式中不兼容）。64位代码在执行后会解码另一组32位原生指令，在通过远返回把处理器切换回32位模式后，接着这些32位指令就会运行。最终，这一部分的32位代码会解密出密钥。

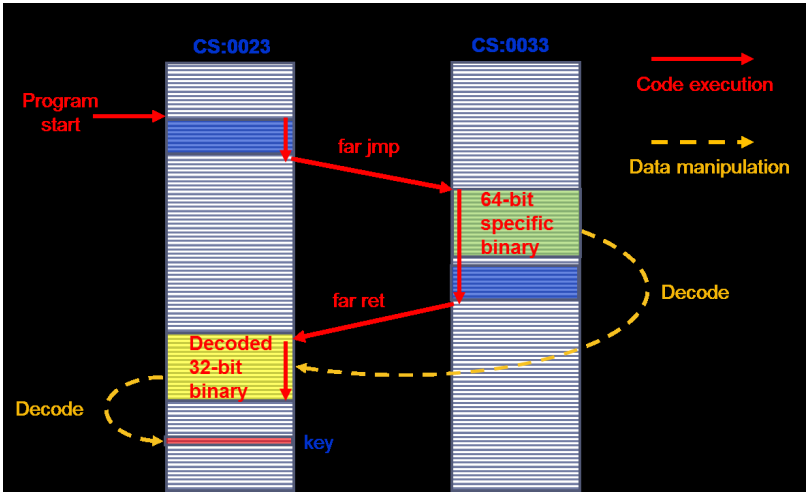


图5-使用了交叉模式编码的示例程序示意图

图6中是这个交叉模式代码在命令行中执行的结果。在十六进制中打印出的代码段选择器证实了，处理器确实从32位转换到了64位，又从64位转换成了32位。

```
C:\Users\Wild Sator\Documents\Visual Studio 2013\Projects\Xmode\exe>xmodeobf.exe
Encoded key = LlgàL^~US!11!^J00
CS selector = 23
CS selector = 33
CS selector = 23
Decoded key by Xmode Obfuscation = SOURCE Seattle 2015
```

图6-交叉模式代码示例在命令行中的执行

3.1静态分析工具

当使用32位的IDA Pro来分析上面的交叉模式代码时，很明显，在通过远转移进行模式交换后，IDA仍然会把64位的原生代码反汇编成32位的指令，导致反汇编结果与实际的64位指令出现偏差。（图7）

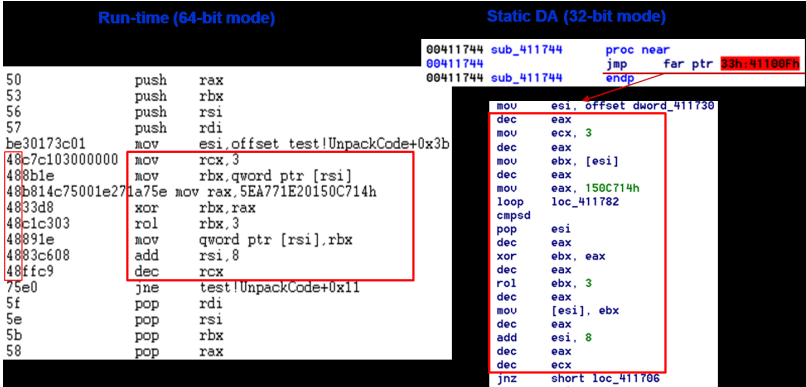


图7-对比IDA Pro的运行时反汇编与静态反汇编

如果使用64位版本的IDA Pro，IDA仍然会在32位模式下分析这一交叉模式代码，因为IDA会自动把这些代码识别成一个32位应用。此时，再通过远转移来转换模式，反汇编就会出错，并把控制流带到错误的地址（图8）。

```
00411744 sub_411744 proc near ; CODE XREF: sub_411032fj
00411744 jmp far ptr byte_41133F
00411744 sub_411744 endp

0041133F byte_41133F db 0D1h dup(0CCh) ; CODE XREF: sub_411744J
00411410 ; ===== SUBROUTINE =====
00411410
```

图8-64位IDA Pro的静态反汇编显示了错误的远转移地址

除了广泛使用的IDA，我们还使用了另一个静态分析工具-Radare2来测试我们的交叉模式代码。Radare2是一个开源的命令行框架，支持多个平台上的逆向工程。类似于IDA，Radare2在32位模式下无法正确地反汇编64位原生代码，而在64位下，无法正确地除了32位原生代码。

3.2动态分析工具

在运行时代码分析中，动态二进制插桩（DBI）是一种广泛应用的技术。在这次研究中，我们选择了两个常用的二进制翻译工具- DynamoRIO 和 Pin来运行测试程序。

这两个工具都无法处理从32位到64位的运行时模式转换，并且在通过远转移进行模式转换时会崩溃。图9显示，DynamoRIO 和 Pin都无法在64位模式下，打印出代码段选择器的值。

```
C:\Users\Wild Sator\Desktop\SOURCE 2015\demo>xmode_obfs.exe
Encoded Key =  LlgãL^~JUS!¶!!^J00
CS selector = 23
CS selector = 33
CS selector = 23
Decoded key by Xmode Obfuscation = SOURCE Seattle 2015
Press any key to continue . . .
```

(a)

```
C:\Users\Wild Sator\Desktop\SOURCE 2015\DynamoRIO-Windows-5.1.0-RC1\bin32>drrun.
exe .\..\demo\xmode_obfs.exe
Encoded Key =  LlgãL^~JUS!¶!!^J00
CS selector = 23
C:\Users\Wild Sator\Desktop\SOURCE 2015\DynamoRIO-Windows-5.1.0-RC1\bin32>
```

(b)

```
C:\Users\Wild Sator\Desktop\SOURCE 2015\pin-2.14-71313-msvc12-windows>pin.exe --
.\demo\xmode_obfs.exe
Encoded Key =  LlgãL^~JUS!¶!!^J00
CS selector = 23
C:\Users\Wild Sator\Desktop\SOURCE 2015\pin-2.14-71313-msvc12-windows>
```

(c)

图9-（a）命令行（b）DynamoRIO（c）Pin执行交叉模式代码的情况

由于需要切换到32位模式，DynamoRIO在执行64位应用时也遇到了类似的崩溃问题，而Pin直接停止运行，显示错误信息“不支持通过远返回来转换不同的代码段”。

有趣的是，在仅仅使用兼容指令时，有时候DBI工具可能会在运行了用于切换模式的远转移后，继续运行，但是，实际上模式并没有切换。图10中显示，DynamoRIO使用兼容指令运行了代码，与2.1中讨论的例子一样。可以看到，程序运行到了模式交换点之后，但是，没能真正的转换到代码段 0x33，而是停留在了CS 0x23。eax中的返回值是4而不是8，这与直接执行时的结果不同，从而为DBI环境监测创造了机会。

```
C:\Users\Wild Sator\Documents\Visual Studio 2013\Projects\Xmode\exe>32_64_compc
ode.exe
CS selector = 23
Return value under32bit = 4
CS selector = 33
Return value under64bit = 8
```

(a)

```
C:\Users\Wild Sator\Documents\Visual Studio 2013\Projects\Xmode\BT\DynamoRIO-Win
dows-5.1.0-RC1\bin32>drrun.exe .\..\..\exe\32_64_compcode.exe
CS selector = 23
Return value under32bit = 4
CS selector = 23
Return value under64bit = 4
```

(b)

图9-（a）命令行（b）DynamoRIO执行兼容交叉模式的代码

3.3调试工具

我们还发现，在调试包含运行时模式切换的代码时，交叉模式代码还会给OllyDbg和WinDbg这样的常用调试工具造成问题。32位的OllyDbg（目前64位不可用）在从32位切换到64位模式后，无法继续调试。而64位的WinDbg可以正确的运行交叉模式代码，但是在交叉模式条件下，修改过的代码会出现单步执行问题。

0x04 总结

利用64位Windows系统向下兼容32位程序的优势，交叉模式编码可以作为一种非常有效的混淆技术。目前，大多数静态分析工具、DBI工具和调试工具都是基于单一的处理模式，这些工具在处理包含运行时模式交换的代码时就会遇到问题。而且，只要在设计时稍加注意，交叉模式代码还

从理论上说，只要让这些分析工具能适应动态模式交换，那么交叉模式问题就会迎刃而解，只不过需要大量的工程努力。

0x05 致谢

感谢李晓宁，亚欧对此次研究和审阅做出的贡献。

0x06 参考

- {1}<https://en.wikipedia.org/wiki/WoW64> (<https://en.wikipedia.org/wiki/WoW64>)
- {2}Intel® 64 and IA-32 Architectures Software Developer's Manual
- {3}The Performance Cost of Shadow Stacks and Stack Canaries. Thurston H.Y. Dang, Petros Maniatis, David Wagner. ASIACCS 2015
- {4}Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz. 36th IEEE Symposium on Security and Privacy (Oakland), May 2015
- {5}Exploring Control Flow Guard in Windows 10. Jack Tang. Trend Micro Threat Solution Team, 2015
- {6}ROP is Still Dangerous: Breaking Modern Defenses. Nicholas Carlini and David Wagner. 23rd USENIX Security Symposium, Berkeley 2014
- {7}Windows 10 Control Flow Guard Internals. MJ0011, POC 2014
- {8}Write Once, Pwn Anywhere. Yu Yang. Blackhat 2014
- {9}Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of
- {10}Embedded Systems Against Software Exploitation. Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. DAC 2014
- {11}Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Columbia University, 22nd USENIX Security Symposium 2013
- {12}kBouncer: Efficient and Transparent ROP Mitigation. Vasilis Pappas. Columbia University 2012 Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using
- {13}Performance Counters. Liwei Yuan, Weichao Xing, Haibo Chen, Binyu Zang. APSYS 2011
- {14}Transparent Runtime Shadow Stack: Protection against malicious return address modifications. Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. 2008
- {15}Control-Flow Integrity Principles, Implementations, and Applications. Martín Abadi, Mihai Budiu, Ulfar Erlingsson, Jay Ligatti. CCS2005
- {16}IROP – interesting ROP gadgets, Xiaoning Li/Nicholas Carlini, Source Boston 2015

☆收藏 分享

昵称

验证码



写下你的评论...

发表



雪村宅也 2015-12-09 17:30:29
这tag里基本都是炒股的.....

回复




造化不弄僵尸大阿叔 2015-12-09 16:11:30
废P，，，DOS还用16位，说不兼容不就完了，，，，推堆还蓝屏呢，，，，


👤 回复

 **Samsong** 2015-12-09 14:48:36
赞，有意思


👤 回复

 **Doric-2015** 2015-12-09 14:42:44
牛逼。。。

👤 回复

 **ramnit** 2015-12-09 12:12:51
这个写的不错，涨了姿势，感谢作者

👤 回复

 **fuzz-ing** 2015-12-09 12:04:26
这个很好，火钳留名

👤 回复

感谢知乎授权页面模版