

SOPS - Laborator 7 - ROP

1. Introducere

În laboratorul trecut am văzut cum, prin intermediul vulnerabilității buffer overflow, putem pune propriul nostru cod malițios pe stivă și cum îl putem executa. Totodată am observat ce mecanisme de protecție împiedică atacul. Desigur, cel care ne-a pus cele mai mari probleme a fost ASLR. În laboratorul de astăzi vom învăța o nouă tehnică de a monta atacuri numită Return Oriented Programming (ROP).

Principiul este asemănător cu cel utilizat în laboratorul trecut. Vom avea nevoie de un program vulnerabil care să aibă o vulnerabilitate de tip buffer overflow. Diferența majoră față de atacul precedent este faptul că nu vom mai injecta codul nostru malițios pe stivă, ci îl vom compune, asemenea unui puzzle, din bucăți de cod legitime care sunt prezente deja în programul vulnerabil. Astfel pe stivă vom pune adresele acestor instrucțiuni speciale și cu ajutorul lor vom monta atacul. Aceste bucăți speciale de cod sunt numite **gadget-uri**.

2. Gadget-uri

Gadget-urile sunt bucăți de cod scrise în limbaj de asamblare care efectuează operații elementare. Dintre instrucțiunile scrise în limbaj de asamblare care se găsesc atunci când este dezasamblată o bibliotecă, se remarcă niște instrucțiuni numite folositoare de către atacatorul care vrea să conceapă un atac de tip „Return Oriented Programming”. Pentru un atacator care vrea să monteze un astfel de atac, instrucțiunile folositoare sunt acele instrucțiuni terminate în ret (bitul c3) deoarece acestea presupun executarea instrucțiunilor și întoarcerea la contextul anterior apelării instrucțiunii. Mai precis, atacatorul pune adresele acestor instrucțiuni pe stivă, iar după ce o instrucțiune este executată, controlul este redat pe stivă și următoarea instrucțiune este executată (instrucțiunea *ret* este un alias pentru succesiunea de instrucțiuni *pop \$eip; jmp \$eip*). Punând pe stivă adresele mai multor gadget-uri se formează un lanț de gadget-uri numit și ROP chain. Găsirea de gadget-uri nu este un lucru ușor deoarece depinde de biblioteca în care acestea sunt căutate.


Aceste gadget-uri apar din cauza faptului că instrucțiunile pe arhitectura x86 nu sunt aliniate. Astfel dintr-o instrucțiune perfect validă și inofensivă (poate adăugată de compilator) putem sări la o adresă de memorie din interiorul ei și din byte-i interpretați astfel, să rezulte o nouă instrucțiune care să manipuleze regiștrii spre exemplu.

Interesant este și modul în care aceste gadget-uri sunt găsite. Gadget-urile sunt căutate în secțiunile executabile ale bibliotecii. Fiecare astfel de secțiune are o adresă virtuală, o dimensiune și un deplasament de care trebuie să se țină cont atunci când instrucțiunea este decodată. În caz contrar, adresa de memorie la care se află instrucțiunea și care va fi furnizată către atacator va fi incorectă, iar atacul nu va avea succes. Deoarece nu orice secvență de biți reprezintă o instrucțiune folositoare și cu sens, în procesul de căutare se urmărește ca în primă etapă să se găsească biți c3 corespunzători instrucțiunilor ret. După ce biții c3 sunt găsiți, se analizează instrucțiunea compusă dintr-un număr de biți anteriori (număr ales de atacator) care are c3 ca ultim byte. Această instrucțiune se decodează, iar dacă reprezintă o instrucțiune cu sens, se poate constata că s-a găsit un gadget. Iată cum, prin analiza unei instrucțiuni aparent inofensive, plecând de la byte-ul al treilea, se poate obține un gadget:

```

B8 13 00 00 00  mov $0x13, %eax
E9 C3 F8 FF FF  jmp 3aae9

```



```

00 00  add %al, (%eax)
00 E9  add %ch, %cl
C3     ret

```

3. Configurarea mediului

Vom utiliza un tool special pentru căutarea gadget-urilor numit ROPgadget (<https://github.com/JonathanSalwan/ROPgadget>). Vom avea nevoie de manager-ul de pachete de la python pe care îl vom instala cu comanda

```
sudo apt-get install python-pip
```

Pentru dezasamblarea fișierelor binare, ROPgadget are nevoie de o dependență pe care o instalăm cu comanda

```
sudo pip install capstone
```

Acum suntem pregătiți să instalăm ROPgadget:

```
pip install ropgadget
```

4. Căutarea gadget-urilor

În cele ce urmează vom analiza biblioteca standard de C care este inclusă în cam toate programele C. Dacă utilizați ca sistem de operare Linux, cel mai probabil biblioteca standard de C libC-2.26.so (2.26 este versiunea bibliotecii – puteți avea o versiune diferită) se găsește în */lib32* pentru versiunea bibliotecii pe 32 de biți.

Practic: Analizați librăria standard de C redirecționând rezultatul într-un fișier. Biblioteca poate fi analizată folosind comanda

```
ROPgadget --binary /lib32/libc-2.26.so > gadgets.txt
```

Observați gadget-urile găsite. Ce ar putea reprezenta adresele de la începutul fiecărui gadget (având în vedere faptul ca sunt mici și diferite față de o adresă din programul nostru) ?

5. Pregătirea atacului

Astăzi vom încerca să montăm atacul din laboratorul precedent compunând shellcode-ul din gadget-uri și formând un ROP chain cu ajutorul căruia să lansăm un shell. Practic, în loc să rescriem adresa de întoarcere a funcției cu adresa shellcode-ului de pe stivă, o vom rescrie cu adresa primului gadget. Datorită „ret-ului implicit” dat de mecanismul de apelare al funcțiilor, lanțul de gadget-uri își va începe execuția. După ce primul gadget va fi executat, datorită instrucțiunii *ret*, adresa următorului gadget va fi luată de pe stivă și gadget-ul va fi executat. Astfel se execută lanțul de gadget-uri.

Deoarece în montarea atacului suntem constrânși de ce gadget-uri găsim, pentru simplitate am ales să modific puțin codul vulnerabil din laboratorul trecut astfel încât acesta să utilizeze *gets* pentru a citi de la tastatură input-ul. Astfel nu trebuie să ne mai facem griji de eventuali byte-i null-i din input. Programul vulnerabil este următorul:

```
#include <stdlib.h>
#include <stdio.h>
void func()
{
    char buffer[128];
    gets(buffer);
}
int main(int argc, char *argv[])
{
    func();
    return 0;
}
```

Îl vom compila cu protecția Stack Canary dezactivată:

```
gcc -g -m32 -fno-stack-protector -o overflow overflow.c
```

Deoarece mecanismul ASLR implică și încărcarea bibliotecilor la adrese diferite la fiecare rulare, va trebui să îl dezactivăm cu comanda

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Apelul de sistem care va fi făcut în urma atacului este `execve(„/bin/sh”, NULL, NULL)`, deci va trebui să compunem un lanț de gadget-uri care să facă acest apel. Vom începe prin a încărca în regiștrii `$ecx` și `$edx` ultimele două argumente ale apelului de sistem (valorile NULL) pentru că nu avem argumente pentru comanda `„/bin/sh”` și nici variabile de mediu. Pentru aceasta se caută în fișierul `kernel/proc/sys/kernel/randomize_va_space` cu gadget-uri instrucțiuni de tipul `pop reg ; ret` pentru fiecare din cei doi regiștrii. Cel mai des se găsesc în biblioteca de C împreună ca un singur gadget `pop ecx; pop ebx; ret`.

Aceste adrese se pun pe stivă împreună cu adresa ce reprezintă NULL (0x0). Putem face acest lucru deoarece funcția `gets` nu termină citirea atunci când întâlnește byte-i null-i. Astfel, după fiecare instrucțiune `pop` se pune pe stivă adresa nulă. După ce ultimele două argumente au fost încărcate în regiștrii, trebuie încărcat în registrul `$ebx` primul argument ce reprezintă calea către programul care lansează linia de comandă. Calea reprezintă șirul `„/bin/sh”` a cărei adresă va trebui pusă pe stivă.

Pentru a găsi adresa string-ului `„/bin/sh”` vom căuta dacă nu cumva acesta este și el prezent în biblioteca standard. Mai precis, vom rula programul cu `gdb`, vom pune un breakpoint în main, vom rula și cu ajutorul comenzii

```
p system
```

care va da adresa unde se află funcția `system()` și a comenzii

```
find $1, 9999999999, "/bin/sh"
```

care va căuta string-ul `"/bin/sh"` începând la adresa lui `system` și continuând până la final (un număr foarte mare), vom putea identifica adresa string-ului. Este posibil ca această comandă să nu funcționeze pe anumite sisteme de operare. Ca alternativă puteți instala un plugin de `gdb` numit `gdbpeda`:

```
git clone https://github.com/longld/peda.git ~/peda
```

```
echo "source ~/peda/peda.py" >> ~/.gdbinit
```

Va trebui să pornim iar programul cu `gdb`, să punem un breakpoint în main, să dăm `run`, și apoi comanda

```
searchmem „/bin/sh”
```

Acum se va pune pe stivă adresa a unei instrucțiuni `pop $ebx ; ret` urmată de adresa string-ului `„/bin/sh”` găsită anterior. Acum că parametrii au fost încărcăți în regiștrii, mai rămâne doar să încărcăm în registrul `$eax` identificatorul apelului și să inițiem apelul.

Identificatorul pentru `execve` este 11 (poate fi găsit printr-o scurtă căutare în fișierele de tip header asociate bibliotecii de C, dar nu face obiectul laboratorului nostru). Pentru a încărca valoarea în registrul `$eax` va trebui să ne asigurăm că valoarea stocată în `$eax` este 0. Pentru aceasta vom folosi instrucțiunea `xor $eax, $eax ; ret` care aplică operația pe biți XOR registrului `$eax`, celălalt argument fiind el însuși. Instrucțiunea are ca efect încărcarea valorii 0 în registrul `$eax`. Urmează ca în `$eax` să încărcăm valoarea 11, acest lucru făcându-se adunând 11 (0xb) la valoarea din registrul `$eax` cu ajutorul instrucțiunii `add $eax, 0xb`. Adresele celor două instrucțiuni se vor pune pe stivă. Ultimul pas pentru încheierea atacului este inițierea apelului de sistem. Aceasta se face cu ajutorul instrucțiunii `int 0x80` a cărei adresă trebuie pusă pe stivă.

Combinând toate aceste adrese puse pe stivă obținem inputul malițios necesar atacului, care adăugat la un șir de caractere poate reprezenta încărcătura pentru atac exploatănd vulnerabilitatea de tip buffer overflow.

Acum că avem toate elementele, putem să ne gândim la montarea atacului. Stiva după atac ar trebui să arate așa:

```
//higher memory
+-----+
| INT0x80      | syscall should be "execve( "/bin/sh",0,0)
+-----+
| add eax, 0xb ; ret | add 0xb to EAX (to call execve with 11)
+-----+
| xor eax,eax ; ret | ensure EAX is 0
+-----+
| ptr to "/bin/sh/" |
+-----+
| \x00\x00\x00\x00 |
+-----+
| pop $ecx,pop $ebx; ret | load ECX with NULL and EBX with 'bin/sh'
+-----+
| \x00\x00\x00\x00 |
+-----+
| pop $edx, ret | load EDX with NULL
+-----+
| EBP = "BBBB" | Our overflow
+-----+
| filler A's |
+-----+
//lower memory
```

6. Montarea atacului

Practic: Căutați în fișierul rezultat la task-ul 1 adresele gadget-urilor necesare, precum și adresa string-ului „/bin/sh” și notați aceste adrese.

Gadget-urile se pot găsi ajutându-ne de fișierul de la primul task și de comanda grep

```
cat [ fișier ] | grep „[ gadget ]”
```

Exemplu: cat result.txt | grep „xor eax, eax ; ret”

Adresele găsite reprezintă adresele relative la începutul bibliotecii și de aceea, pentru ca atacul să funcționeze, va trebui să adunăm la adresa fiecărui gadget găsit, adresa unde este încărcată biblioteca în programul vulnerabil (nu este necesar să facem această operație și pentru adresa string-ului „/bin”). Puteți folosi calculatorul din laboratorul trecut:

<https://www.calculator.net/hex-calculator.html>

Pentru a afla unde este încărcată biblioteca standard de C vom folosi informațiile date de Linux despre bibliotecile încărcate dinamic ale unui proces. Va trebui să executăm programul vulnerabil cu ./overflow, iar dintr-un alt terminal vom vedea informații despre procesul care rulează programul vulnerabil cu comanda

```
ps -aux | grep overflow
```

Ne uităm la a doua coloană și luăm pid-ul procesului, iar cu comanda

```
cat /proc/[ pid ]/maps
```

putem vedea adresele de început și sfârșit ale tuturor bibliotecilor încărcate dinamic. Aici va trebui să căutăm biblioteca standard de C (libc-2.26.so - vor fi afișate mai multe, dar pe noi ne interesează prima) și să reținem adresa de început (prima adresă).

Practic: Încercați să montați atacul, punând în input adresele gadget-urilor găsite alături de adresa string-ului „/bin/sh” în ordinea potrivită (formând un ROP chain) dată de imaginea stivei de mai sus. Trebuie, deci, să faceți overflow și să rescrieți adresa de întoarcere cu adresa primul gadget din lanț. Adresele vor fi puse în input exact ca în laboratorul precedent. Nu uitați să adunați la toate adresele (mai puțin la adresa string-ului), adresa unde este încărcată biblioteca de C.

Pentru a putea rula programul cu input malițios, input-ul va fi dat prin pipe programului vulnerabil (cat - este folosită pentru a ține shell-ul deschis).

Pentru a evita problemele cu byte-i null-i de la extinderea comenzii în bash, vom scrie input-ul într-un fișier și apoi din fișier îl vom da programului.

Exemplu: `python -c '...' > fisier`

`(cat fisier; cat -) | ./overflow`

Comanda va arăta cam așa:

```
python -c 'print 136*"A" + 4*"B" + [addr pop edx; ret] + „\x00\x00\x00\x00” + [addr pop ecx; pop ebx; ret] + „\x00\x00\x00\x00” + [addr „/bin/sh”] + [addr xor eax, eax; ret] + [addr add eax, 0xb; ret] + [addr int 0x80]' > fisier
```

`(cat fisier; cat -) | ./overflow`