MRLAB

Windows applikasiya proqramlaşdırma interfeysinin hook edilməsi

Azərbaycan Respublikası Xüsusi Rabitə və İnformasiya Təhlükəsizliyi Dövlət Xidməti – Kompüter İnsidentlərinə qarşı Mübarizə Mərkəzi -Malware Research Lab – S. Abasov - 4 May 2022

Api hooking (qarmaq) istər zərərvericilər istər təhlükəsizlik proqram təminatları tərəfindən geniş istifadə edilən bir mexanizmdir. Məqalədə bu mexanizm haqqında qısa məlumat verililəcəkdir. Bu mexanizmin daha yaxşı anlamaq üçün ilk öncə api nədir sualına cavab verməliyik.

Windows Application Programming Interface nədir?

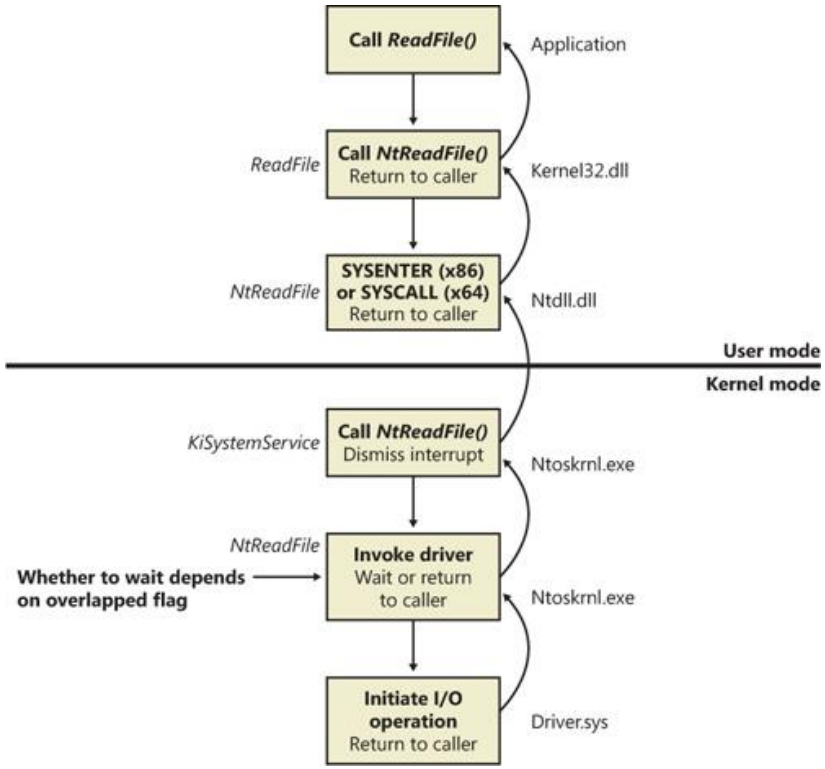
Proqramlaşdırma ilə məşğul olan insanlara yəgin ki, bu termin o qədərdə yad olmayacaqdır. Bəsit dil ilə izah etməyə çalışsaq hər hansı bir A tərəfinin digər tərəfə (B) verdiyi api üzərindən B tərəfi A tərəfinin təqdim etdiyi api üzərindən lazımı əməliyyatları görmək üçün istifadə edir. WinApi – Windows application programming interface eyni məntiq üzərindən fəaliyyət göstərən bir mexanizmdir. Misal olaraq müxtəlif proqramlaşdırma dillərində sistemdə hər hansı bir fayl yaratmaq istədiyiniz zaman həmin dildə yazılan kod, sistemdə fəaliyyət göstərən windows əməliyyat sisteminin təqdim etdiyi api üzərindən bunu icra edə bilir. Və ya hər hansı bir proses haqqında məlumat almaq üçün yenə eyni şəkildə api üzərindən prosesi açmağınız mütləqdir. Bu interfeysi digər proqramlaşdırma dillərinə təqdim edən məhz WinApi adı verilən bu interfeysdır. Yuxarıda qeyd edilənləri pratik olaraq görmək üçün kiçik bir test keçirək. Sistemdə fəaliyyət göstərən hər hansı bir prosesin fəaliyyətini dayandırmaq üçün python dilində kiçik bir kod yazaq.

```
C:\Users\admin\Desktop>tasklist | findstr "notepad"
notepad.exe                  2496 Console                    1      4

C:\Users\admin\Desktop>python
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC v.1600
Type "help", "copyright", "credits" or "license" for more information
>>> from ctypes import *
>>> hProcess = windll.kernelbase.OpenProcess(0x0001, 0, 2496)
>>> windll.kernelbase.TerminateProcess(hProcess, 0)
1
```

Gördüyünüz kimi python ctypes modulu üzərindən 2496 PID (notepad) fəaliyyətini sonlandıra bildik. Digər proqramlaşdırma dillərindədə eyni əməliyyatı oxşar şəkildə etmək lazımdır. Burada **OpenProcess** və **TerminateProcess** funksiyalarını winapi **kernelbase.dll** kitabxanası üzərindən bizə təqdim edir. Bu funksiyaları çağırmadan əməliyyat sistemində fəaliyyət göstərən proseslərə müdaxilə etmək şansımız yoxdur və bu bütün proqramlaşdırma dillərinə aiddir. Lakin məsələ bununlada bitmir. Bu funksiyalarda həmçinin öz növbəsində əməliyyat sisteminin kernelinə müraciət edirlər.Belə ki, OpenProcess funksiyası çağrıldığı zaman arxa fonda **ntdll.dll** kitabxanası içərisində olan **NtOpenProcess** funksiyasına müraciət edilir. Bu funksiyada öz növbəsində kernel səviyyəsinə keçərək (syscall instr) prosesə handle alır və həmin handle dəyərini geri qaytarır. Aşağıda ki, şəkildə hər hansı bir

fayldan məlumat oxumaq istədiyiniz zaman arxa fonda sisteminin nələr etdiyini daha yaxşı görmək mümkündür.



Oxşar əməliyyat biz OpenProcess funksiyasını çağırdığımız zamanda olur. OpenProcess funksiyası özünə gələn parametrləri NtOpenProcess funksiyasına göndərir. **NtOpenProcess** funksiyası isə lazımı sazlamaları edərək kernel(*ring0*) səviyyəsinə keçir. Kernel səviyyəsində prosesin handle dəyəri alındıqdan sonra thread yenidən user(*ring0*) səviyəsinə keçid edir və istifadəçiyə əldə etdiyi handle dəyərini qaytarır. Kernel bu məqalənin mövzusu olmadığı üçün bu hissə bizə maraqlı deyil. Gəlin debugger ilə əməliyyat axınına baxaq. Kernelbase.dll kitabxanasında **OpenProcess** funksiyasına breakpoint (**int3**) set edirik və python da eyni əməliyyatı icra edirik.

```
75EE813C <kernelbase.OpenProcess>   8BFF   MOV EDI,EDI
C:\Python34\python.exe
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18)
Type "help", "copyright", "credits" or "license" for more
>>> from ctypes import *
>>> windll.kernelbase.OpenProcess(0x0001, 0, 2496)
```

Aşağıda ki,şəkildə OpenProcess funksiyasına stack üzərindən gələn parametrleri görürsünüz.

```
Default (stdcall, EBP sta   5
1: [ebp+8] 00000001
2: [ebp+C] 00000000
3: [ebp+10] 000009C0
```

Bu funksiya içərisində bəzi sazlamaları etdikdən sonra NtOpenProcess funksiyası çağrılır.

```
MOV DWORD PTR SS:[EBP-0x1C],ESI
MOV DWORD PTR SS:[EBP-0x18],ESI
MOV DWORD PTR SS:[EBP-0x10],ESI
MOV DWORD PTR SS:[EBP-0xC],ESI
CALL DWORD PTR DS:[<&NtOpenProcess>]
```

Bu funksiya isə EAX registerinə 0xBE dəyərini ataraq carı thread-ı kernel səviyəsinə salmaq üçün KiFastSystemCall funksiyası çağrılır.

```
77CE5910 <ntdll.NtOpenProcess>    B8 BE000000    MOV EAX,0xBE
77CE5915                          BA 0003FE7F    MOV EDX,<&KiFastSystemCall>
77CE591A                          FF12           CALL DWORD PTR DS:[EDX]
77CE591C                          C2 1000        RET 0x10
```

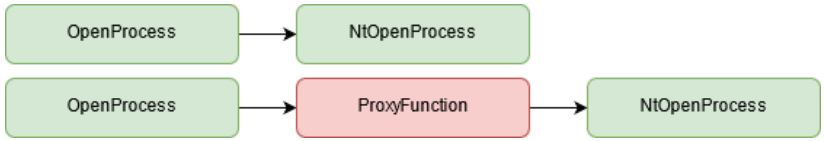
KiFastSystemCall isə sysenter əmri ilə carı threadı kernel səviyyəsinə salaraq sistem servis dispatch table (SSDT) adı verilən tabloda 0xBE indeksinə qarşılıq gələn funksiyanı çağırır.

```
<ntdll.KiFastSystemCal    8BD4    MOV EDX,ESP
                          0F34    SYSENTER
<ntdll.KiFastSystemCal    C3      RET
```

WinApi işləmə prinsipi haqqında qısa məlumat aldıqdan sonra keçirik hooking(qarmaq) mexanizminə.

Hooking yuxarıda qeyd edilən əməliyyat zəncirinə öz halqanızı daxil etmək mexanizminə verilən ümumi addır. Yəni **OpenProcess** funksiyası ilə **NtOpenProcess** funksiyası arasına öz proxy funksiyanızı yerləşdirərək əməliyyat zəncirinə müdaxilə edəbilərsiniz.

Normal və hook edilən əməliyyat zənciri.



Burada əsas məsələ funksiya-ya(OpenProcess vs) gələn parametrləri oxumaq vəya müdaxilə etməkdir.Yəni məsələn zərərli A prosesinin onu **TerminateProcess** ilə terminate etməsini istəmir. Bunun üçün **Proxy** funksiyası içərisində əgər **OpenProcess** funksiyasının özünə handle almaq istədiyini qeydə alar isə bu zaman həmin PID dəyərini var olmayan bir dəyər ilə dəyişir və **OpenProcess** funksiyası zərərli prosesə handle ala bilmir.

Hook əməliyyatını həyata keçirmək üçün hədəf prosesin yaddaşına müdaxilə etmək lazımdır. Bu tip əməliyyatlar zamanı ən effektiv metod dll inyeksiyadır. Proxy funksiyamızı bu dll içərisinə yazaraq dll faylını inyeksiya edirik. Daha sonra hook ediləcək funksiya adresinə bizim proxy funksiyasına tullanacaq(jump) şəklində patch edirik.Məqalədə bunun üçün **jmp** instruction dan istifadə edəcəyik. Lakin başqa instructionlarında köməkliyi ilə eyni işi görmək mümkündür.Ilk olaraq hook funksiyamızı yazıram. Məqalə üçün **OpenProcess** funksiyasına gələn PID dəyərini ekrana yazdıran bir funksiya olacaq.

```
VOID Hook_OpenProcess(DWORD dwDesiredAccess,
    DWORD bInheritHandle,
    DWORD dwProcessId)
{
    printf("Hook funksiyasi PID=%d\n:", dwProcessId);
}
```

Hook funksiyasını yazdıqdan sonra dll faylını, müdaxilə etmək istədiyimiz prosesin yaddaşına inyeksiya edirik.

```
Thread E54 created, Entry: <kernel32.LoadLibraryW>
DLL Loaded: 005B0000 C:\Users\admin\Desktop\HookDll.dll
```

OpenProcess funksiyasını hook edəcəyimiz üçün orijinal **OpenProcess** və **Hook_OpenProcess** funksiyasının adreslərini

öyrənirik. Daha sonra JMP instruction ilə orijinal funksiyanın bizim hook funksiyasına jump edəcək şəkildə patch etməliyik.

0DCE813C <kernelbase.OpenProcess>

005B1040 <hookdll._Hook_OpenProcess>

Burada jmp (0xE9) instruction üçün uzaqlıq məsafəsini hesablamalıyıq. Bunun üçün ilk olaraq hədəf ünvandan cari ünvanı çıxırıq. Daha sonra jmp opcode (+ünvan) ölçüsünü (5 bayt) çıxaraq offsetı tapırıq.

```
>>> from ctypes import *
>>> hook_addr = c_ulong(0x005B1040)
>>> orig_addr = c_ulong(0x0DCE813C)
>>> offset = c_ulong(hook_addr.value - orig_addr.value - 5)
>>> hex(offset.value)
'0xf28c8eff'
```

Offsetimizi aldıqdan sonra keçirik **OpenProcess** funkasiyasının olduğu adresdəki opcode-ları patch etməyə.

```
$  E9 FF8E8CF2   JMP <hookdll._Hook_OpenProcess>
```

Gördüyünüz kimi artıq defolt OpenProcess funksiyası bizim hook funksiyamıza jump edəcək şəkildə patch edilib. Test üçün **OpenProcess** funksiyasını çağırıram.

```
>>> windll.kernelbase.OpenProcess(0x011,0,1149)
Hook funksiyasi PID=1149
```

Gördüyünüz kimi nəticə olaraq **OpenProcess** funksiyası bizim hook funksiyamızı çağırır. Artıq hook funksiyası içərisində defolt funksiyaya istədiyimiz kimi müdaxilə edə bilərik.

Burada bir digər məsələ ortaya çıxır. Bəs hook funksiyasından sonra nə baş verəcək? Əlbətdəki hook funksiyası içərisində lazımı müdaxilələr edildikdən sonra proqram normal axışına davam etməlidir. Yəni hook funksiyasından sonra orijinal OpenProcess funksiyası çağrılmalıdır. Lakin biz orijinal OpenProcess funksiyasında prolog qismində patch əməliyyatı aparmışıq.

```
8BFF    MOV EDI,EDI
55      PUSH EBP
8BEC    MOV EBP,ESP
```

Proqramın normal axışının davam etməsi üçün patch edilən baytların defolt baytlar ilə əvəzlənməsi lazmdır. Bunun üçün patch əməliyyatından öncə patch əməliyyatının ölçüsü qədər (5 bayt) məlumat hook dll içərisində bufferdə saxlanılır. Hook funksiyası çağrıldıqdan sonra və lazımı müdaxilələr edildikdən sonra bu baytlar defolt baytlar ilə əvəzlənir. Daha sonra hook funksiyası içərisindən defolt OpenProcess funksiyası çağrılır. Bundan sonra isə **OpenProcess** funksiyası hook funksiyasını çağıracaq şəkildə yenidən patch edilir. Bura qədər hook (qarmaq) mexanizminin işləməsi haqqında məlumat aldınız. Lakin bir hook engine (qarmaq mühərriki) yaratmaq bu qədər sadə deyil.

Burada bir çox şeylərə diqqət yetirmək lazımdır. Xüsusi ilə thread təhlükəsizliyi, trampoline funksiyaları, opcode uzunluğu və s.

İstinadlar

https://en.wikipedia.org/wiki/Windows_API
https://docs.microsoft.com/en-us/windows/win32/api/
https://attack.mitre.org/techniques/T1056/004/
https://github.com/TsudaKageyu/minhook
https://www.codeproject.com/Articles/44326/MinHook-TheMinimalistic-x-x-API-Hooking-Libra