Emeric STUDER

Introduction aux buffer overflow, CS335

http ://www.phrack.org/archives/49/P49-14

1. Smashing the stack for fun and profit by Aleph One November 1996

Aleph1 United Kingdom : www.aleph1.co.uk
Mail : aleph1@SECURITYFOCUS.COM

Cet article a été rédigé et publié par Aleph1 en Novembre 1996 dans le « magasine » Phreak. Phreak est l'ezine le plus connus en hackinf, phreaking et carding. Les premiers articles datent de 1985, le dernier de 2005 (jusqu'à la prochaine team). Cet article est tiré du 49è opus. Il n'est pas récent, mais c'est l'article principal sur le sujet. En 1996 les buffers overflow étaient déjà connus, mais en général, ils ne faisaient que des erreurs de segmentation et au pire crashaient la machine. Aleph One a été le premier à écrire un document sur la possibilité de prendre le contrôle d'une machine grâce aux erreurs présentent dans les applications. A cette époque, l'époque de nt4 entre autre, très peu de failles étaient exploités. A cela deux raison, la première, peu de personne s'y intéressait vraiment, en général le social engineering étant plus performant. La deuxième était la faible proportion de systèmes interconnectés. En effet, c'est après 1998 qu'Internet a vraiment explosé, et avec lui, toute la scène. C'est après l'an 2000 que les dépassements de tampons ont vraiment connus leurs plus belles heures. En effet, dès qu'un groupe spécialisé dans le domaine découvrait une de ces failles, tout une machinerie étaient en place pour créer un « proof of concept » voir même un ou plusieurs exploits. Les plages d'adresses intéressantes (universités, grosses entreprise etc.) étaient alors scannées et attaquées. On peut citer par exemple le buffer overflow de février 2004 qui a été exploité en masse sur tous les systèmes Microsoft Windows, même le tout dernier né, 2003 serveur sensé être protégé de ce type d'attaque. (L'overflow provenait d'une mauvaise gestion des requêtes RPC sur LSASS) Microsoft n'a sorti un bulletin de sécurité qu'en avril et 17 jours plus tard, plusieurs vers utilisaient les sources rendues publiques afin d'infecter un maximum d'ordinateur. La plupart des systèmes étant vulnérables, les vers se sont multipliés et propagés à une vitesse folle. Il est certain que les vulnérabilités liées au codage sont les failles les plus dangereuses actuellement.

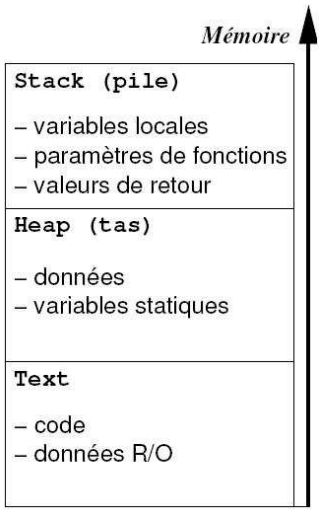
Aleph One Limited

Aleph One, est actuellement employé par la société security focus très connue dans la sécurité informatique. Cette société est la plus complète sur les alertes de sécurités des systèmes informatiques.

2. Synthèse de l'article :

Définitions :
Débordement de buffer (« buffer overflow ») : il est possible d'écrire dans un tableau déclaré auto plus loin que sa taille réelle.
Corruption de pile (« stack smashing ») : un débordement de buffer peut corrompre la pile d'exécution et changer l'adresse de retour d'une fonction.

Organisation des processes dans la mémoire :
Les processes sont divisés en trois régions: Text, Data et Stack.
La région du texte (Text) est divée par le programme et inclut du code (instructions) et des données en lecture seule. Cette région correspond à la section texte d'un exécutable. Elle est normalement marquée en lecture seule et toute tentative d'écriture résulterait dans une violation de segmentation.
La région des données (Data) contient des données initialisées ou non.
Les variables statiques sont stockées dans cette région. La région des données correspond aux sections data-bss d'un exécutable.



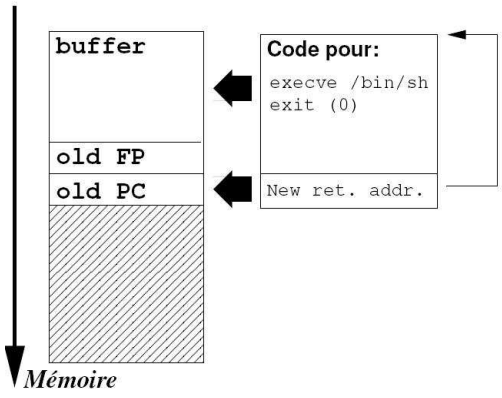
La stack :
La région stack : Une pile est un bloc contigu de mémoire contenant des données. Il y a un pointeur indiquant le début de la pile et un autre pointeur indiquant le sommet de la pile (stack pointer : SP). Elle est entre autre utilisée pour allouer dynamiquement les variables locales utilisées dans les fonctions, passer des paramètres aux fonctions, et retourner des valeurs en sortant des fonctions. Une pile d'objets a la propriété que le dernier objet empilé sera le premier objet dépilé. Cette propriété est souvent décrite comme "last in, first out", ou LIFO. Selon l'implémentation, la pile se développera vers les adresses les plus basses ou les plus hautes. Sur beaucoup de processeur y compris les Intel, Motoroma, SPARC et MIPS la pile se développe vers les adresses basses.

Appel d'une fonction :
Avant l'appel d'une fonction, l'adresse de l'instruction suivant (adresse de retour) ainsi que les valeurs contenues dans les différents registres sont sauvegardées dans la pile afin de restaurer l'environnement à la fin de la fonction. Les variables locales et les paramètres sont stockés dans le registre FP. Lors de l'appel de la fonction,l'adresse de retour est empilée (ret appelé parfois "return eip"), puis l'ancien registre FP que l'on va appeller SFP (save FP) est à son tour empilé. Puis le "stack pointer" est incrémenté afin de laisser de la place pour les variables locales de la fonction dans les différents buffers.

base de la mémoire sommet de la mémoire
Buffers sfp ret
<------ [ ] [ ] [ ] (...variables de la fonction père)
Sommet de pile bas de pile

Principe du buffer overflow :
Si dans la fonction, on écrit des données dans le buffer (strcpy() par exemple écrit des données sans en vérifier la taille, jusqu'à ce qu'elle trouve un \0), si on dépasse la place réservé pour ces buffers, on va alors écrire plus loin et écraser SFP puis si on va encore plus loin l'adresse de retour etc.
A la fin de la fonction, les buffers sont détruits, SFP remis dans le registre FP et le processeur lit la prochaine instruction dont l'adresse est contenue dans RET. Mais si cet emplacement a été écrasé, il est fort probable que les données écrites représentent un emplacement mémoire qui est hors de l'espace mémoire alloué au process, provoquant une erreur de segmentation ("segmentation fault"). C'est ce qui est appellé "buffer overflow".

Stack smashing :
La partie la plus intéressante est que l'on peut donc modifier l'adresse de retour de la fonction et donc faire exécuter à peu près ce que l'on veut au processeur, pourvu que les données à exécuter soit présentent dans la mémoire utilisée par la fonction.
Il y a donc trois problèmes, écrire les données que l'on veut que le processeur exécute, arriver à savoir



l'emplacement exact de l'emplacement ou est stocké l'adresse de retour, afin de créer un offset parfait (trop peu et on ne crée pas d'overflow, et trop, on a un risque de SIGSEGV) et savoir quoi mettre à la place de l'ancienne adresse de retour pour lancer notre code, c'est à dire savoir à quel endroit débute le buffer.
La manière la plus simple de résoudre le premier problème est d'écrire nos données directement dans le buffer où l'on va provoquer l'overflow à l'emplacement réservé (ou dans le data segment),. Pour cela, il faut que le buffer soit assez grand, ce qui n'est pas toujours le cas et d'écraser l'adresse de retour pour qu'elle pointe en arrière dans le buffer.
Le deuxième problème est un peu plus dur à résoudre. En effet, si on ne connait pas le code d'un programme, il est difficile de savoir combien de mémoire est alloué à la fonction.
Les solutions sont les essais successifs ou le désassemblage. Si le buffer est plus grand que notre code, on peut mettre des NOP qui est une instruction servant uniquement à temporiser.
Le dernier problème est un peu plus subtile. Lors de l'exécution d'un process, l'adresse du début de la mémoire alloué au programme est dépendante de l'environnement de la machine. Elle ne sera donc pas la meme. Néanmoins, en général, toutes les piles débutent à la meme adresse (OxFF). On peut considérer que peu de mémoire est occupé et deviner à quel endroit commencera le buffer. Pour améliorer les résultats, il suffit de rajouter un maximum de NOP avant notre code, comme ça si on pointe trop en avant du code, le processeur va faire tous les nop et arriver à notre code.

Shellcode :
Le point suivant est la création du code que nous voulons exécuter. En général, un simple shell suffit. C'est ce qu'on appelle un shellcode.
Ce shellcode doit etre petit, écrit en assembleur et indépendant de l'environnement. Sous unix on va lancer /bin/sh. Si le programme sur lequel on fait le buffer overflow est lancé avec le setuid ou setgid ou lancé directement avec les droits root, le shell sera root.
Pour créer ce shell, on écrit un code C, on le compile et on le désassemble.
Il y a généralement 2 gros problèmes. En effet, en général, les fonctions qui écrivent dans la mémoire nous imposent certaines contraintes. Pour strcpy() par exemple, il ne faut pas avoir de caractère nul dans notre code ASM. Il serait en effet considéré comme caractère d'échappement. Parfois, c'est plus subtil, et on a le droit uniquement à certaines commandes (à cause d'un traitement des données sur des chaines de texte par exemple). Il faut donc mettre la chaine /bin/sh suivit d'un caractère nul à la fin de notre shellcode.
Le deuxième problème est que généralement, on ne connait pas l'espace mémoire réservé au programme ce qui nous empêche de savoir exactement où sauter. On peut par contre utiliser des CALL et des JMP qui nous permettent de sauter à un endroit dans la mémoire par rapport à l'endoit actuel. On peut avec ça mettre notre chaine /bin/sh en mémoire à l'aide d'un CALL, (il va sauvegarder l'emplacement de la prochaine instruction et donc de la chaine) et jumper vers notre code.
En général un programme C finit par un exit(0). Il est donc souvent interessant de mettre exit(0) à la fin de notre shellcode, afin de fermer l'application comme prévu et d'éviter que le processeur n'exécute n'importe quoi en cas de problème.

L'exploit :
La dernière partie de l'article explique comment automatiser la création de l'exploit, comment trouver rapidement la taille de l'offset et la taille du buffer.
La principale optimisation est l'utilisation de NOP, comme vu au dessus, ce qui nous permet de pointer à peu près n'importe où dans le buffer.

La fin concerne les différents problèmes, si par exemple le buffer est trop petit pour contenir notre shellcode. La technique consiste à mettre notre code dans une variable d'environnement et ensuite faire déborder le buffer avec l'adresse de cette variable en mémoire. Les variables d'environnement sont stockées en sommet de pile quand le programme est lancé, les modifications par setenv() sont allouées ailleurs.

Tout à la fin, l'auteur explique comment trouver des buffer overflow (les fonctions strcat(), strcpy(), sprintf(), et vsprintf() ne vérifient pas la taille des paramètres) et donne quelques shellcodes pour différents OS. Ceci répondant à la question 5.

3. Actions à entreprendre afin de s'assurer du niveau de sécurité du parc

En tant qu'administrateur d'un parc informatique, plusieurs actions peuvent être envisagés.

Si l'administrateur a le temps et les compétences suffisantes, il peut vérifier les programmes tournant sur le parc. Soit en analysant le code pour les applications codés par l'entreprise ou grace à divers outils vérifiant les principales erreurs.

Il peut etre envisager de protéger la pile des machines, le principe est de stocker une clé, générée à l'exécution entre la fin de la pile et la valeur de retour. Si cette clé est modifiée, l'exécution est avortée (disponible en option dans les compilateurs C récents - éventuellement avec recours à des patchs). La principale critique étant que l'appel de fonction est ralenti. Ou de la rendre non exécutable.
Mais ces stratagèmes peuvent facilement être évités en utilisant l'environnement de la machine, avec par exemple sous windows, les DLL.
La meilleur manière pour éviter les buffer overflow a été mise en oeuvre il y a plus d'un an sur les noyaux Unix et très récemment sous Windows avec Vista. Le principe est simple, au lieu de démarrer à une adresse connue, l'adresse du début de la pile est aléatoire et rend donc très difficile l'exécution d'un buffer overflow.

Il est donc indispensable de maintenir les systèmes à jour avec les derniers noyaux sous Unix et de forcer les mises à jour automatiques sous Windows.

Le réseau doit lui aussi être conçu de manière a limiter les risques pour le reste du réseau au cas où une machine serait compromise.

Dans un premier temps, il est nécessaire de diviser les menaces en deux catégories. Les menaces extérieures et les menaces provenant de l'intérieur.

On est en droit de penser que les personnes ayant accès aux machines sont des personnes de confiance et que les accès sont limités et logués. Afin de s'assurer ce cela, il est nécessaire d'interdire l'accès au domaine à toute station n'ayant rien à faire dans le parc informatique, cela limitera les essais des bidouilleurs. Dans un second temps, il faut proscrire l'utilisation d'un compte administrateur sur les machines lors d'un usage quotidien. L'utilisation d'un antivirus suffira en général à contenir toute tentative de « privilege escalation » et la propagation des vers ou autres scripts.

Afin de contenir les attaques sur les serveurs et surtout les attaques provenant de l'extérieur, il est nécessaire d'utiliser un firewall, de préférence physique afin qu'il soit plus difficilement compromis.

Il est indispensable de mettre en place une DMZ et d'y placer les serveurs fournissant des services web avec un firewall entre la DMZ et le réseau local et un entre la DMZ et Internet. Il faut configurer les

firewall afin de ne laisser passer uniquement les requêtes aux services dédiés aux différents serveurs. Il est primordial de fermer les ports dont l'accès n'est pas nécessaire (exemple sur Windows OS les ports 135 et 445) et de ne pas répondre aux pings pour les stations ayant un accès direct à Internet.

Les accès distants doivent être logués et en cas de trafic anormal, analysés.

Il est impossible de sécuriser à 100% un réseau, il faut surtout pouvoir limiter les attaques, et en cas de machine compromise, de ne pas compromettre tout le réseau. Pour cela, il est important de rendre inaccessible depuis l'extérieur les serveurs gérant le domaine. Si le serveur de domaine tombe, l'attaquant peut devenir « domain administrator » et faire ce qu'il veut.

Il est aussi possible de faire appel à des sociétés spécialisées dans les audits de sécurités. Leur méthode est assez simple. Dans un premier temps, il analysent les risquent en essayant de rentrer sur le réseau par l'intermédiaire des services ouverts sur l'extérieur. Ils vérifient ensuite les patches de sécurités disponibles ainsi que les versions des logiciels utilisés et les failles connus sur ceux-ci. Ils analysent les configurations des routeurs et des firewall, du DHCP, des actives directory etc.
Ensuite ils passent du coté des utilisateurs en vérifiant si les stations ont des antivirus, des antispy etc. et si l'entreprise le souhaite, ils forment les utilisateurs aux bases de la sécurité.
Si vraiment l'entreprise a beaucoup d'argent, ils peuvent analyser les logiciels de manière minutieuse afin d'y trouver des failles et de les faire corriger.

4. Grandes familles de vulnérabilités liées au codage

Stack smashing & Buffer overflow qui permettent:

- obtention de privilèges supplémentaires
- instabilité de l'application ou du serveur
- injection de code malveillant

Input validation error tel que :

- Command injection et SQL injection :
L'attaque par SQL injection, ou plus généralement par command injection consiste à injecter du code malveillant, en l'occurrence une requête SQL dans un paramètre. Si ce paramètre est insuffisamment contrôlé par l'application et est utilisé au sein d'une commande système, où d'une requête de base de données, alors cette dernière va voir son comportement modifié : des actions non prévues vont être pilotées à distance par un pirate.

Cross site :
Le cross site scripting requière que la valeur d'au moins un paramètre de l'URL ou du formulaire soit intégré au contenu de la page web sans contrôle suffisant. Ainsi il suffit de remplacer cette valeur par du code exécutable pour exécuter une action sur le serveur (Shell, ...) ou chez le client (JavaScript, ActiveX, Flash ...).

- Directory traversal :
Exploit permettant de passer à travers des répertoires où on n'a pas les droits afin d'accéder à des fichiers qui est normalement inaccessible.

- Bypass Issue (passer à travers l'authentification, cela s'appuie en général sur des méthodes comme SQL injection)

Memory corruption :
Faille qui arrive lorsqu'un programme modifie malencontreusement une partie de la mémoire. Quand la mémoire est ensuite utilisée, le programme crash ou sature le système.
Ex : Denial of service (peut être considéré comme une attaque de type brute force, mais en général, les données sont formée de manière à rendre le service indisponible, en provoquant une utilisation importante de la mémoire par exemple).

Format string bugs :
Vulnérabilité découverte en 1999, elle n'a pas encore été exploitée, un seul exploit a été rendu public. L'attaque est rendue possible par l'omission de la spécification exacte du format à utiliser pour afficher une chaîne, en C. Classiquement, le programmeur écrit printf(str); à la place de printf("%s", str);. Du coup, si printf est un paramètre entré par l'utilisateur, celui-ci se retrouve avec beaucoup de latitude pour jeter un oeil sur la pile.

Improperly handling shell metacharacters :
Des méta caractères qui sont mal traités et qui sont alors interprétés.

Remote file inclusion :
Faille très fréquente sous PHP par exemple qui consiste à inclure son propre code directement sur le serveur en modifiant les variables des fonction tel que include().

Time-of-check-to-time-of-use (TOCTOU) :
Faille qui vient du fait que les droits sont vérifies lors de l'accès à une ressource et plus ensuite. Si les droits ont changés entre temps, l'utilisateur n'est pas affecté.

Symlink races :
Peut survenir lorsqu'une application crée un fichier de manière incorrecte. Un utilisateur mal intentionné pourrait alors créer un lien vers un fichier dont il n'a pas accès et en modifier à l'aide du programme le contenu.

Privilege escalation : (plutôt type d'attaque que type de faille)
Une mauvaise gestion des droits utilisateurs permettant une ascension locale des privilèges en permettant l'exécution de programmes avec les droits administrateur par exemple.
ex : Cross Zone scripting dans les browser.

Session hijacking (vol de session, problème de codage quand la prédiction est possible) :
Le Session hijacking par prédiction est l'art de deviner un identifiant de session valide. Cette attaque concerne les serveurs dont l'algorithme de génération de cet identifiant permet sa prédictibilité. Ainsi, avec un peut d'analyse, en fonction de certaines informations connues (un login courant ou obtenu de façon légitime, la date, et autres informations que l'on peut obtenir facilement...), le pirate peut forger un identifiant rattaché à une session en cours. Cette attaque est d'autant plus facile à réaliser que l'application possède un grand nombre d'utilisateurs.

5. Ce problème est-il lié à un système d'exploitation en particulier ?

Non, tous les OS actuels sont touchés, de Unix à Windows en passant par Mac OS X.

Par contre, le noyau actuel de Unix rendent l'adresse du début de la pile aléatoire et donc du début du buffer, ce qui rend le travail beaucoup plus dure voir même impossible. Il en est actuellement de même pour le dernier OS de microsoft : Vista. (bien que Windows 2003 Server devait lui aussi être immunisé contre les overflow ...).

Le plus connu de tous les shellcode est celui permettant de lancer un Shell sous Unix :

```
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 »;
```

Je vous invite à aller sur http://shellcode.org/ pour avoir les plus courant sur une multitude l'OS ou encore sur des sites de sécurité comme www.k-otik.com (maintenant FrSIRT) qui rendait certains exploits publics.

6. Explications simples sur la faille du code source présent dans le sujet

```
int doit()
{
	printf("Perdu\n");
}
int main(int argc, char *argv[])
{
	int (*p1)();
	char buffer[128];
	p1=&doit;
	if ( argc > 1 ) strcpy(buffer,argv[1]);
	p1();
	exit(1);
}
```

La commande char buffer[128] réserve 128 emplacement de type char (1 octet) soit 128 octets.

strcpy(buffer,argv[1]); copie en mémoire du première argument passé en paramètres sans en vérifier la taille (jusqu'à trouver un \0). Si cet argument est plus grand que 128 octets, il va dépasser et être écrit en mémoire à un endroit non réservé. Il va donc écraser des informations et peu être aussi l'adresse de la prochaine instruction, p1(). Au pire, le programme va planter et peu etre faire planter le système, et au pire, quelqu'un peut intentionnellement entrer un shellcode en paramètre et en écrasant l'adresse de retour de la fonction strcpy(), faire exécuter son propre code et prendre la main sur la machine. (Voir résumé de l'article).

Pour éviter cela, il y a plusieurs moyens, soit vérifier la taille du paramètre, soit utiliser des fonctions sécurisés (strncpy() à la place de strcpy()), soit utiliser un langage plus sure que C ou C++.

7. *Apport ou restriction de la fonction exit()*

La fonction exit() permet d'éviter au processeur d'exécuter n'importe quoi en cas de problème avec la fonction précédente.

Mettre exit(1) alors qu'en général il y a exit(0) peut permettre de déceler un problème, si quelqu'un clos l'application et en pensant que l'application se finit par exit(0) met un 0 dans le registre EBX, cela peut mettre la puce à l'oreille de l'admin.

Annexe

Quelques sites intéressants :

http://en.wikipedia.org/wiki/Vulnerability_%28computing%29#Identifying_and_removing_vulnerabilities
http://www.lrde.epita.fr/~didier/comp/lectures/os_11_overflow.pdf

Sites spécialisés sur la sécurité, sur les exploits 0 day et les alertes :

www.k-otik.com (maintenant www.frsirt.com)
www.governmentsecurity.org
www.eeye.com