Overflow

Table des matières

Une pile (en anglais stack) est une structure de données fondée sur le principe « dernier arrivé, premier sorti » (en anglais LIFO pour last in, first out). La plupart des microprocesseurs gèrent nativement une pile. Elle correspond alors à une zone de la mémoire, et le processeur retient l'adresse du dernier élément.

1. Dépassement d'entier

Un dépassement d'entier (integer overflow) est une condition qui se produit lorsqu'une opération mathématique produit une valeur numérique supérieure à celle représentable dans l'espace de stockage disponible. Par exemple, l'ajout d'une unité au plus grand nombre pouvant être représenté entraîne un dépassement d'entier.

Le nombre de bits d'un espace de stockage détermine la valeur maximale qui peut y être représentée. Les nombres de bits des espaces de stockage les plus courants et les valeurs maximales associées sont :

- 8 bits : valeur maximum représentable = $2^8 - 1 = 255$;
- 16 bits : valeur maximum représentable = $2^{16} - 1 = 65\,535$;
- 32 bits : valeur maximum représentable = $2^{32} - 1 = 4\,294\,967\,295$;
- 64 bits : valeur maximum représentable = $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$;

Exemple :

L'exemple suivant est tiré d'une vulnérabilité réelle touchant OpenSSH (versions 2.9.9 à 3.3).

```
nresp = packet_get_int();    // extrait un entier d'un paquet reçu par OpenSSH.

if ( nresp > 0 )
{
   // alloue un tampon de: nresp * 4 octets
  response = xmalloc(nresp * sizeof(char*));
  for (int i = 0; i < nresp; i++)
       response[i] = packet_get_string(NULL);
}
```

Dans le code présenté ci-dessus, un entier est extrait d'un paquet reçu par OpenSSH. Cet entier est ensuite multiplié par la taille (sizeof) d'un pointeur de char, c'est-à-dire habituellement 4 octets sur un système en 32 bit. Le résultat de cette multiplication est ensuite passé en paramètre à la fonction xmalloc (semblable à malloc) qui sert à allouer un tampon.

Si l'entier reçu dans le paquet (variable nresp) a une valeur de 1 073 741 824, le résultat de l'opération « nresp*sizeof(char*) » équivaut à :

$1\,073\,741\,824 \times 4 = 4\,294\,967\,296$.

La valeur maximale représentable d'un entier sur 32 bits étant de $2^{32} - 1$, la troncature s'opère donc à 2^{32}. Le paramètre passé à la fonction xmalloc dans le code vaut donc (où % est l'opérateur modulo) : $(1073741824 * 4) \% 2^{32} = 0$

soit en écriture mathématique standard : $(1\,073\,741\,824 \times 4) \equiv 0\ [2^{32}]$

Comme dans la plupart des cas les allocations mémoire autorisent une allocation de 0 octet, la fonction xmalloc renvoie un pointeur valide sur un tampon de 0 octet. La boucle suivant directement la fonction d'allocation va ajouter des données dans ce tampon de 0 octet provoquant alors un débordement de tas.

2. Débordement de tas

Un dépassement ou débordement de tas (heap overflow) est un bug de la classe des dépassements de tampon (buffer overflow), dans lequel le débordement concerne un tampon alloué dans le tas (la mémoire allouée dynamiquement lors de l'exécution d'un programme).

Cette technique est un moyen pour une personne malveillante d'utiliser une erreur d'écriture d'un programme, en particulier en injectant des données des variables différentes de celles d'origine, et ainsi de prendre possession du système cible à hauteur de celui qui a lancé ce programme (c'est-à-dire avec les mêmes privilèges et droits que celui-ci).

Le Heap Overflow requiert des connaissances précises du système d'exploitation ciblé pour pouvoir être exploité, et notamment l'implémentation du sous-système d'allocation dynamique de la mémoire.

3. Dépassement de buffer

Un dépassement de tampon ou débordement de tampon (buffer overflow) est un bug par lequel un processus, lors de l'écriture dans un tampon, écrit à l'extérieur de l'espace alloué au tampon, écrasant ainsi des informations nécessaires au processus.

Lorsque le bug se produit, le comportement de l'ordinateur devient imprévisible. Il en résulte souvent un blocage du programme, voire de tout le système.

Le bug peut aussi être provoqué intentionnellement et être exploité pour violer la politique de sécurité d'un système. Cette technique est couramment utilisée par les pirates. La stratégie de l'attaquant est alors de détourner le programme bugué en lui faisant exécuter des instructions qu'il a introduites dans le processus.

Une vulnérabilité de ce type permet d'écrire des données au delà du tampon qui leur est alloué. Avec le langage C, cela peut typiquement se produire lorsque des fonctions telles que gets (lecture d'une suite de caractères sur le flux d'entrée standard) ou strcpy (copie d'une chaîne de caractères), qui ne contrôlent pas le nombre de caractères écrits en mémoire, sont appelées sur des entrées non fiables. Il devient alors possible pour l'attaquant de modifier les variables situées à la suite du tampon.

3.1. Attaques génériques

Sur la plupart des architectures de processeurs, l'adresse de retour d'une fonction est stockée dans la pile d'exécution. Lorsqu'un dépassement se produit sur un tampon situé dans la pile d'exécution, il est alors possible d'écraser l'adresse de retour de la fonction en cours d'exécution. L'attaquant peut ainsi contrôler le pointeur d'instruction après le retour de la fonction, et lui faire exécuter des instructions arbitraires, par exemple un code malveillant qu'il aura introduit dans le programme.

3.2. Attaques spécifiques

Des vulnérabilités de dépassement de tampons peuvent exister dans des programmes dans lesquels l'adresse de retour est protégée (par exemple par le Stack-Smashing Protector), ou exister ailleurs que la pile d'exécution. Dans certains cas, elles peuvent tout de même être exploitées pour mettre à mal la sécurité du programme, par exemple en écrasant d'autres pointeurs que le pointeur d'instruction, ou en modifiant des données spécifiques du programme pour en altérer la logique.

3.3. Contre mesures

Afin d'éviter ces dépassements, certaines fonctions ont été réécrites pour prendre en paramètre la taille du tampon dans lequel les données sont copiées, et éviter ainsi de copier des informations à l'extérieur du tampon. Ainsi strlcpy est une version de strcpy qui tient compte de la taille du tampon. De même, fgets est une version de gets prenant en compte la taille du tampon1. Les fonctions strlcpy et strlcat ont été initialement disponibles sous OpenBSD et elles tendent à se répandre dans divers logiciels comme rsync et KDE.

3.4. Détails techniques sur architecture x86 (Intel)

Un programme en exécution (un processus) découpe la mémoire adressable en zones distinctes :

1. la zone de code où sont stockées les instructions du programme en cours ;
2. la zone des données où sont stockées certaines des données que manipule le programme ;
3. la zone de la pile d'exécution ;
4. la zone du tas.

Contrairement aux deux premières, les deux dernières zones sont dynamiques, c'est-à-dire que leur pourcentage d'utilisation et leur contenu varient tout au long de l'exécution d'un processus.

La zone de la pile d'exécution est utilisée par les fonctions (stockage des variables locales et passage des paramètres). Elle se comporte comme une pile, c'est-à-dire dernier entré, premier sorti. Les variables et les paramètres d'une fonction sont empilés avant le début de la fonction et dépilés à la fin de la fonction.

Une fonction est une suite d'instructions. Les instructions d'une fonction peuvent être exécutées (en informatique, on dit que la fonction est appelée) à partir de n'importe quel endroit d'un programme. À la fin de l'exécution des instructions de la fonction, l'exécution doit se continuer à l'instruction du programme qui suit l'instruction qui a appelé la fonction.

Pour permettre le retour au programme qui a appelé la fonction, l'instruction d'appel de la fonction (l'instruction call) enregistre l'adresse de retour dans la pile d'exécution. Lors de l'exécution de l'instruction ret qui marque la fin de la fonction, le processeur récupère l'adresse de retour qu'il a précédemment stockée dans la pile d'exécution et le processus peut continuer son exécution à cette adresse.

Plus précisément, le traitement d'une fonction inclut les étapes suivantes :

1. l'empilement des paramètres de la fonction sur la pile d'exécution (avec l'instruction push) ;
2. l'appel de la fonction (avec l'instruction call) ; cette étape déclenche la sauvegarde de l'adresse de retour de la fonction sur la pile d'exécution ;
3. le début de la fonction qui inclut :
   - la sauvegarde de l'adresse de la pile qui marque le début de l'enregistrement de l'état actuel du programme,
   - l'allocation des variables locales dans la pile d'exécution ;
4. l'exécution de la fonction ;
5. la sortie de la fonction qui inclut la restauration du pointeur qui marquait le début de l'enregistrement de l'état du programme au moment de l'appel de la fonction,

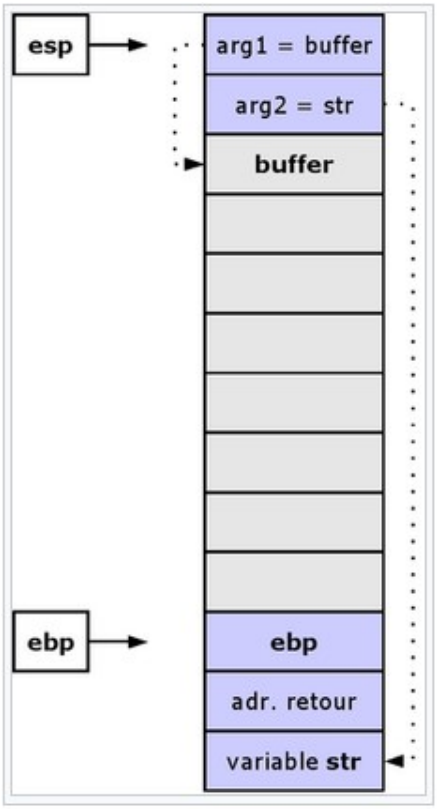
6. l'exécution de l'instruction ret qui indique la fin de la fonction et déclenche la récupération de l'adresse de retour et le branchement à cette adresse.

Soit l'extrait de programme C suivant (volontairement simplifié) :

```
#include <stdio.h>
#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;
}
```

Ce qui est traduit ainsi par un compilateur C (ici le compilateur GCC[1] avec architecture x86) :

```
push ebp                ; entrée de la fonction
mov ebp,esp
sub esp,40              ; 40 octets sont « alloués » (32 + les 2
                        ; variables pour l'appel de strcpy)

mov eax,[ebp+0x8]       ; paramètre de la fonction (str)
mov [esp+0x4],eax       ; préparation de l'appel de fonction
                        ; deuxième paramètre
lea eax,[ebp-0x20]      ; ebp-0x20 contient la variable
                        ; locale 'buffer'
mov [esp],eax           ; premier paramètre
call strcpy
                        ; sortie de la fonction
leave                   ; équivalent à mov esp,ebp et pop ebp
ret
```



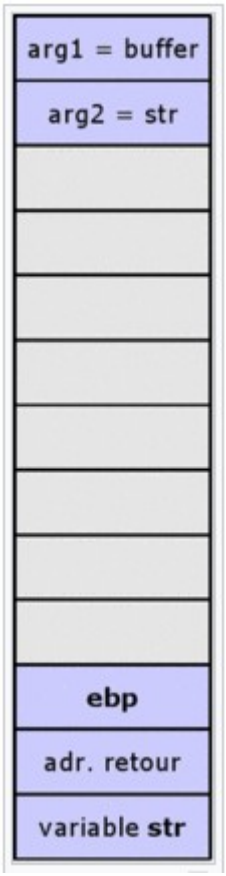
Ci-contre, l'état de la pile d'exécution et des deux registres (ebp et esp) juste avant l'appel de la fonction strcpy.

La fonction strcpy copiera le contenu de str dans buffer. La

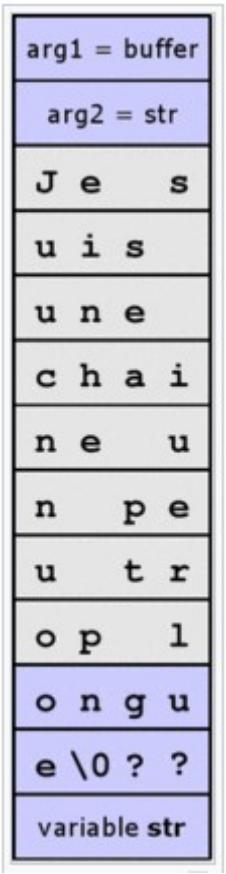
1 GNU C Compiler

fonction strcpy ne fait aucune vérification : elle copiera le contenu de str jusqu'à ce qu'elle rencontre un caractère de fin de chaîne (caractère nul).

Si str contient 32 octets ou plus avant le caractère nul, la fonction strcpy continuera à copier le contenu de la chaîne au-delà de la zone allouée par la variable locale buffer. C'est ainsi que les informations stockées dans la pile d'exécution (incluant l'adresse de retour de la fonction) pourront être écrasées comme indiqué dans l'illustration suivante :



Dépassement de tampon : la fonction strcpy copie la chaîne vers la zone mémoire indiquée, dépasse la zone allouée et écrase l'adresse de retour de la fonction foo.



Dans l'exemple précédent, si str contient un code malveillant sur 32 octets suivi de l'adresse de buffer, au retour de la fonction, le processeur exécutera le code contenu dans str.

Pour que cette « technique » fonctionne, il faut deux conditions :

1. le code malveillant ne doit contenir aucun caractère nul, sans quoi strcpy() arrêtera sans copier les octets suivants le caractère nul ;
2. l'adresse de retour ne doit pas non plus contenir de caractère nul.

Le dépassement de tampon avec écrasement de l'adresse de retour est un bug du programme. L'adresse de retour ayant été écrasée, à la fin de la fonction, le processeur ne peut brancher vers l'adresse de retour originale, car cette adresse a été modifiée par le dépassement de tampon. Dans le cas d'un bug involontaire, l'adresse de retour a généralement été remplacée par une adresse en dehors de la plage adressable et le programme plante en affichant un message d'erreur (erreur de segmentation).

Un hacker peut utiliser ce comportement à ses fins. Par exemple, en connaissant la taille du tampon (dans l'exemple précédent 32 octets), il peut écraser l'adresse de retour pour la remplacer par une adresse qui pointe vers un code à lui, de manière à prendre le contrôle du programme. De cette

façon, il obtient les droits d'exécution associés au programme qu'il a détourné et peut dans certains cas accéder à des ressources critiques.

La forme d'attaque la plus simple consiste à inclure dans une chaîne de caractères copiée dans le tampon un programme malveillant et d'écraser l'adresse de retour par une adresse pointant vers ce code malveillant.

Pour arriver à ses fins, l'attaquant doit surmonter deux difficultés :

1. trouver l'adresse de début de son code malveillant dans le programme attaqué ;
2. construire son code d'exploitation en respectant les contraintes imposées par le type de variable dans lequel il place son code (dans l'exemple précédent, la variable est une chaîne de caractères).

3.5. Trouver l'adresse de début du code malveillant

La technique précédente nécessite généralement pour l'attaquant de connaître l'adresse de début du code malveillant. Ceci est assez complexe, car cela demande généralement des essais successifs, ce qui n'est pas une méthode très « discrète ». De plus, il y a de fortes chances que l'adresse de la pile change d'une version à l'autre du programme visé et d'un système d'exploitation à l'autre.

L'attaquant veut généralement exploiter une faille sur le plus de versions possible du même programme (afin peut-être de concevoir un virus ou ver). Pour s'affranchir du besoin de connaître l'adresse du début du code malveillant, il doit trouver une méthode qui lui permette de brancher sur son code sans se préoccuper de la version du système d'exploitation et du programme visé tout en évitant de faire de multiples tentatives qui prendraient du temps et dévoileraient peut-être sa présence.

Il est possible dans certains cas de se servir du contexte d'exécution du système cible. Par exemple, sur des systèmes Windows, la plupart des programmes, même les plus simples contiennent un ensemble de primitives système accessibles au programme (DLL[2]). Il est possible de trouver des bibliothèques dont l'adresse mémoire lors de l'exécution change peu en fonction de la version du système. Le but pour l'attaquant est alors de trouver dans ces plages mémoires des instructions qui manipulent la pile d'exécution et lui permettront d'exécuter son code.

Par exemple, en supposant que la fonction strcpy manipule un registre processeur (eax) pour y stocker l'adresse source. Dans l'exemple précédent, eax contiendra une adresse proche de l'adresse de buffer au retour de strcpy. Le but de l'attaquant est donc de trouver dans la zone mémoire supposée « fixe » (la zone des DLL par exemple) un code qui permet de sauter vers le contenu de eax (call eax ou jmp eax). Il construira alors son buffer en plaçant son code suivi de l'adresse d'une instruction de saut (call eax ou jmp eax). Au retour de strcpy, le processeur branchera vers une zone mémoire contenant call eax ou jmp eax et puisque eax contient l'adresse du buffer, il branchera de nouveau vers le code et l'exécutera.

3.6. Préventions

Pour se prémunir contre de telles attaques, plusieurs options sont offertes au programmeur. Quelques-unes de ces options sont décrites dans les deux sections suivantes.

3.6.1. Protections logicielles

1. Bannir de son utilisation les fonctions dites « non protégées ». Préférer par exemple strncpy

2 Dynamic Link Library

à strcpy ou alors fgets à scanf qui effectue un contrôle de taille. Les compilateurs récents peuvent prévenir le programmeur s'il utilise des fonctions à risque, même si leur utilisation demeure possible.

2. Utiliser des options de compilation permettant d'éviter les dépassements de mémoire tampon menant à la corruption de pile, comme le SSP[3] (technique du canari).
3. Se reposer sur les protections offertes par le système d'exploitation, comme l'Address space layout randomization ou la Data Execution Prevention.
4. Utiliser des outils externes pendant le développement pour détecter les accès mémoire invalides à l'exécution, par exemple la bibliothèque Electric Fence ou Valgrind.
5. Utilisation de langages à contexte d'exécution managé implémentant la vérification de bornes des tableaux (e.g. Java).

En 2017, SSP ou des protections similaires sont activées par défaut dans la plupart des compilateurs. Cependant, les protections contre les dépassements de tampons restent souvent absentes des systèmes embarqués et plus contraints en ressources.

3.6.2. Technique du canari

La technique du canari fait référence aux canaris qui étaient utilisés jadis pour détecter les fuites de grisou dans les mines de charbon.

Le principe est de stocker une valeur secrète, générée à l'exécution, juste avant l'adresse de retour (entre la fin de la pile et l'adresse de retour). Lorsque la fonction se termine, cette valeur est contrôlée. Si cette clé a changé, l'exécution est avortée. Cette méthode permet d'empêcher l'exécution de code corrompu, mais augmente la taille du code et ralentit donc l'appel de la fonction.

Cette protection est disponible en option dans les compilateurs C actuels(GCC version 4.93), éventuellement avec recours à des patchs.

3.6.3. Protections matérielles

Les microprocesseurs récents, 64 bits notamment, implémentent des protections efficaces (technologies NX Bit et XD bit).

4. Dépassement de pile

Un dépassement de pile ou débordement de pile (stack overflow) est un bug causé par un processus qui, lors de l'écriture dans une pile, écrit à l'extérieur de l'espace alloué à la pile, écrasant ainsi des informations nécessaires au processus.

L'expression dépassement de pile peut s'appliquer à toutes les piles. Cependant, lorsque l'on parle de dépassement de pile, on fait habituellement référence à la pile d'exécution. Il serait alors plus précis de dire dépassement de la pile d'exécution.

Dans tous les langages de programmation, la pile d'exécution contient une quantité limitée de mémoire, habituellement déterminée au début du programme. La taille de la pile d'exécution dépend de nombreux facteurs, incluant le langage de programmation, l'architecture du processeur, l'utilisation du traitement multithread et de la quantité de mémoire vive disponible. Lorsque trop d'informations sont enregistrées dans la pile d'exécution, la pile déborde et écrase des zones de

3 Stack-Smashing Protector

programme à l'extérieur de la pile. On dit alors qu'il y a dépassement de pile ou dépassement de la pile d'exécution. Il en résulte généralement une interruption du programme.

Un dépassement de pile d'exécution est généralement causé par l'une des deux erreurs de programmation suivantes :

1. une récursivité infinie ;
2. une allocation de variables trop grandes dans la pile.

4.1. Récursivité infinie

La cause la plus fréquente des dépassements de pile est une récursivité trop profonde ou infinie.

Il est à noter qu'une récursivité profonde ou même infinie ne cause pas toujours un dépassement de pile. En effet, certains langages, comme Scheme, permettent une sorte de récursivité infinie, la récursion terminale (tail recursion) sans dépassement de pile. Pour ce faire, ces langages transforment la récursivité en une itération, éliminant ainsi l'utilisation de la pile d'exécution.

Exemples de récursivité infinie :

```
void a()
{
  a();
}

int main()
{
  a();
  return 0;
}
```

4.2. Allocation de variables trop grandes dans la pile

L'autre grande cause de dépassement de pile résulte d'une tentative d'allouer plus d'espace dans la pile que ce que la pile peut contenir. Cela est généralement le résultat de la déclaration de variables locales demandant trop de mémoire. Pour cette raison, les tableaux de plus de quelques ko devraient toujours être alloués dynamiquement plutôt que comme une variable locale.

Exemple d'allocation de variables trop grandes dans la pile :

```
int main()
{
  double n[10000000];
  return 0 ;
}
```

Le tableau déclaré consomme plus de mémoire que ce qui est disponible dans la pile.