تغییر ماهیت[1] نرم افزار با دگرشکلی های کد اسمبلی

در این مقاله روشی را برای تغییر ماهیت نرم افزار ارائه می دهیم. این کار با اِعمال دگرشکلی ها (یا تغییرشکلها)[2] روی کدهای اسمبلی انجام می شود. ما مثالی را ارائه می دهیم که امکان پذیری این تکنیک را تشریح می کند و سپس روی فواید بالقوه این خط مشیء بحث می کنیم. همچنین کاربردهای احتمالی دیگرِ این چنین تکنیک ها را نیز ذکر خواهیم کرد.

۱. مقدمه

در بعضی از شرکتها، هزینه های کپی شدن نرم افزار سالانه بیش از ۱۰ میلیارد دلار به مسئولان آن شرکت ضربه می زند. یک راه برای دفاع در مقابل کپی نرم افزار، اعمال یک تغییر ماهیت روی آن است. گرچه این کار جلوی تمام دزدی های نرم افزاری (که منظور همان کپی های غیرمجاز است) را نخواهد گرفت، اما یک تغییر ماهیت اساسی، می تواند منبع نرم افزار کپی شده را تعیین کند و از این رو حجم زیادی از این گونه عملیات کاسته می شود.

طراحی طرح های تغییر ماهیت دیجیتالی که بتواند در یک محیط طاقت فرسا بخوبی کار کند، کار دشواری است. بسیاری از تکنیک های تغییر ماهیت، بسادگی حذف یا بمحض شناسایی تحریف شده اند. برای مثال، "stirmark" براحتی یک ماهیت اعمال شده را از یک تصویر دیجیتالی بر می دارد.

در این مقاله، ما یک طرح تغییر ماهیت مبتنی بر دستکاری روی نرم افزار در سطح کدهای اسمبلی ارائه می دهیم. این دستکاری ها نسخه های گوناگونی از یک قطعه نرم افزاری را نتیجه می دهند که هر نسخه کد مجزایی دارد، اما عاملیت آن با دیگر نسخه ها برابر میباشد!!

خط مشیء تغییر ماهیت ما، از ویروس های کامپیوتری دگردیس (دگرشکل) الهام گرفته شده است. ویروس های دگردیس، با هر بار تکرار، درون تکه کد مجزا تغییر شکل می یابند. این عمل تشخیص آنها را بسیار دشوار می سازد، چرا که هیچ امضای ثابتی برای نرم افزار پویش ویروس جهت تشخیص آنها وجود ندارد. در زمینه تغییر ماهیت، مسئله دگردیسی اجازه می دهد که تغییر ماهیت در هر قسمت و نسخه از نرم افزار به صورت یکتا جاسازی شود. اگرچه هنوز هم این امکان وجود دارد که یک تغییر ماهیت را در یک قسمت یا نسخه معین از نرم افزار حذف کنیم، اما تغییرپذیری (و قابلیت تنظیم و تغییر) کد، حذف اتوماتیک تغییرماهیت ها را از تعداد زیادی از اقسام کد بسیار دشوار می سازد. در نتیجه، حذف تغییرماهیت هنوز به صورت یک فرآیند دستی (غیرخودکار) و طاقت فرسا باقی خواهد ماند.

تغییرات کدهای دگردیس نیز بعنوان روشی جهت افزایش ضریب تفاوت در نسخه های نرم افزار مطرح شده است. در مقایسه با تنوع ژنتیکی سیستم های زیستی، توافق انجام شده که نرم افزارهایی که گوناگونی آنها افزایش یافته است، می توانند تاثیر حملات مضر را روی سیستم های کامپیوتری محدود کنند. سود دیگری که از خط مشیء ما برای تغییرماهیت نرم افزار عاید می شود، تولید یک نرم افزار تغییرشکل یافته و متفاوت می باشد و لذا ممکن است مقاومت چنین نرم افزارهایی در مقابل بسیاری از گونه های حمله ای افزایش یابد.

---

[1] اصطلاحا به این کار Watermarking می گوییم.

[2] Transformation

۲. تغییرماهیت نرم افزار

برای تشریح خط مشی ای که ما در عملیات تغییرماهیت اتخاذ کرده ایم، برنامه "Hello World" (که تقریبا در هر کتاب برنامه نویسی در هر زبانی نمونه ای از این دست معرفی می شود) را مورد نظر قرار می دهیم. ما کار را با یک کد منبع C شروع می کنیم (hello.c) که در نتیجه آن یک کد اسمبلی با نام hello.s تولید می شود. نمونه ای از چنین کدهای اسمبلی در تصویر یک نشان داده شده است.

سپس، ما تغییرات خود را به کد اسمبلی hello.s الحاق می کنیم. در تصویر دو، ما تغییرماهیتی را جاسازی کرده ایم که حاوی شماره مشتری با مقدار ۴۴۳۶ می باشد. این تغییرماهیت با بلوک کد زیر اضافه شد:

```
push %eax
movl $44, %eax
movl $36, %eax
pop %eax
```

همچنین تغییرات زائد (و اضافی) نیز به کد اضافه کرده ایم تا کار نفوذگر را سخت تر کنیم. فرض می کنیم که نفوذگر چندین نسخه از کد تغییریافته را دارد، در این صورت می تواند در جهت مقایسه تفاوت های آنها اقدام کند. با اضافه کردن چندین تغییر ساختگی، نفوذگر در تعیین کد مربوط به تغییرماهیتِ اصلی مشکلات بیشتری خواهد داشت. در کل ما چندین نقطه را انتخاب می کنیم که اطلاعات مربوط را می توان در آنجا مخفی کرد. در کنار آن، مناطق دیگری را جهت اضافه کردن تغییرات ساختگی خود انتخاب می کنیم. سپس، یک فایل اجرایی را از این کد اسمبلی تغییریافته تولید می کنیم.

```
                .file   "hello.c"
                .def    ___main;        .scl    2;      .type   32;     .endef
                .text
LC0:
                .ascii "Hello, world\12\0"
                .align 2
.globl  _main
                .def    _main; .scl     2;      .type   32;     .endef
_main:
                pushl   %ebp
                movl    %esp, %ebp
                subl    $8, %esp
                andl    $-16, %esp
                movl    $0, %eax
                movl    %eax, -4(%ebp)
                movl    -4(%ebp), %eax
                call    __alloca
                call    ___main
                movl    $LC0, (%esp)
                call    _printf
                movl    $0, %eax
                leave
                ret
                .def    _printf;.scl    2;      .type   32;     .endef
```

تصویر ۱: **Hello World**

برای تولید N نسخه تغییرماهیت یافته و مجزا از نرم افزار، مراحل فوق را برای هر نسخه تکرار می کنیم. تمامی فایل های ساخته شده از لحاظ عاملیت و کارکرد یکسان هستند، اما در مجموعه ی تغییرات مورد استفاده، منطقه انجام تغییرات و همچنین شماره مشتری جاسازی شده در تغییرماهیت، تفاوت دارند.

برنامه "Hello World" تنها برای مقاصد تشریحی انتخاب شد. در عمل، جاسازی شماره مشتری در یک وضعیت غیرمستقیم بسیار مستحکم تر خواهد بود. برای مثال، تغییرات مشخصی می توانند به صورت "1" ظاهر شوند و تغییراتی هم به صورت "0". در این صورت، می توانیم معادل باینری ID (مشخصه مشتری) را جاسازی کنیم و برای استحکام هر چه بیشتر می توانیم تغییرماهیت مطلوب را چندین بار اضافه کنیم. روش ما در جاسازی تغییرماهیت، چندین هدف را دنبال می کند. اگر شماره مشتری مستقیما در کد جاسازی شود، برای نفوذگر تشخیص اطلاعات حیاتی در کد بسیار راحت خواهد بود. همچنین، با اضافه کردن چندین کپی از تغییر ماهیت، نفوذگر بایستی هر یک از این کپی ها را به منظور خنثی سازی تغییرماهیت لغو کند. با گنجایش تغییرات ثانویه (یعنی تغییرات اضافی که نه به صورت ۰ و نه به صورت ۱ ظاهر می شوند)، حذف تغییرماهیت اِعمال شده بسیار دشوار می شود.

تکنیک تغییرماهیت ما، تنها می تواند تغییرماهیت را از یک فایل اجرایی معین بخواند. برای استخراج تغییرماهیت از یک فایل اجرایی، ما کار را به این صورت دنبال می کنیم: ابتدا، فایل اجرایی را با استفاده از یک Disassembler مانند IDA Pro (یکی از بهترین دیزاسمبلرهای موجود)، disassemble می کنیم. بسیاری از تغییراتی که قبلا در کد اسمبلی اضافه کرده ایم، در خلال فرآیند Assembly/Disassembly از بین خواهند رفت. اما ما اطلاعاتی که تغییرات را در خود دارند به دقت انتخاب کرده ایم، بطوریکه آنها از این رویداد محافظت می شوند. بعد از diassembly، بدنبال تغییرات مربوطه در کد دیزاسمبل شده می گردیم. نرم افزار IDA Pro، به شخص اجازه می دهد تا یک فایل اجرایی را به منظور تولید یک فایل "asm"، دیزاسمبل کند که این فایل، یک اسمبلی از فایل اجرایی می باشد. برای مثال در تصویر ۲، ما در فایل asm تولید شده به دنبال الگوی زیر می گردیم:

```
push    eax
mov     eax, xxh
nop
mov     eax, xxh
pop     eax
```

hello.s :برنامه اصلی	hello.s :کد تغییر یافته

_main:
pushl %ebp
movl %esp, %ebp
subl $8, %esp
andl $-16, %esp
movl $0, %eax
 	
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
 	
call __alloca
call ___main
movl $LC0, (%esp)
	_main:
	pushl %ebp
	movl %esp, '
	subl $8, %e
	andl $-16, %esp
	movl $0, %eax
	push %eax
	movl $44, %eax
	nop
	movl $36, %eax
	pop %eax
	movl %eax, -4(%ebp)
	movl -4(%ebp), %eax
	xor $0, %ebp
	call __alloca
	call ___main
	movl $LC0, (%esp)
	push %ebp
	movl $23, %ebp

Watermark "4436"

گنگ کردن کد

call _printf movl $0, %eax leave ret .def _printf; .scl 2; .type 32; .endef	pop %ebp call _printf _sub1: movl $0, %eax nop leave ret .def _printf; .scl 2; .type 32; .endef
کد تغییر شکل یافته	تصویر ۲:

در پیدا کردن این الگو (یعنی تغییر ماهیت خودمان)، ما اطلاعات مشخصه مشتری (یعنی ۴۴۳۶) را استخراج می کنیم. در این مثال، تغییرات باقیمانده گیج کننده هستند و می توانیم چگونگی دسزاسمبل شدن آنها را بفهمیم. به صورت خلاصه، با یک فایل منبع معین (در اینجا hello.c)، ما یک فایل اسمبلی از آن تهیه می کنیم (hello.s). سپس تغییرات را در آن اعمال می کنیم که این تغییرات شامل تغییراتی که تغییرماهیت (watermark) ما را دارا می باشند نیز می شود. آنگاه، این فایل تغییریافته را لینک می کنیم (hello_new.s) تا یک فایل اجرایی به صورت hello_new.exe ایجاد شود.

تغییراتی که تغییرماهیت اصلی را دارا می باشند بایستی از جانب ما در خود فایل اجرایی قابل شناسایی باشند و تغییرات حیاتی بایستی در خلال فرآیند **assembly/disassembly** بدون تغییر و دست نخورده بمانند. در خواندن تغییرماهیت، ما تنها بدنبال اطلاعاتی که تغییرات ما را دارا می باشند می گردیم.

در قسمت بعد راجع به چندین تغییر احتمالی که می توانند به کد اضافه شوند تا به این صورت یا تغییرماهیت را حمل کنند یا به صورت عامل گیج کننده عمل کنند. تغییرات زیادی به این صورت امکان پذیر است، اما ما تنها موردهای ساده را ذکر می کنیم.

۳. دگرشکلی ها یا تغییرشکل ها (یا به صورت کوتاه تغییرات)

۳,۱ اضافه کردن NOP ها

NOP مخفف عبارت No Operation می باشد. **NOP** به پردازنده دستور می دهد که از این دستورالعمل (منظور دستوری که پردازنده در حال حاضر روی آن کار می کند) بپرد و آنرا نادیده انگارد. بنابراین، یک توالی از **NOP** ها را می توان در برنامه اضافه کرد تا، یا به عنوان بخشی از تغییرماهیت عمل کند، یا بعنوان یک **cave** یا **placeholder** (فضایی که می توان در باینری به راحتی آنرا با کدهای دلخواه خود پر کرد) برای تغییراتی که بعدا اضافه می کنیم عمل کند. در این روش، اندازه کلی فایل باینری از یک نسخه نسبت به نسخه ای دیگر تفاوت محسوسی نخواهد داشت.

۳,۲ برچسب های زائد

این تکنیک بسادگی پارازیت هایی را در کد اضافه می کند و به ما کمک می کند که بین نسخه های متفاوت از یک نرم افزار، تنوع و گوناگونی لازم را ایجاد کنیم، بنابراین تولید آن برای یک مهندس معکوس یا Reverse Engineer مشکلات بیشتری ایجاد می کند (از جنبه گمراهی). برچسب های زائد را می توان در هر جایی از برنامه اضافه کرد و برنامه نیز به هیچ وجه تحت تاثیر این برچسب ها قرار نخواهد گرفت. مثال زیر را در نظر گیرید:

- کد اصلی:

```
mov ax,1
mov ax,2
```

- کد تغییریافته:

```
x1: mov ax,1
x2: mov ax,2
```

۳,۳ تغییرات منطقی و حسابی

بسیاری از دستورات منطقی و حسابی را می توان مورد استفاده قرار داد، بطوریکه مقادیر ثبات ها صفر بنظر آید یا اینکه ظاهرا در مقدار ثبات تغییری انجام شده است (در حالی که در عمل این طور نبوده است)! برای مثال، ما می توانیم یک ثبات را با مقدار صفر XOR کنیم و مقدار آن بدون تغییر می ماند! به همین ترتیب، می توانیم مقادیر ثبات ها را یک واحد اضافه کنیم و مجددا یک واحد کم کنیم تا به این صورت، هم فضای کد را شلوغ کرده باشیم (برای سخت کردن کارِ مهندس معکوس) و هم مقادیر ثبات ها بدون تغییر بمانند! همچنین می توانیم مقدار یک ثبات را با صفر OR کنیم یا اینکه مقدار ثبات را دوبار NEG (خلاصه عبارت negate) کنیم یا از ADD/SUB بعنوان تغییرات استفاده کنیم، در حالیکه ثباتها هنوز هم بدون تغییر مانده اند!!

تقریبا تعداد نامحدودی از این تغییرات وجود دارند. دستوراتی که می توانند برای چنین تغییراتی مورد استفاده قرار گیرند به صورت زیر هستند (اما تنها محدود به این موارد نمی شوند):

ADD, SUB, OR, XOR, INC, DEC, NEG

برای مثال عبارت زیر مقدار صفر را به ثبات BX اضافه می کند:

```
add bx, 0;
```

۴. مشکلات

توانایی خواندنِ صحیحِ تغییرماهیت از یک دیزاسمبلی بزرگ، مشکل دارترین قسمت کار است. چون بسیاری از تغییرات در خلال اسمبلی/دیزاسمبلی از دست رفته اند، لذا تغییرات را بایستی بدقت انتخاب کرد. برای مثال، تغییرات موجود در مثال تصویر ۲ را می توان از یک دیزاسمبلی از فایل اجرایی خواند. در حالت کلی، این امکان وجود ندارد که یک فایل اجرایی را به صورت اتوماتیک دیزاسمبل کرد و سپس نتایج را در یک فایل اجرایی عامل (منظور فایل اجرایی است که نتایج کار را می توان در آن مشاهده کرد) اسمبل کرد. این جمله تقریبا درست است، چرا که تشخیص کد از داده ها بدون مداخله دستی (یعنی مداخله خود فرد، نه ابزار اتوماتیک یا ...) غیر ممکن است! در نتیجه، حملات کلی روی طرح تغییرماهیت ما، احتمالا بصورت دستی (غیراتوماتیک) باقی خواهند ماند!

۵. دیگر نرم افزارهای تحت تاثیر

چندین نرم افزار برای تکنیک های تغییرشکل کد اسمبلی در این قسمت بحث شده اند.

۵,۱ تغییرماهیت مخفی یا پنهان (invisible)

نحوه تغییرماهیت برنامه در بالا توضیح داده شد. چون گونه های بسیاری از تغییرماهیت وجود دارد، لذا تاکید می کنیم که خط مشیء ما برای الحاق یا اضافه کردن یک تغییرماهیت مخفی و گنگ طراحی شده است (منظور از تغییرماهیت گنگ و مخفی این است که نتوان براحتی آن تغییرماهیت را تشخیص داد).

۵,۲ یگانگی کد (Code Uniqueness)

یگانگی نرم افزار، تکنیکی است که چندین کپی از نرم افزار را تولید می کنیم در حالیکه همه آنها عاملیت یکسانی دارند، اما هر کپی منحصر بفرد می باشد. یگانگی کد جهت تولید سیستمی طراحی شده است که نفوذگر ممکن است در شکستن و مغلوب شدن بر یکی از نسخه های نرم افزار پیروز شود و این کار معمولا با عملیات مهندسی معکوس بدون احاطه بر کل سیستم انجام می شود.

۵,۳ محافظت در برابر ویروس ها

تکنیک های بحث شده در بالا، یک نرم افزار دگردیس (تغییرشکل یافته) را تولید می کند –نسبت به نسخه اصلی بسیار متفاوت، اما هنوز با همان عاملیت یکسان. در نتیجه، یک ویروس، برای مثال ممکن است در یافتن *نقطه ورودی (entry point)* که جهت حمله به نرم افزار لازم است، ناکام بماند. همچنین این امکان وجود دارد که آسیب پذیری را که ویروس اکسپلویت می کند فقط در شرایط قبلی و مناسب خود (منظور همان شرایطی که نرم افزار اولیه وجود داشته است) قابل اکسپلویت باشد. برای مثال حملات سرریزبافر اغلب بسیار حساس هستند و به تزریق دقیق کد حمله نیاز دارند. در نرم افزار دگردیس، حتی اگر یک سرریز بافر در تمام نسخه ها وجود داشته باشد، سناریو حمله در یک نسخه نسبت به نسخه دیگر تفاوت خواهد داشت و لذا یک کد اکسپلویت روی تمام نسخه ها کارکرد نخواهد داشت. در این روش، گویا نوعی لایه محافظتی به برنامه اضافه شده است!! لذا، طرح تغییرماهیت ما محافظات بیشتری در مقابل ویروس ها نیز ارائه می دهد.

۶. نتایج

تکنیک تغییرماهیت نرم افزار که در بالا ذکر شد، احتمالا روشی جالب (و قابل توسعه) برای جاسازی یک تغییرماهیت گنگ و مخفی در نرم افزار می باشد. این تکنیک فواید بالقوه ای نیز دارد: نخست اینکه، اگر نتوان فرآیند حذف تغییرماهیت را بنوعی اتوماتیک کرد، این عمل بسیار دشوار خواهد بود. دوم اینکه، تغییرماهیت را می توان با اضافه کردن چندباره آن به کد سخت تر و گنگ تر ساخت. سوم اینکه، این تکنیک یک جنبه امنیتی مثبت نیز دارد که در نتیجه تولید نرم افزار متفاوت رنگ می گیرد.

منابع

- Report: Software Piracy Dips http://www.cnn.com/2003/TECH/biztech/06/03/software.piracy.reut/
- M. Stamp, Risks of Monoculture, to appear in *Communications of the ACM*
- The IDA Pro Disassembler and Debugger http://www.datarescue.com/idabase/
- Mayer, J., Assembly Language Programming: 8086/8088, 8087, John Wiley & Sons, 1988.
- M. Stamp, Digital Rights Management: http://home.earthlink.net/~mstamp1/papers/DRMpaper.pdf
- Mishra, P. A Taxonomy of Uniqueness Transformations, 2003.

تالیف و ترجمه: سعید بیکی (cephexin@secumania.net)

© Secumania Security & Vulnerability Research Lab
www.secumania.net