Obfuskace strojového x86 kódu

MazeGen, 2010-06-21

Revize: 1.0

Obfuskace kódu je oblíbenou technikou pro ztížení jeho analýzy nebo jeho rozpoznání na základě vzorů, což dnes hojně využívá například malware nebo softwarové protektory spustitelných souborů.

Článek byl původně napsán pro prielom #25.

Obfuskací kódu (angl. code obfuscation) se obecně označuje technika pro přetvoření kódu (i zdrojového) do formy těžko pochopitelné člověkem, ale i nějakým programem. V tomto kontextu ale půjde jenom o „rozbití“ existujícího strojového x86 kódu, i když obfuskátory asemblerovských zdrojáků (za účelem obfuskovaného výsledného strojáku) existují taky (třeba PELock obfuscator). Anglicky se tomu občas říká i code morphing. Nebudeme se tedy zabývat generací úplně nového kódu např. na základě nějakých abstraktních vzorů.

Napsat i jednoduchý obfuskátor strojáku není úplně triviální záležitost, protože na začátku vyžaduje pokročilejší, alespoň statický disassembler, který nějak popíše vstupní instrukce, a na konci reasembler, který celý kód opraví a znovu sestaví do funkční podoby. K tomu, aby bylo možné kód bezpečně automaticky obfuskovat, musí být splněna řada podmínek (např. je potřeba najít všechna návěští v kódu, které musí být možné nakonec opravit). Už to je problém sám pro sebe, který navíc nemusí být řešitelný (viz Halting problem). Řeší se tak, že kód, který má být obfuskovaný, musí být už předem připraven, třeba tak, že obsahuje zvláštní nápovědu pro obfuskátor. To je ale téma na zvláštní článek.

K pochopení celého procesu a jednotlivých metod, jak kód „zamlžit“, pomůže pohled do historie vývoje obfuskátorů, který prudce probíhal a dal se sledovat u vývoje klasických virů, napadajících spustitelné soubory. V tomto smyslu se často mluví o metamorfismu (angl. metamorphism), protože virus vlastně obfuskuje sama sebe.

Metody obfuskace

Jedním z nejznámějších průkopníků metamorfních virů byl Z0MBiE. Popsal třeba podmínky, za kterých je možné bezpečně prohazovat instrukce, což je jedna z důležitých metod, jak se zbavit jak konstantních vzorů v kódu, tak ztížit pochopení algoritmu. Zmiňuje tuto posloupnost instrukcí:

```
[A] add eax, ecx
[B] adc eax, edx
[C] add ebx, ecx
[D] adc ebx, edx
```

Instrukce **B** závisí na **A** (`ADC` testuje příznak `CF` nastavený instrukcí `ADD`) a instrukce **D** závisí na **C**. Současně **C** nezávisí na **A** ani na **B**, proto je lze prohodit (níž jsou další příklady):

```
[C] add ebx, ecx
[D] adc ebx, edx
[A] add eax, ecx
[B] adc eax, edx
```

Aby mohl s kódem manipulovat, vytvořil XDE, eXtended Disassembler Engine, který je založen na ADE, který umožňuje počáteční disasembling a konečný zpětný asembling, což je jedna z nutností popsáných výše.

Metamorfismus později pěkně popsal Mental Driller. Zmiňuje celý proces, kterým musí kód projít, aby ho šlo modifikovat a aby byl na konci stále funkční. Zmiňuje další důležitou techniku, rozložení instrukce na více jiných, ale se stejným výsledkem, konkrétně `PUSH REG` na (níž jsou další ukázky):

```
push reg          ; původní instrukce, např. push eax
 ->
mov [mem], reg    ; předem připravená dočasná paměť
push [mem]
```

Do (automatické) analýzy takového kódu je potom navíc potřeba zahrnout nový odkaz do paměti, což ji zpomaluje a komplikuje.

Zatím byly zmíněny dvě metody obfuskace: Prohazování instrukcí (to zahrnuje i celé bloky instrukcí) a jejich rozklad. Dá se říct, že tyto jsou ty nejdůležitější, protože využívají pouze původních instrukcí a kvůli jejich komplikovanosti není lehké navrátit kód do původního stavu.

Změna toku řízení

Další metoda, jejíž podstatou není přidávání nových instrukcí, je změna toku řízení (angl. control-flow-graph mutation). Jde prostě o „vytržení“ kusu kódu, jeho přemístění a slinkování s původním kódem pomocí instrukcí `JMP`:

```
 ; původní kód

 mov al, [ebx]
 inc ebx          ; tato instrukce bude přesunuta
 sub al, 32

 ; obfuskovaný kód

 mov al, [ebx]
 jmp get_over
get_back:
 sub al, 32
 jmp skip        ; přeskoč přesunutou instrukci
get_over:
 inc ebx
 jmp get_back    ; vrať se na původní místo
skip:
```

Za splnění dalších podmínek lze využít i instrukce `CALL-RET` a tím vlastně vytvořit falešnou proceduru:

```
 mov al, [ebx]
 call get_over
 sub al, 32
 jmp skip
get_over:
 inc ebx
 ret
skip:
```

Další metody už přidávají nové instrukce:

Falešné skoky

Jde o přidávání nadbytečných skoků:

```
 ; původní kód

 mov al, [ebx]
 inc ebx
 sub al, 32

 ; obfuskovaný kód

 mov al, [ebx]
 jc bebacksoon  ; falešný skok, který se možná neprovede, nezáleží na tom
get_back:
 inc ebx
 sub al, 32
 jmp skip       ; přeskoč přidaný skok zpět
bebacksoon:
 jmp get_back   ; pouze návrat na původní místo
skip:
```

Nebo „zřetězení“ existujícího skoku za účelem zakrytí skutečné cílové adresy:

```
 ; původní kód

get_null:
 mov al, [ebx]
 inc ebx
 cmp al, 0
 jne get_null

 ; obfuskovaný kód

get_null:
 mov al, [ebx]
 jmp skip        ; přeskoč přidaný skok na get_null
xget_null:
 jmp get_null
skip:
 inc ebx
 cmp al, 0
 jne xget_null   ; zřetězení s meziskokem
```

Nebo vytvoření skoku do instrukce, což dokáže zmást ty disasemblery, které nedokáží disasemblovat na návěští skoku, které je uvnitř už dříve disasemblované instrukce. Přidáme např. operační kód `B8`, který vytvoří instrukci `MOV EAX, původní_4_bajty`. Dál přidáme instrukci skoku, která zajistí, že se spustí skutečná původní instrukce (ve sloupci vlevo je strojový kód instrukcí):

```
; původní kód

8A83 00010000  mov al, [ebx+100]

; obfuskovaný kód

EB 01          jmp $+3            ; přeskoč přidaný bajt 0xB8
B8 8A830001    mov eax, 100838A   ; 8A, 83, 00, 01 = 4 bajty z původního kódu
0000           add [eax], al      ; 0000 = zbývající 2 bajty
```

```
; jinak zapsáno

EB 01          jmp skip
B8             db 0B8
              skip:
8A83 00010000  mov al, [ebx+100]
```

Takže jsme vlastně přidali jenom 3 bajty `EB`, `01`, `B8` před původní instrukci.

Nicnedělající kód

Pro zpomalení jak automatické analýzy (emulátoru), tak ručního prohledávání kódu obfuskátory vkládají kód, jenž ve výsledku nemá žádný význam. Třeba Z0MBiE proto vytvořil ETG, Executable Trash Generator, který měl za úkol zpomalit antivirové emulátory vytvořením spousty nicnedělajícího kódu, vloženého mezi dekryptor a samotný kód viru.

Takovým nicnedělajícím kódem může být třeba následující funkce, volaná z náhodně vybraných míst původního kódu:

```
do_nothing:
 push ebp
 mov ebp, esp    ; simuluj vytvoření stack frame
 push ecx
 mov ecx, 0      ; nastav na nula, takže
 repe cmpsw      ;  REPE se neprovede ani jednou
 pop ecx
 pop ebp         ; uvolni stack
 retn
```

Dál se dá uvažovat například o vkládání méně známých NOP instrukcí jako `FNENI` nebo `FNDISI` a dalších metodách.

Obfuskace prakticky

Pro ilustraci všech uvedených metod budeme krok za krokem obfuskovat jednoduchou implementaci funkce strcpy ze standardní knihovny:

```
strcpy:
 push ebp
 mov ebp, esp          ; parametry adresuj přes EBP
 mov edx, [ebp+08]     ; zapamatuj si parametr r pro výstup
 mov esi, [ebp+0C]     ; zdrojový řetězec cs
 mov edi, [ebp+08]     ; cílový řetězec r

copy:
 mov al, [esi]
 mov [edi], al         ; kopíruj bajt po bajtu
 inc esi
 inc edi               ; posuň ukazatele
 cmp al, 0             ; konec řetězce?
 jne copy

 mov eax, edx          ; vrať r
 pop ebp               ; obnov EBP
 ret
```

Rozklad instrukcí

Začneme rozkladem instrukcí, který nám zvýší počet instrukcí pro aplikaci ostatních metod.

`push ebp` - interně jde o dvě operace: snížení `ESP` o 4 bajty a vložení `EBP` na `[ESP]`. Toho využijeme, ale celou operaci ještě víc zamotáme. Nejdřív provedeme `push` náhodně vybrané konstanty, kterou potom přepíšeme registrem `EBP`:

```
push ebp
 ->
push 0F4DCE10
mov [esp], ebp
```

`mov ebp, esp` - použijeme rozklad, který nejprve dočasně modifikuje zdrojový registr `ESP` náhodně vybraným registrem, vloží ho do `EBP` a potom opraví oba registry na správné hodnoty. Poznámka: předpokládáme, že kód běží v user mode, kde můžeme dočasně narušit hodnotu `ESP`.

```
mov ebp, esp
 ->
sub esp, ebx
mov ebp, esp
add ebp, ebx
add esp, ebx
```

`mov edx, [ebp+08]` a následující podobné instrukce: pokud nemá obfuskátor informaci o tom, že zdrojová paměť je zapisovatelná (jde o stack), nemůže použít řadu rozkladů, které nejprve nějak modifikují zdrojovou paměť. Pořád ale existuje několik možností, konkrétně využijeme možnosti postupného plnění registru `EDX`:

```
mov edx, [ebp+08]
 ->
mov dx, [ebp+0A]   ; vyšší WORD
shl edx, 10        ; do vyššího WORD EDX
mov dh, [ebp+09]
mov dl, [ebp+08]
```

Instrukce `SHL` na rozdíl od původního `MOV`, který rozkládáme, sice přepíše příznaky (jako `CF`, `ZF` atd.), ale obfuskátor dokáže analýzou zjistit, že příznaky lze přepsat, protože na nich žádná z následujících instrukcí nezávisí (nenásleduje např. instrukce `ADC` nebo `JZ`).

`mov esi, [ebp+0C]` - použijeme třeba jednoduchý `push-pop` rozklad:

```
mov esi, [ebp+0C]
 ->
push dword [ebp+0C]
pop esi
```

`mov edi, [ebp+08]` - můžeme použít např. načtení adresy do náhodně vybraného registru (který si nejprve uložíme a potom zase obnovíme):

```
mov edi, [ebp+08]
 ->
push edx
lea edx, [ebp+08]
mov edi, [edx]
pop edx
```

`mov al, [esi]` - obfuskátor může využít faktu, že nakonec je celý `EAX` přepsán, takže nezáleží na původní hodnotě v `AH`:

```
mov al, [esi]
 ->
mov ah, [esi]
xchg ah, al
```

`mov [edi], al` - pro vytvoření iluze závislosti na předchozí hodnotě `[EDI]` ji nejdřív nastavíme na konkrétní hodnotu:

```
mov [edi], al
 ->
mov byte [edi], 5F
xor [edi], al
xor byte [edi], 5F
```

`inc esi` - pro ztížení analýzy použijeme konstantu ze systémového registru, konkrétně `CR0`, jehož spodních 16 bitů (registr `MSW`) můžeme na libovolné úrovni oprávnění číst instrukcí `SMSW`. Využijeme faktu, že bit 0 (Protection Enable) je v chráněném módu vždy nastaven (`MSW` načteme do náhodně vybraného registru, který napřed uložíme):

```
inc esi
 ->
push ebp
smsw bp
and ebp, 1
add esi, ebp
pop ebp
```

`inc edi` - operaci zakryjeme odečtením náhodně vybraného registru a jeho opětovným přičtením pomocí `ADC` s nastaveným `CF`:

```
inc edi
 ->
sub edi, eax
stc
adc edi, eax
```

`cmp al, 0` - nevýhoda instrukce `CMP` je, že nezapisuje do cílového operandu, na rozdíl od `SUB`, takže není tolik možností, jak rozkládat. V tomto případě ale dokáže obfuskátor zjistit, že hodnota `AL` je ve všech možných větvých kódu nakonec přepsána (na návěští `copy` je instrukce `MOV AL` a dál v kódu zase `MOV EAX`), takže `AL` může být před testováním nějak upraven, aby konstanta (0) nebyla tak viditelná. Zase využijeme náhodně vybraný registr:

```
cmp al, 0
 ->
add al, ch
cmp al, ch
```

`jne copy` - `JNE` skočí na návěští, pokud není nastaven `ZF`. Protože ale předchází aritmetická instrukce `CMP`, můžeme výsledek testovat na vyšší nebo nižší pomocí dvou instrukcí `JA` a `JB`. A ještě to vylepšíme inverzí na `JNB` a pomocným skokem:

```
 jne copy
  ->
 ja copy
 jnb finished
 jmp copy
finished:
```

`mov eax, edx` - budeme předpokládat, že obfuskátor ví (např. díky tomu, že zná volací konvenci této funkce), že může přepsat příznaky i registr `EDX` (jde o *volatile* registr). Potom jde tato instrukce rozložit třeba na:

```
mov eax, edx
 ->
xor eax, edx
xor edx, eax
xor eax, edx
```

`pop ebp` - `POP` nejdřív vloží hodnotu do `EBP` a potom přičte k `ESP` 4 bajty. Samotné přičtení provedeme malým trikem pro pobavení.

```
pop ebp
 ->
mov ebp, [esp]
pop dword [esp+4]
```

`ret` - znovu předpokládáme, že obfuskátor zná volací konvenci a tudíž ví, že registry `EAX`, `ECX` a `EDX` může přepsat, a proto jde `RET` nahradit následující sekvencí.

```
ret
 ->
pop ecx
jmp ecx
```

Celá funkce vypadá v tuto chvíli takto:

```
strcpy:
 push 0F4DCE10       ; push ebp
 mov [esp], ebp

 sub esp, ebx        ; mov ebp, esp
 mov ebp, esp
 add ebp, ebx
 add esp, ebx

 mov dx, [ebp+0A]    ; mov edx, [ebp+08]
 shl edx, 10
 mov dh, [ebp+09]
 mov dl, [ebp+08]

 push dword [ebp+0C]; mov esi, [ebp+0C]
 pop esi

 push edx            ; mov edi, [ebp+08]
 lea edx, [ebp+08]
 mov edi, [edx]
 pop edx
```

```
copy:
 mov ah, [esi]      ; mov al, [esi]
 xchg ah, al

 mov byte [edi], 5F ; mov [edi], al
 xor [edi], al
 xor byte [edi], 5F



 push ebp           ; inc esi
 smsw bp
 and ebp, 1
 add esi, ebp
 pop ebp

 sub edi, eax       ; inc edi
 stc
 adc edi, eax

 add al, ch         ; cmp al, 0
 cmp al, ch

 ja copy            ; jne copy
 jnb finished
 jmp copy
finished:

 xor eax, edx       ; mov eax, edx
 xor edx, eax
 xor eax, edx

 mov ebp, [esp]     ; pop ebp
 pop dword [esp+4]

 pop ecx            ; ret
 jmp ecx
```

Prohazování instrukcí

Pro zjednodušení se pokusíme prohazovat pouze sousedící instrukce, ne celé bloky instrukcí:

```
strcpy:
   push 0F4DCE10
A) mov [esp], ebp

B) sub esp, ebx
   mov ebp, esp
   add ebp, ebx
   add esp, ebx
```

Tyto nejde prohodit, protože **A** zapisuje na paměť adresovanou `ESP` a současně **B** mění `ESP`.

```
B) sub esp, ebx
   mov ebp, esp
   add ebp, ebx
C) add esp, ebx
```

```
D) mov dx, [ebp+0A]
D) shl edx, 10
   mov dh, [ebp+09]
   mov dl, [ebp+08]
```

Tyto (**C** a **D**) prohodit jde, protože žádný z operandů instrukcí se nepřekrývá:

```
   sub esp, ebx
   mov ebp, esp
   add ebp, ebx
D) mov dx, [ebp+0A]
D) shl edx, 10
C) add esp, ebx
   mov dh, [ebp+09]
E) mov dl, [ebp+08]

F) push dword [ebp+0C]   ; původně mov esi, [ebp+0C]
   pop esi
```

Protože předpokládáme, že vrchol zásobníku (`ESP`) nekoliduje s `ebp+08` ani s `ebp+0C` (jinak by sme nemohli rozložit původní `MOV` z `[ebp+OC]` na `PUSH`), instrukce můžeme prohodit:

```
   mov dh, [ebp+09]
F) push dword [ebp+0C]
E) mov dl, [ebp+08]
G) pop esi

H) push edx
   lea edx, [ebp+08]
   mov edi, [edx]
   pop edx
```

Instrukce **G** a **H** prohodit nejde, protože obě manipulují se zásobníkem.

```
   push edx
   lea edx, [ebp+08]
   mov edi, [edx]
I) pop edx

  copy:
J) mov ah, [esi]
   xchg ah, al
```

Instrukce **I** a **J** zase prohodit nejde, protože je mezi nimi návěští skoku. Jinými slovy, `POP EDX` se nemůže stát součástí `copy` smyčky, nebo `MOV AH` z ní nemůže vypadnout.

```
   mov ah, [esi]
K) xchg ah, al

L) mov byte [edi], 5F
   xor [edi], al
   xor byte [edi], 5F
```

Přehodit **K** a **L** je možné, žádné z operandů instrukcí spolu nekolidují:

```
   mov ah, [esi]
L) mov byte [edi], 5F
K) xchg ah, al
   xor [edi], al
M) xor byte [edi], 5F

N) push ebp
N) smsw bp
   and ebp, 1
   add esi, ebp
   pop ebp
```

Přehodit **M** a **N** je zase možné. `XOR` sice na rozdíl od `PUSH` a `SMSW` přepisuje příznaky, ale žádná z následujících instrukcí na nich nezávisí.

```
   xor [edi], al
N) push ebp
N) smsw bp
M) xor byte [edi], 5F
   and ebp, 1
   add esi, ebp
O) pop ebp

P) sub edi, eax
P) stc
   adc edi, eax
```

Instrukce **O** a **P** spolu prohodit můžeme, protože `POP` nemění příznaky (`ADC` závisí na příznaku `CF`):

```
M) xor byte [edi], 5F
   and ebp, 1
   add esi, ebp
P) sub edi, eax
P) stc
O) pop ebp
Q) adc edi, eax

R) add al, ch
   cmp al, ch
```

Naopak instrukce **Q** a **R** nejde prohodit, protože `ADD` by změnila `CF`.

```
   add al, ch
S) cmp al, ch

T) ja copy
T) jnb finished
T) jmp copy
  finished:

U) xor eax, edx
   xor edx, eax
   xor eax, edx
```

Žádnou z instrukcí **S**, **T** a **U** přehodit nejde, protože instrukce **T** mění řízení a navíc mezi **T** a **U** je návěští, přes které nelze nikdy instrukce přehazovat.

```
   xor eax, edx
   xor edx, eax
V) xor eax, edx

W) mov ebp, [esp]
W) pop dword [esp+4]
```

Instrukce **V** a **W** žádné z operandů nesdílejí, takže je prohodíme.

```
   xor eax, edx
   xor edx, eax
W) mov ebp, [esp]
W) pop dword [esp+4]
V) xor eax, edx

X) pop ecx
   jmp ecx
```

Instrukce **V** a **X** spolu také můžeme ze stejného důvodu prohodit:

```
W) mov ebp, [esp]
W) pop dword [esp+4]
X) pop ecx
V) xor eax, edx
   jmp ecx
```

A poslední instrukce `JMP` je zase neprohoditelná.

A už to začíná vypadat pěkně:

```
strcpy:
 push 0F4DCE10
 mov [esp], ebp

 sub esp, ebx
 mov ebp, esp
 add ebp, ebx
 mov dx, [ebp+0A]
 shl edx, 10
 add esp, ebx
 mov dh, [ebp+09]
 push dword [ebp+0C]
 mov dl, [ebp+08]
 pop esi

 push edx
 lea edx, [ebp+08]
 mov edi, [edx]
 pop edx

copy:
 mov ah, [esi]
 mov byte [edi], 5F
 xchg ah, al
 xor [edi], al
 push ebp
 smsw bp
```

```
 xor byte [edi], 5F
 and ebp, 1
 add esi, ebp
 sub edi, eax
 stc
 pop ebp
 adc edi, eax

 add al, ch
 cmp al, ch

 ja copy
 jnb finished
 jmp copy
finished:

 xor eax, edx
 xor edx, eax
 mov ebp, [esp]
 pop dword [esp+4]
 pop ecx
 xor eax, edx
 jmp ecx
```

Změna toku řízení

Obfuskátor náhodně vybere části kódu pro přesun. S použitím instrukcí JMP nebývá problém, v případě instrukcí závisejících na zásobníku nesmí být pro přesun použita instrukce CALL, která vkládá na zásobník návratovou adresu. Je vidět, že došlo k docela slušné fragmentaci kódu:

```
strcpy:
 push 0F4DCE10
 jmp moved1   ; instrukce mov [esp], ebp až
back1:        ;  mov dx, [ebp+0A] přemístěny
 shl edx, 10
 add esp, ebx
 mov dh, [ebp+09]
 push dword [ebp+0C]
 mov dl, [ebp+08]
 jmp skip_moved2   ; vytvoř prostor pro přemístěný kód

moved2:
 smsw bp
 xor byte [edi], 5F
 and ebp, 1
 add esi, ebp
 sub edi, eax
 stc
 retn

skip_moved2:
 pop esi
 push edx
 lea edx, [ebp+08]
 mov edi, [edx]
 pop edx
```

```
copy:
 mov ah, [esi]
 mov byte [edi], 5F
 xchg ah, al
 xor [edi], al
 push ebp
 call moved2   ; smsw bp až stc
 pop ebp
 adc edi, eax
 add al, ch
 cmp al, ch
 jmp moved3   ; instrukce ja copy a jnb finished
back3:
 jmp copy

moved1:   ; za původní „jmp copy" šlo bezpečně vložit kód
 mov [esp], ebp
 sub esp, ebx
 mov ebp, esp
 add ebp, ebx
 mov dx, [ebp+0A]
 jmp back1

finished:
 xor eax, edx
 xor edx, eax
 jmp skip_moved3   ; vytvoř prostor pro přemístěný kód

moved3:
 ja copy
 jnb finished
 jmp back3

skip_moved3:
 mov ebp, [esp]
 pop dword [esp+4]
 pop ecx
 xor eax, edx
 jmp ecx
```

Falešné skoky

Obfuskátor přidá jeden nadbytečný podmíněný skok JNC, dva skoky (CALL a JMP) zřetězí a vytvoří několik skoků do instrukce.

Pro prohlížení skoků do instrukce v debuggeru může být potřeba vypnout analýzu, která je odhalí a kód zobrazí správně.

```
strcpy:

 jmp skip_new_byte
 db 0B8          ; vytvoří falešnou instrukci MOV EAX, xxxxxxxx
skip_new_byte:   ;  (pohltí následující čtyři bajty)

 push 0F4DCE10
 jmp moved1
```

```
bebacksoon:   ; za nový „jmp moved1" šlo bezpečně vložit kód
 jmp goingback

back1:
 shl edx, 10
 add esp, ebx
 jmp skip_moved2_chain   ; vytvoř prostor pro meziskok

moved2_chain:
 jmp moved2   ; meziskok

skip_moved2_chain:
 mov dh, [ebp+09]
 push dword [ebp+0C]
 mov dl, [ebp+08]
 jmp skip_moved2

moved2:
 smsw bp
 xor byte [edi], 5F
 and ebp, 1
 add esi, ebp

 jnc bebacksoon   ; vložený podmíněný skok
goingback:        ;  (někdy se provede, někdy ne, nezáleží na tom)

 sub edi, eax
 stc
 retn

skip_moved2:
 pop esi
 push edx

 jmp skip_new_word
 db 05, 12       ; vytvoří falešnou instrukci ADD EAX, xxxxxx12
skip_new_word:   ;  (pohltí následující tři bajty)

 lea edx, [ebp+08]
 mov edi, [edx]
 pop edx
copy:
 mov ah, [esi]
 mov byte [edi], 5F
 xchg ah, al
 xor [edi], al
 push ebp

; call moved2   ; zřetězení skoku
 call moved2_chain


 pop ebp
 adc edi, eax
 add al, ch
 cmp al, ch
 jmp moved3
back3:

; jmp copy   ; zřetězení skoku
 jmp copy_chain
```

```
moved1:
 mov [esp], ebp
 sub esp, ebx

 jmp skip_new_dword
 db 66, 81, 34, 9C    ; vytvoří falešnou instrukci XOR WORD [ESP+EBX*4], xxxx
skip_new_dword:       ;  (pohltí následující dva bajty)

 mov ebp, esp
 add ebp, ebx
 mov dx, [ebp+0A]
 jmp back1

finished:
 xor eax, edx
 xor edx, eax
 jmp skip_moved3

moved3:
 ja copy
 jnb finished
 jmp back3

copy_chain:   ; za nový „jmp back3" šlo bezpečně vložit meziskok
 jmp copy

skip_moved3:
 mov ebp, [esp]
 pop dword [esp+4]
 pop ecx
 xor eax, edx
 jmp ecx
```

Nicnedělající kód

Nakonec obfuskátor vloží nicnedělající kód. Použijeme falešnou proceduru zmíněnou výše, která bude vložená na náhodně určené místo a náhodně volaná. Ještě ji ale vylepšíme tím, že bude jakoby přijímat jeden parametr (který se předává přes zásobník pomocí `PUSH`). Tím bude náhodně zvolená hodnota nebo registr.

```
strcpy:
 jmp skip_new_byte
 db 0B8
skip_new_byte:
 push 0F4DCE10

 push edx             ; náhodný parametr
 call do_nothing_2    ; volání falešné funkce

 jmp moved1

bebacksoon:
 jmp goingback
```

```
back1:
 shl edx, 10
 add esp, ebx
 jmp skip_moved2_chain

moved2_chain:
 jmp moved2

skip_moved2_chain:
 mov dh, [ebp+09]
 push dword [ebp+0C]
 mov dl, [ebp+08]
 jmp skip_moved2

 ; za nový „jmp skip_moved2" šlo bezpečně vložit falešnou funkci

do_nothing_2:
 push ebp
 mov ebp, esp   ; simuluj vytvoření stack frame
 push ecx
 mov ecx, 0     ; nastav na nula, takže
 repe cmpsw     ;  REPE se neprovede ani jednou
 pop ecx
 pop ebp        ; uvolni stack
 retn 4         ; odstraň vstupní DWORD parametr

moved2:
 smsw bp
 xor byte [edi], 5F
 and ebp, 1
 add esi, ebp
 jnc bebacksoon

 push 1E3D22         ; náhodný parametr
 call do_nothing_2   ; volání falešné funkce

goingback:
 sub edi, eax
 stc
 retn

skip_moved2:
 pop esi
 push edx
 jmp skip_new_word
 db 05, 12
skip_new_word:

 lea edx, [ebp+08]
 mov edi, [edx]
 pop edx
copy:
 mov ah, [esi]
 mov byte [edi], 5F
 xchg ah, al
 xor [edi], al
 push ebp
 call moved2_chain
 pop ebp
 adc edi, eax
 add al, ch
 cmp al, ch
 jmp moved3
```

```
back3:
 jmp copy_chain

moved1:
 mov [esp], ebp
 sub esp, ebx
 jmp skip_new_dword
 db 66, 81, 34, 9C
skip_new_dword:
 mov ebp, esp
 add ebp, ebx
 mov dx, [ebp+0A]
 jmp back1

finished:
 xor eax, edx

 push ebp             ; náhodný parametr
 call do_nothing_2    ; volání falešné funkce

 xor edx, eax
 jmp skip_moved3

moved3:
 ja copy
 jnb finished
 jmp back3

copy_chain:
 jmp copy

skip_moved3:
 mov ebp, [esp]
 pop dword [esp+4]

 push 80001           ; náhodný parametr
 call do_nothing_2    ; volání falešné funkce

 pop ecx
 xor eax, edx
 jmp ecx
```

Vyzkoušejte to

Zdrojáky ve flat assembler syntaxi a spustitelné soubory pro Windows i Linux jsou tady. Program nemá pro zjednodušení žádný výstup, takže je potřeba ho natáhnout do debuggeru. Pro ověření správného výstupu stačí provést *step over* první instrukci `CALL` a podívat se do dat, jestli se překopírování povedlo.

strcpy_orig_win.asm, strcpy_orig_win.exe

strcpy_obfus_win.asm, strcpy_obfus_win.exe

strcpy_orig_nix.asm, strcpy_orig_nix

strcpy_obfus_nix.asm, strcpy_obfus_nix

fasm můžete stáhnout tady. Překlad přímo do spustitelné podoby jde prostě přes `fasm source.asm`.

Další metody obfuskace

Spousta metod obfuskace, které jde také automaticky provádět, nebyla zmíněna. Jde třeba o samomodifikující se kód (SMC), kdy kód sám vytváří následující instrukce, takže je téměř nemožné je zjistit statickou analýzou kódu:

```
; původní kód

8BC3              mov eax, ebx
                 smc_me:
03C1              add eax, ecx

; SMC zajistí přidání instrukce ADD dynamicky:

8BC3              mov eax, ebx
66C705xxxxxxxx    mov word [smc_me], 0C103
EB00              jmp $+2
                 smc_me:
0401              add al, 1    ; falešná instrukce, která bude přepsána
```

Skok `jmp $+2` na následující instrukci zajistí, že dojde ke znovunačtení kódu do cache procesoru, jinak by skutečně mohlo dojít ke spuštění původní instrukce `add al, 1`.

Další zajímavou cestou pro zabránění statické analýzy je dynamická alokace paměti pro proměnné. V kapitole o prohazování instrukcí není zmíněno prohazování celých nezávislých bloků instrukcí, jako třeba prohození dvou funkcí.

Další oblíbenou a dnes často používanou ochranou kódu na úrovni instrukcí je tenký virtuální stroj. Představit si ho jde tak, že instrukce nejsou vykonávány přímo, ale vykonává je emulátor. Aby ale nebylo možné snadno získat a analyzovat původní kód, který emulátor spouští, tak jsou vstupní instrukce ve formátu x86 převedeny do virtuálního formátu, který zná jenom emulátor. Pro zjištění formátu instrukcí je potom potřeba analyzovat samotný emulátor. V kombinaci s obfuskací a proměnlivým formátem instrukcí jde o docela slušnou ochranu proti analýze. Virtualizace je ale téma na samostatný článek.

Zpomalení kódu

Jak je vidět, kromě prohazování instrukcí přidávají metody obfuskace spoustu instrukcí. Jak je to ale potom s rychlostí kódu? Na dnešních procesorech, na rozdíl od nějakého 80386, už nehraje samotný počet instrukcí takovou roli, snad jedině ve smyčkách, vykonávaných v řádu miliónů. Důležitějším faktorem je, jak program zachází s datovou cache procesoru, a to obfuskace většinou neovliňuje.

Závěr

Pokud jste zkoušeli volání obfuskované funkce v debuggeru, určitě jste si všimli, že k určení, co vlastně dělá, není potřeba studovat kód, stačí sledovat paměť (analyzovat data). Obfuskace instrukcí není všemocná právě proto, že pracuje jenom na úrovni instrukcí. Jinak řečeno, v první řadě to musí být design, který je těžké analyzovat, obfuskace je až doplňující prvek.

Ideální by bylo mít nástroj, který pochopí, co vstupní kód dělá, a ten stejný algoritmus vygeneruje jednou z mnoha (tisíc, miliónů) možných cest. Implementovat ale něco takového je prakticky nemožné. Zajímavou cestou, jak se k tomu trochu přiblížit, je poskytnout obfuskátoru nějaké abstraktní vzory namísto konkrétního kódu, který by obfuskátor už generoval sám. Toto by ale bylo téma na jiný článek, protože jde o samostatnou metodu

Komentáře

Pokračujte na diskuzním fóru.

Můj kontakt naleznete zde.

Revize

2010-06-21 1.0 Publikace kopie článku pro prielom MazeGen
(formáty dat odpovídají ISO 8601)