Sécurité informatique : Sécurité et logiciels

Matthieu Amiguet

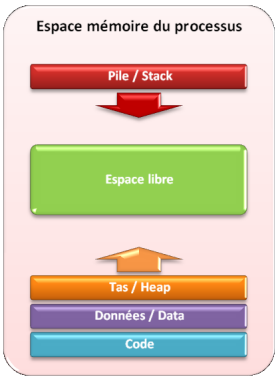
2006 – 2007

haute école arc ingénierie
neuchâtel berne jura — saint-imier le locle porrentruy

Débordement de tampon 2

- On parle de *débordement de tampon* lorsqu'une zone mémoire réservée pour une variable est remplie avec un contenu *trop long* pour cette zone mémoire
- En : "Buffer overflow"
- Le contenu déborde donc de la zone qui lui est attribuée
- Dans la plupart des cas, ceci va provoquer un plantage du programme (erreur d'accès mémoire)
- Dans certains cas, on peut *exploiter* cette faille pour exécuter du code arbitraire
- Le débordement de tampon est, encore maintenant, une des premières causes de faille de sécurité du logiciel.

Rappel : organisation de la mémoire 3



Rappel : organisation de la mémoire – suite 4

- Certaines zones mémoire peuvent être attribuées à la compilation (code, variables globales, . . .)
- Le reste est attribué dynamiquement dans deux structures : la pile (*stack*) et le tas (*heap*)
- La pile :
  - Essentiellement utilisées pour stocker les données locales à une procédure
  - Gestion automatique par le processeur et/ou le système d'exploitation
- Le tas :
  - Espace disponible pour les données dynamiques du programme
  - Gestion "à la main" (malloc/free. . .).

Fonctionnement de la pile (architecture x86) 5

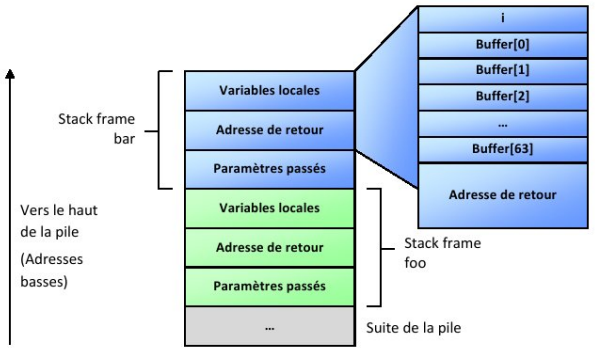
- Lors d'un appel de procédure, on empile, dans l'ordre :
  - Les paramètres passés à la procédure
  - L'adresse de retour
  - L'espace pour les variables locales
- Le tout forme un "Stack frame"

```
int foo(char c)
{
    float x, y;
    ...
    bar(x, y);
}

void bar(float x, float y)
{
    int i;
    char Buffer[64];

 → ...
}
```


Fonctionnement d'un appel (architecture x86) – suite 6



Débordement sur la pile 7

- Si on remplit la variable `buffer` avec une chaîne trop longue, on va écraser l'adresse de retour
- La plupart du temps, cette adresse pointera en-dehors de la mémoire adressable, ou vers une zone non-autorisée et cela provoquera une erreur d'accès mémoire
- Dans certains cas, on pourra sauter vers une zone "plausible". La suite de l'exécution sera donc complètement erratique. . .
- . . . sauf si on s'est arrangé pour sauter vers un endroit qui exécute du code "maîtrisé" !

Code vulnérable 8

```
#include <string.h>

void foo(char *str)
{
  char buffer[32];

  strcpy(buffer, str);

  /* ... */
}
int main(int argc, char *argv[])
{
  if (argc > 1) {
    /* appel avec le premier argument de la ligne de commandes */
    foo(argv[1]);
  }

  /* ... */

  return 0;
}
```

Exploitation 9

- Comment le code ci-dessus peut-il être utilisé pour exécuter du code arbitraire ?
- Il faudrait pouvoir écraser l'adresse de retour avec une adresse dont on maîtrise le contenu...
- ... mais les espaces d'adressages des processus étant distincts, il faut que cette zone soit dans le processus courant...
- ... On va donc utiliser le `buffer` lui-même !
- Il "suffit" donc de concevoir le `argv[1]` du programme ci-dessus de manière à ce que
  - Il écrase l'adresse de retour avec l'adresse du début du buffer
  - Le début du buffer contient le code que l'on veut exécuter.

Shellcode 10

- Traditionnellement, le code "injecté" dans le processus servait à lancer un "shell" (souvent `/bin/sh`)
  - L'avantage est qu'il est lancé avec les droits du processus attaqué
- Par extension, les codes utilisés dans un débordement de tampon s'appellent souvent *shellcodes*
- Il s'agit d'une chaîne contenant la forme *compilée* du code à exécuter
- "Petite" contrainte : un shellcode ne peut pas contenir l'octet `0x00` (pourquoi ?)
  - Pour cette raison et par "discrétion", les shellcodes sont souvent codés par un XOR octet par octet.
- On prévoit parfois une "zone d'atterrissage" de NOP's

Exemple 11

- Un shellcode ultra-classique pour Linux/x86 : lancement de `/bin/sh` par appel de l'interruption `0x80`
- ```
  char shellcode[] = "
  \xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46
  \x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e
  \x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8
  \x40\xcd \x80\xe8\xdc\xff\xff\xff/bin/sh";
  ```


Quelques protections contre l'exploitation 12

- Programmation
  - Utilisation de langages de haut niveau (Java, C#, Python, ...)
  - Utilisation de fonctions sûres (`strncpy`, ...)
- Compilation
  - Utilisation de "canaris" sur la pile (StackGuard, ProPolice, MS Visual Studio 7, ...)
- OS
  - Randomisation de l'espace d'adressage (OpenBSD, Pax (Linux), Exec Shield (Linux), Windows Vista, ...)
- Matériel
  - Prévention de l'exécution de certaines zone mémoires (SPARC, Alpha, PPC, Pentium IV et M, ...)
  ```

Débordement sur le tas 13

- Le débordement de tampon sur la pile est le plus courant et le plus facile à exploiter
- C'est aussi le plus "portable"
- Dans certains cas, on peut également exploiter un débordement de tampon sur le tas
  - Plus délicat à mettre en place
  - Plus dépendant du système d'exploitation, voire de sa version

Integer overflow 14

- En langage de bas niveau (C, . . .), aucun test n'est fait lors de la manipulation d'entiers
- Exemple de principe

```
len1 = strlen(s1);
len2 = strlen(s2);
s3 = (char*)malloc(len1+len2);
memcpy(s3, s1, len1);
memcpy(s3+len1,s2,len2);
```

- Ce code a l'air correct, mais si s1 et s2 sont *très* longs, il peut provoquer un débordement de tampon
  - Par exemple, si on arrive à soumettre des chaîne non terminées par `\0`...

Validation des entrées utilisateur 15

- Dans l'exemple précédent, un utilisateur malicieux pourrait faire planter le programme
- Supposons qu'il s'agisse d'un programme serveur, et que l'utilisateur y accède par le biais d'un client
  - Les validations sont faites au niveau du client. . . donc tout doit être en ordre
  - Mais. . . si l'utilisateur contourne le client et envoie des données directement, on peut avoir un problème !
- De manière générale, et surtout dans les applications client-serveur, *toutes* les entrées utilisateur doivent être validées (sur le serveur) !

Exemple : injections SQL 16

- Considérons un site web dynamique où la connexion d'un nouvel utilisateur est gérée par la requête SQL suivante :

```
SELECT user_id WHERE user_name = '$userName' AND
user_password = '$hashedPass'
```

- En temps normal :

```
SELECT user_id WHERE user_name = 'dupont' AND
user_password = '4EF359675FF'
```

- Mais si l'utilisateur, dans le champ userName, a entré `dupont'---` , la requête devient

```
SELECT user_id WHERE user_name = 'dupont'--- ' AND
user_password = 'peu importe !'
```

- On peut donc se connecter sans connaître le mot de passe de dupont !

Exemple : injections SQL – suite

- On peut faire plus méchant :
  - ```
    SELECT email, passwd, login_id, full_name
    FROM members
    WHERE email = '$email';
    ```
  - `email="x'; DROP TABLE members; --"...`

  ⇝
  ```
  SELECT email, passwd, login_id, full_name
  FROM members
  WHERE email = 'x'; DROP TABLE members; --';
  ```
- On peut de même récupérer des informations, modifier la base de données, ...

Injections SQL : prévention

- Filtrer les entrées utilisateurs
  - échapper les guillemets
  - ... mais ça ne suffit pas !
- Utiliser des *frameworks* de plus haut niveau
- Séparer les privilèges d'accès à la base de données
- Utiliser les procédures stockées au lieu des requêtes SQL
- ...

Virus

- Un virus est un (bout de) programme capable de se *reproduire* lui-même en contaminant d'autres programmes (par remplacement ou ajout de code)
- La plupart des virus ont, à côté de leur fonction auto-reproductive, une "charge utile"
  - plaisanterie (affichage de messages, ...)
  - malveillance pure (effacement de fichiers, ...)
  - chantage (cryptage de disque, ...)
  - prise de contrôle (botnets, ...)
  - ...

Détection

- Il est très difficile de dire, en observant un binaire, si celui-ci est auto-reproducteur ou non
  - Au fait, on peut montrer que le problème est indécidable...
- La détection directe et générique de virus est donc impraticable
- On peut par contre essayer de reconnaître des virus identifiés
  - Si on connaît son code binaire, il devrait suffit de faire une comparaison...
  ```

Virus : techniques de discrétion 21

- Furtivité : se mettre là où on aurait jamais l'idée d'aller les chercher (secteurs déclarés défectueux, modifications de la table d'allocation des fichiers, . . .)
- Polymorphisme : s'arranger pour que le code ne soit pas toujours le même
  - par *réécritures* de bouts de code
  - par *chiffrage* du code compilé (avec un *loader* qui s'occupe de le décrypter. . .)
- . . .

La faille : signatures 22

- Un virus doit éviter d'infecter plusieurs fois le même fichier (pour éviter de se faire repérer. . .)
- Il doit donc pouvoir savoir facilement s'il a déjà infecté un fichier
- Pour ce faire, il y intègre généralement une *signature*
- C'est cette signature, souvent, qui va permettre à l'anti-virus de le repérer !
- Autres techniques :
  - Repérer les codes suspects
  - Repérer les comportements suspects
  - Contrôles d'intégrité des fichiers
  - . . .

Vers 23

- Un ver est, comme un virus, un code auto-reproducteur
- La différence est le support
  - Un virus utilise d'autres fichiers exécutables
  - Un ver se propage tout seul à travers des fonctionnalités réseau (en exploitant une faille de sécurité)
- Cette caractéristique peut leur permettre une propagation extrêmement rapide
- Exemple : en 2000, le vers ILOVEYOU s'est répandu en quelques heures sur probablement des millions de machines à travers le monde. . .

Gentils virus ? 24

- Certains chercheurs proposent d'utiliser les techniques virales *au profit* de la sécurité informatique. . .
- Par exemple, un vers utilisant une faille de sécurité. . . pour la patcher !
- De telles techniques posent de nombreuses questions
  - Sur le plan légal
  - Sur l'acceptation par l'utilisateur
  - Sur la possibilité à maîtriser le phénomène
  - . . .

Malwares 25

- Il existe de nombreux types de logiciels qui introduisent plus ou moins volontairement des failles dans le système où ils s'exécutent
- Si le logiciel fait quelque chose d'utile mais qu'un point d'accès a été introduit (ou oublié), on parle de *porte dérobée* (ou *backdoor*)
- Si le logiciel *semble* faire quelque chose d'utile (ou de "sympa") mais que son but premier est d'introduire une faille, on parle de *Cheval de Troie* (*Trojan Horse* ou *Trojan*)
- Il existe toute une palette de logiciels malicieux (*malwares*) avec différents buts
  - publicité (*adwares*)
  - récupération d'information (*spywares*)
  - ...

Techniques 26

- À première vue, ces différents logiciels ne demandent pas de techniques particulières
- On joue souvent sur "l'ingénierie sociale" pour obtenir le lancement du programme
- Il est donc beaucoup plus facile d'écrire un Cheval de Troie qu'un virus, par exemple
- Quant aux portes dérobées dans un gros logiciel compilé, elles sont pratiquement indétectables
- Un cas intéressant (techniquement parlant) est celui des portes dérobées dans les logiciels *open source*...

Exemple 1 : le noyau Linux 27

- En 2003, un inconnu a réussi à introduire une petite modification dans la source officielle du noyau Linux

```
if ((options == (__WCLONE|__WALL))
&& (current->uid = 0))
retval = -EINVAL;
```

- Ce code a priori inoffensif permet de gagner un accès administrateur sur la machine concernée
- La modification a été découverte avant distribution des nouvelles sources et le problème a pu être évité... en tout cas cette fois-là !

Trusting Trust 28

- En 1984, Ken Thompson propose une technique d'introduction de porte dérobée particulièrement astucieuse (et qui a le mérite de soulever le problème de la *confiance*...)
- Première étape
  - On modifie un compilateur pour qu'il identifie les fois où il compile le code du *login* et qu'il y introduise une porte dérobée
  - Si on étudie le code du *login*, il paraît donc sain, même si le binaire ne l'est pas...
  - ... mais une analyse du code du compilateur révèle le pot aux roses

Trusting Trust – suite 29

- Deuxième étape
  - On modifie à nouveau le compilateur pour qu'il identifie les fois où *il se compile lui-même*
  - Dans ce cas, il introduit le code de la première étape s'il n'y est pas encore
- Troisième étape
  - On compile le compilateur précédent, et on retire tout code suspect de sa source
- Résultat
  - Il n'existe plus nulle part de code suspect
  - Toute recompilation du *login* introduira la porte dérobée
  - Toute recompilation du compilateur produira un compilateur infecté

Ingénierie inverse 30

- Pour différentes raisons, on peut être amené à désassembler un programme compilé pour voir ce qu'il fait
  - Audit de sécurité
  - Analyse de code malicieux
  - Interopérabilité
  - "Craquer" le logiciel
  - Espionnage industriel
  - ...
- Ce processus peut être plus ou moins légal suivant les pays et les produits
- On l'appelle généralement *ingénierie inverse* (*reverse engeneering*)

Obfuscation de code 31

- Pour se protéger, certains éditeurs de logiciels tentent de rendre difficile le désassemblage et/ou la compréhension du code
- Cette technique s'appelle *obscurcissement / assombrissement / obfuscation de code* (*code obfuscation*)
- Elle peut se faire
  - Par modification du code source
  - Par modification du code compilé
  - Par cryptage du code compilé
  - ...
- Il existe des logiciels qui obscurcissent automatiquement un code compilé donné : les *packers*
  - Tout ce qui est fait automatiquement peut être défait automatiquement...

Étude de cas 32

- Un bon exemple de protections contre l'ingénierie inverse est celui de Skype
- cf. l'article de Desclaux&Biondi, MISC n°30, mars-avril 2007.