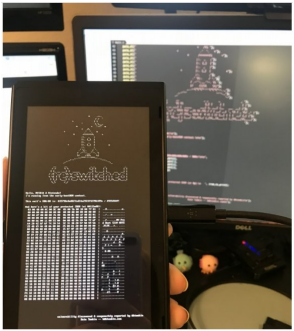
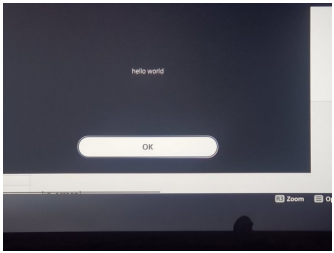
Introduction aux Buffer Overflows

Thomas BYGODT

Novembre 2022

Pourquoi ?





Welcome to http://www.worm.com !

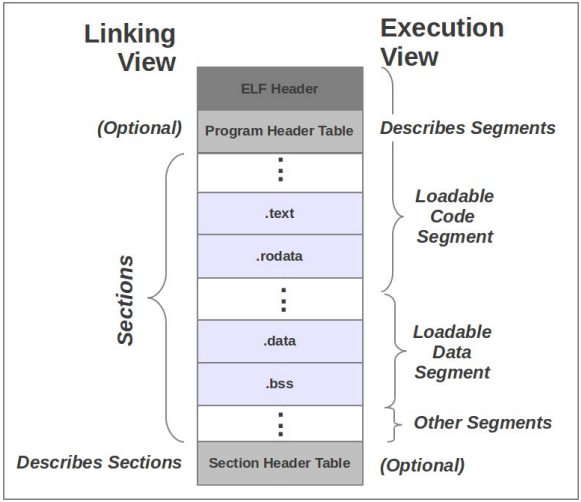
Hacked By Chinese!

https://misc.ktemkin.com/fusee_gelee_nvidia.pdf
https://googleprojectzero.github.io/0days-in-the-wild/0day-RCAs/2021/CVE-2021-30858.html
https://en.wikipedia.org/wiki/Code_Red_(computer_worm)
https://research.checkpoint.com/2017/eternalblue-everything-know/

Programme

Sommaire

Les sections



Source : Mugabi Siro

Les sections

```
$ readelf -l <binary>
[...]
Correspondance section/segment :
Sections de segment...
    00
    01     .interp
    02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
           .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got
           .text .fini .rodata .eh_frame_hdr .eh_frame
    03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
    04     .dynamic
    05     .note.ABI-tag .note.gnu.build-id
    06     .eh_frame_hdr
    07
    08     .init_array .fini_array .jcr .dynamic .got
```

Essayer `readelf -S <binary>`

Les sections

Voici les plus importantes :

- **TEXT** : le code
- **PLT** : résoud les fonctions de la libc exécutées
- **GOT** : les adresses de la libc résolues grâce à la plt
- **BSS** : les variables statiques définies à la compilation (`int v;`)
- **DATA** : les données variables définies à la compilation (`int v = 10;`)

Gestion de la mémoire en C

Statique

La mémoire est fixée au moment de la compilation. Les données sont stockées sur la pile (stack).

- Déclaration de variables : `int *somme;`
- Tableaux : `int somme[2];`

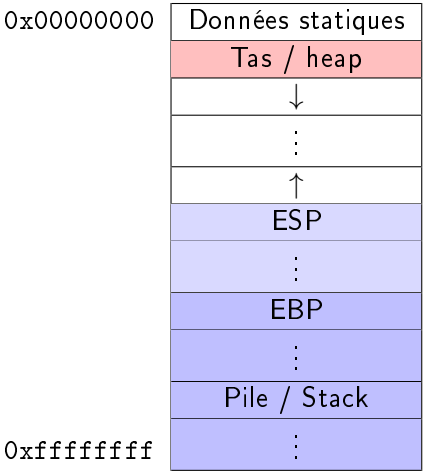
Dynamique

La taille de mémoire nécéssaire est inconnue au moment de la compilation, seuls les types sont connus. Les données sont stockées sur le tas (heap).

- Initialiser une variable : `somme = (int *) malloc(argc * sizeof(int));`
- Libérer une variable inutilisée : `free(somme)`

La pile / The stack

- Pile de type **L**ast **I**n **F**irst **O**ut
- Contient les variables locales
- Séparée en frame contenant le contexte de chaque fonction
- **ESP** : Pointe vers le début de la stack
- **EBP** : Pointe vers le contexte précédent



Protection des binaires

Pour protéger au maximum les binaires, des protections peuvent être ajoutées soit lors de la compilation soit lors de l'exécution :

- **Canary / Stack Smashing Protection (SSP)** : valeur aléatoire défini lors de l'éxecution du programme qui est vérifié en sortie de fonction
- **NX** : Rend la pile non executable
- **Address Space Layout Randomization (ASLR)** : Randomizer les adresses du tas, de la pile et des bibliothèques lors de l'exécution (activé par défaut la plupart du temps)
- **Position Independent Executable (PIE)** : Randomizer les adresses de base des segments d'un binaire (mitige le ROP)
- **RELocation Read Only (RELRO)** : Configure des sections en lecture seule. Le "Full RELRO" est couteux lors du lancemant d'un programme car tout les symboles doivent être résolus avant le début de l'éxecution.
- **FORTIFY_SOURCE** : Protection contre les formats strings, présente au sein de la GLIBC (compilation avec -02)

Checksec

```
$ pwn checksec <binary>
[*] '<binary_path>'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Shellcode

Définition

Un shellcode est une chaine de caractères qui représente un code binaire exécutable.

Shellcode

Définition

Un shellcode est une chaine de caractères qui représente un code binaire exécutable.

Ecrire un shellcode

- Propre à chaque architecture cible
- S'écrit en assembleur directement (la plupart du temps)
- Attention aux caractères interdits (ex : \x00, filtrage ...)
- Ne doit pas être trop important

Liste de shellcodes : `http://shell-storm.org/shellcode`

Shellcode

Définition

Un shellcode est une chaine de caractères qui représente un code binaire exécutable.

Ecrire un shellcode

- Propre à chaque architecture cible
- S'écrit en assembleur directement (la plupart du temps)
- Attention aux caractères interdits (ex : \x00, filtrage ...)
- Ne doit pas être trop important

Liste de shellcodes : `http://shell-storm.org/shellcode`

Remplacé par la technique ROP

Exemple

```
int main(void)
{
    char shellcode_x64[] =
    "\x48\x31\xd2"                                  // xor    %rdx, %rdx
    "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov        $0x68732f6e69622f2f, %rbx
    "\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
    "\x53"                                          // push   %rbx
    "\x48\x89\xe7"                                  // mov    %rsp, %rdi
    "\x50"                                          // push   %rax
    "\x57"                                          // push   %rdi
    "\x48\x89\xe6"                                  // mov    %rsp, %rsi
    "\xb0\x3b"                                      // mov    $0x3b, %al
    "\x0f\x05";                                     // syscall

    (*(void (*)()) shellcode_x64)();

    return 0;
}
```

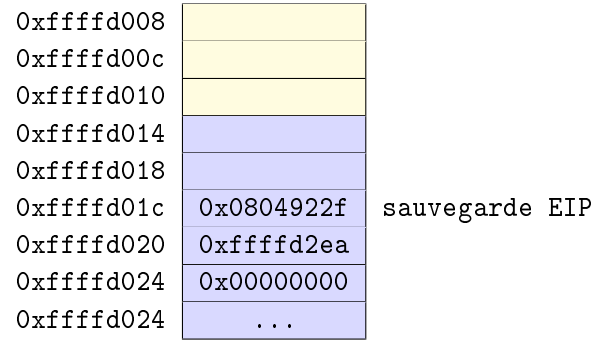
Sommaire

Le fil rouge

```c
// gcc main.c -o tp -fno-stack-protector -no-pie -z execstack -m32 -g
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void shell(){system("sh");}

void vuln(char *arg){
    char buffer[8];
    strcpy(buffer,arg);
    printf("Arg: %s\n", buffer);
}

int main(int argc, char **argv){
    char arg[0xff];
    fgets(arg,0xff,stdin);
    vuln(arg);
}
```

Exemple du cas nominal

Gestion de la mémoire si l'argument égal à "aaaaaa".



À l'appel de `vuln()`

Exemple du cas nominal

Gestion de la mémoire si l'argument égal à "aaaaaa".

0xffffd008		
0xffffd00c		
0xffffd010		
0xffffd014	0xf7fa8000	contexte (EBX)
0xffffd018	0xffffd028	contexte (EBP)
0xffffd01c	0x0804922f	sauvegarde EIP
0xffffd020	0xffffd2ea	
0xffffd024	0x00000000	
0xffffd024	...	

Après la sauvegarde de contexte

Exemple du cas nominal

Gestion de la mémoire si l'argument égal à "aaaaaa".

0xffffd008		buffer[8]
0xffffd00c		buffer[8]
0xffffd010		
0xffffd014	0xf7fa8000	contexte (EBX)
0xffffd018	0xffffd028	contexte (EBP)
0xffffd01c	0x0804922f	sauvegarde EIP
0xffffd020	0xffffd2ea	
0xffffd024	0x00000000	
0xffffd024	...	

Avant la copie

Exemple du cas nominal

Gestion de la mémoire si l'argument égal à "aaaaaa".

0xffffd008	0x61616161	buffer[8]
0xffffd00c	0x00616161	buffer[8]
0xffffd010		
0xffffd014	0xf7fa8000	contexte (EBX)
0xffffd018	0xffffd028	contexte (EBP)
0xffffd01c	0x0804922f	sauvegarde EIP
0xffffd020	0xffffd2ea	
0xffffd024	0x00000000	
0xffffd024	...	

Après la copie

Oui mais si ...

Gestion de la mémoire si l'argument égal à "aaaaaaaaaaaaaaaaaaaaaaa".

Oui mais si ...

Gestion de la mémoire si l'argument égal à "aaaaaaaaaaaaaaaaaaaaaaa".

0xffffd008	0x61616161	buffer[8]
0xffffd00c	0x61616161	buffer[8]
0xffffd010	0x61616161	
0xffffd014	0x61616161	contexte (EBX)
0xffffd018	0x61616161	contexte (EBP)
0xffffd01c	0x61616161	sauvegarde EIP
0xffffd020	0xffffd2ea	
0xffffd024	0x00000000	
0xffffd024	...	

Oui mais si ...

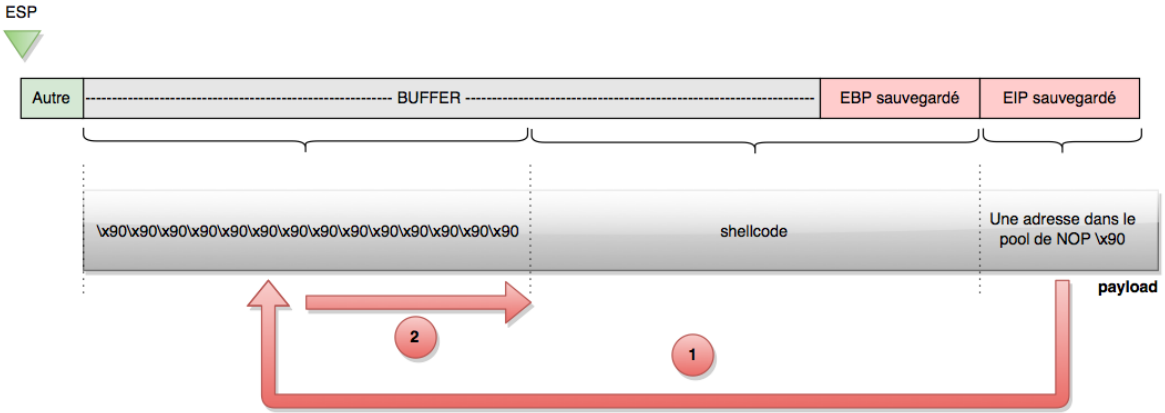
Gestion de la mémoire si l'argument égal à "aaaaaaaaaaaaaaaaaaaaaaa".

0xffffd008	0x61616161	buffer[8]
0xffffd00c	0x61616161	buffer[8]
0xffffd010	0x61616161	
0xffffd014	0x61616161	contexte (EBX)
0xffffd018	0x61616161	contexte (EBP)
0xffffd01c	0x61616161	sauvegarde EIP
0xffffd020	0xffffd2ea	
0xffffd024	0x00000000	
0xffffd024	...	

=> Segmentation fault (core dumped) car EIP = 0x61616161

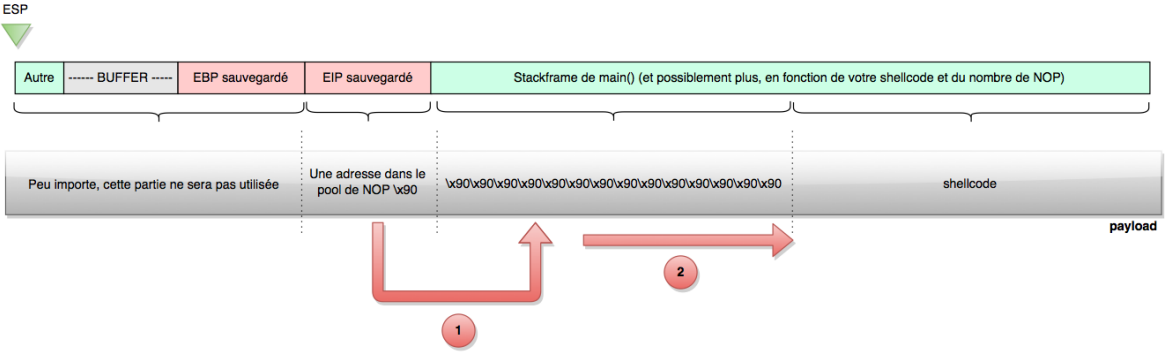
Démonstration

Shellcode : 1er scenario



Source : hackndo.com

Shellcode : 2iéme scenario



Source : hackndo.com

Démonstration

Causes

Attention à la gestion des tableaux

Ne jamais considérer comme acquis la vérification des bornes de votre tableau.

Table – Liste des fonctions à proscire et leur alternative sécurisée

UNSAFE	Safer alternative
atoi/atol	strtol
atoll	strtoll
atof	strtof/strtod
gets	fgets
strcat	strncat
strcpy	strncpy
sprintf	snprintf

Quelques commandes à retenir

GEF

- `checksec`, `gef run $_gef0`
- `pattern create 64`, `pattern search $eip`

pwntool

https://github.com/Gallopsled/pwntools-tutorial
https://docs.pwntools.com/en/stable/

- `p = ELF('./tp')`
- `p.functions['...']`, sections, symbols, etc...
- `cyclic(64)`, `cyclic_find(core.eip)`
- `shellcraft.sh()`
- `io = p.process([args])`, `io.interactive()`

Sommaire

Généralité

ROP : Return-oriented Programming

Définition

Technique de contrôle de la pile d'exécution utilisant une série de courtes instructions dans une zone exécutable du programme ciblé et s'affranchissant des protections binaires classiques.

- Technique publiée en 2007 par Hovav Shacham
  `https://hovav.net/ucsd/papers/s07.html`
- Turing complet (boucles, branchements conditionnels)
- Possible sous n'importe quel système (Linux, Windows, ARM)
- Ret2libc, Ret2plt sont des exemples de techniques ROP.

ROP : Return-oriented programming

Pour faire simple

Trouver et ordonner les bons blocs

Étapes (example pour set une valeur dans une adresse)

1. Visualiser le code attendu : `a = 0x1234ABCD`
2. Décomposer le code en instructions :
   1. `mov [reg1], reg2;`
   2. `pop [regX]` ; x 2
3. Trouver les bons gadgets avec `ropper` : `--search`, `--instructions`
4. Scripter votre payload avec pwntool

Le fil rouge

```
//gcc main.c -o tp -static -no-pie -fno-stack-protector -z execstack -m32 -g
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int secret=0x1234ABCD;

void shell(){
    if(secret==0xC0FFEE11){system("sh");}
}

void vuln(char *arg){
    char buffer[256];
    strcpy(buffer,arg);
    printf("Arg: %s\n", buffer);
}

int main(int argc, char **argv){
    char arg[0xffff];fgets(arg,0xffff,stdin);
    vuln(arg);
}
```

Exploitation

Nous devons modifier la valeur de `secret` par `0xC0FFEE11`.

Exploitation

Nous devons modifier la valeur de `secret` par `0xC0FFEE11`.

Utilisons une instruction `mov` : `mov [reg1], reg2`

Exploitation

Nous devons modifier la valeur de `secret` par `0xCOFFEE11`.

Utilisons une instruction `mov` : `mov [reg1], reg2`

Les registres seront initialisés avec avec l'instruction pop : `pop regX`

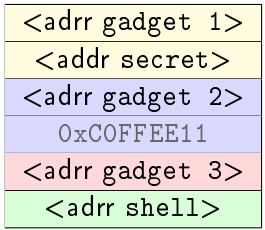
Exploitation

Nous devons modifier la valeur de `secret` par `0xC0FFEE11`.

Utilisons une instruction `mov` : `mov [reg1], reg2`

Les registres seront initialisés avec avec l'instruction pop : `pop regX`

Voici a quoi devra ressembler notre stack à la fin :



gadget 1 : `pop reg1; ret`

gadget 2 : `pop reg2; ret`

gadget 3 : `mov [reg1], reg2; ret`

et on lance la fonction `shell`

Démonstration

Simuler l'appel d'une fonction lors d'un ROP

Vous devez simuler le résultat du lancement dans votre pile

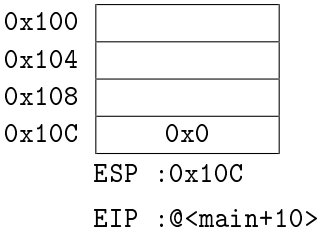
En fonction de votre convention d'appel, votre payload doit être identique aux instructions permettant de lancer une fonction

Simuler l'appel d'une fonction lors d'un ROP

Vous devez simuler le résultat du lancement dans votre pile

En fonction de votre convention d'appel, votre payload doit être identique aux instructions permettant de lancer une fonction

```
<main+10>: push arg2
<main+11>: push arg1
<main+12>: call func1
<main+13>: add ESP, 2*4
```

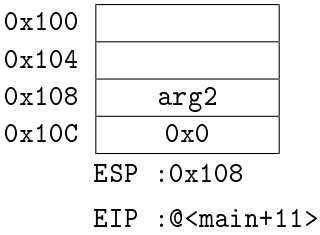


Simuler l'appel d'une fonction lors d'un ROP

Vous devez simuler le résultat du lancement dans votre pile

En fonction de votre convention d'appel, votre payload doit être identique aux instructions permettant de lancer une fonction

```
<main+10>: push arg2
<main+11>: push arg1
<main+12>: call func1
<main+13>: add ESP, 2*4
```

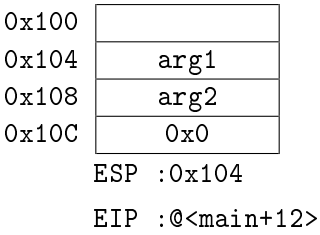


Simuler l'appel d'une fonction lors d'un ROP

Vous devez simuler le résultat du lancement dans votre pile

En fonction de votre convention d'appel, votre payload doit être identique aux instructions permettant de lancer une fonction

```
<main+10>: push arg2
<main+11>: push arg1
<main+12>: call func1
<main+13>: add ESP, 2*4
```



Simuler l'appel d'une fonction lors d'un ROP

Vous devez simuler le résultat du lancement dans votre pile

En fonction de votre convention d'appel, votre payload doit être identique aux instructions permettant de lancer une fonction

```
<main+10>: push arg2
<main+11>: push arg1
<main+12>: call func1
<main+13>: add ESP, 2*4
```

0x100	@<main+13>
0x104	arg1
0x108	arg2
0x10C	0x0

ESP :0x100

EIP :@<func1>

Simuler l'appel d'une fonction lors d'un ROP

Votre payload sera donc de cette forme (pour rappel addr_func sera pop lors de l'instruction "ret" à la fin de la fonction vulnérable à votre stackoverflow) :

```
payload = b"A"*offset + addr_func + addr_ret + arg1 + arg2
```

Enchainez les tous !

Si vous voulez lancer plusieurs fonctions à la suite ou poursuivre avec un gadget derrière, vous devez "pop" vos arguments avant de continuer. Donc l'addr_ret sera un gadget avec autant de pop que vous avez mis d'arguments.

```
payload  = b"A"*offset + addr_func + addr_gadget_pop_2 + arg1 + arg2
payload += addr_func2 + addr_ret + arg1
```

Ret2libc : Return to libc

La ligne `system("/bin/sh");` n'existe "pas" dans les programmes du quotidien. Dans le monde Unix, vous pouvez utiliser la libc pour `system` et trouver la chaine `/bin/sh`.

Exemple

```
payload = b"A"*offset + p32(libc_system) + addr_main + p32(libc_binsh)
```

Ret2plt : Exécuter n'importe quelle fonction de la PLT

Il est rare de posséder l'adresse de la libc lors de l'exécution du programme.
Mais grâce à la PLT (lien entre les fonctions des bibliothèques partagées et le programme), il est possible d'en exécuter certaines.
L'enjeu sera donc d'utiliser ces fonctions pour récupérer une adresse qui a été chargée au sein de la GOT.

Bon à savoir

- L'écart entre deux fonctions de la libc est toujours identique.
- Avec 2 adresses, il est possible de deviner la version utilisée `https://libc.blukat.me/`

Exemple

```
payload = b"A"*offset + plt_puts + addr_main + got_scanf
```

Pour résumer

- Le ROP c'est TOP ! Vous pouvez faire ce que vous voulez avec.
- Schéma classique d'un exploit ROP pour bypass l'ASLR :
  1. Récupérer une adresse de la libc grâce à un ret2plt de puts, printf ou send
  2. En déduire l'adresse de system et de /bin/sh
  3. Faire un ret2libc vers system() pour avoir un shell

Sommaire

Présentation

Origine

C'est la mauvaise utilisation de `printf` où une entrée utilisateur est passée directement. Ex : `printf(argv[1]);`

Les conséquences sont :

- Lecture arbitraire (`"%10$p"`)
- Écriture arbitraire (`"\xaa\xaa\xaa\xaa%3$n"`)

Rappel : les formats

- %x,X —> le word affiche en hexadécimal
- %d —> entier signée
- %u —> entier non signée
- %s —> chaine de caractéres
- %n —> Ecrit le nombre de caractéres écrits par printf dans le paramétre
- %<h|l|L>X —> Modification de la taille : h= short, l= long, L= long double
- %<n>$X —> Sélectionne le n-iéme argument, (("%3$d", var1, var2, var3) affiche var3)

Le fil rouge

```c
//gcc main.c -o tp -m32 -g
#include <stdio.h>
#include <unistd.h>

int main() {
    int secret = 0x59454b53;

    char buffer[128];
    printf("What is your name ? ");
    scanf("%s",buffer);
    printf("\nHello ");
    printf(buffer);
    printf("\nThe secret is %x.\n",secret);

    return 0;
}
```

Pour résumer

Préférer un code explicite pour l'affichage d'une chaine de caractères.

~~`printf(var);`~~ => `printf("%s",var);`

Sommaire

Étapes d'un stackoverflow

1. Repérer l'entrée utilisant une fonction vulnérable
2. Déterminer la taille de l'entrée nécessaire pour réécrire EIP
3. Exécuter une charge utile permettant de prendre la main sur le flux d'exécution :

A - Si aucune protection

1. Déterminer l'architecture cible
2. Créer un shellcode avant ou après EIP en fonction de la taille disponible

B - Sinon utiliser la technique ROP

1. Chercher les gadgets utilisables
2. Executer une chaine de gadgets

Et si l'ASLR est activé ?

Il faudra d'abord récupérer une adresse de la librairie ciblée à l'aide d'un format string (par exemple).

Connaitre son langage

- Ne pas utiliser des fonctions dangeureuses (fonctions ne vérifiant pas les buffers)
- Attention aux fonctions récursives (taille de la stack augmentant à chaque appel)
- Utiliser des pointeurs au lieu de copies de vos structures
- Mettre en place les protections des compilateurs (canary, stack nx, PIE, ASLR, warnings, etc...)

Régles d'or

- Toujours vérifier vos entrées et vos sorties
- Limiter le nombre de paramétres et la surface d'attaque

Changer de langage

- Prendre des langages interprétés comme Python3
- Utiliser des langages plus robustes dans la gestion de la mémoire comme le RUST

Attention

Chaque langage a ces particularités !

Par exemple, le RUST n'a pas de protection à 100% sûr :

Source : https ://rust-lang.github.io/rfcs/0560-integer-overflow.html

Rust, at least, does not have to worry about memory safety violations, but it is still possible for overflow to lead to bugs. Moreover, Rust's safety guarantees do not apply to unsafe code, which carries the same risks as C code when it comes to overflow. Unfortunately, banning overflow outright is not feasible at this time.

"Tester, c'est douter"

Le fuzzing

Jouer votre programme en testant beaucoup de possibilité d'entrées.

Les outils (liste non exhaustive) :

- https ://github.com/google/honggfuzz
- https ://llvm.org/docs/LibFuzzer.html
- https ://aflplus.plus/

Vérifier les fuites de mémoire

Quand il y a une fuite mémoire, c'est qu'il y a probablement un probléme d'allocation et de gestion de celle-ci dans le code

- https ://valgrind.org/

Et après ?

- Le monde de l'exploitation en x64 (ex : magic gadget)
- Les heapoverflow
- Les use-after-free

Sommaire

Généralité

La Heap

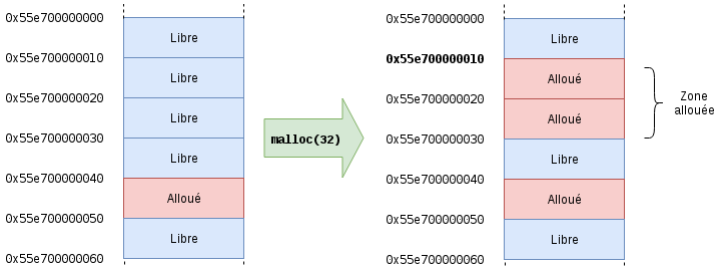
Espace mémoire permettant l'allocation dynamique. L'allocation peut se faire sur n'importe quel bloc mémoire quelque soit son adresse.

Généralité

La Heap

Espace mémoire permettant l'allocation dynamique. L'allocation peut se faire sur n'importe quel bloc mémoire quelque soit son adresse.

- malloc : alloue
- free : libére



Source : hackndo

Use after free

```c
#include <stdlib.h>

int main(int argc, char ** argv) {
    int *pA = NULL; int *pB = NULL;

    pA = malloc(4*sizeof(int));
    pA[0] = 42; pA[1] = 43;
    pA[2] = 44; pA[3] = 45;
    printf("pA = %p\n", pA);
    printf("pA[0] = %d, [...]");
    free(pA); pA = NULL;

    pB = malloc(4*sizeof(int));
    pB[2] = 22; pB[3] = 33;
    printf("pB = %p\n", pB);
    printf("pB[0] = %d, [...]");
    free(pB); pB = NULL;
    return 0;
}
```

Use after free

```c
#include <stdlib.h>

int main(int argc, char ** argv) {
    int *pA = NULL; int *pB = NULL;

    pA = malloc(4*sizeof(int));
    pA[0] = 42; pA[1] = 43;
    pA[2] = 44; pA[3] = 45;
    printf("pA = %p\n", pA);
    printf("pA[0] = %d, [...]");
    free(pA); pA = NULL;

    pB = malloc(4*sizeof(int));
    pB[2] = 22; pB[3] = 33;
    printf("pB = %p\n", pB);
    printf("pB[0] = %d, [...]");
    free(pB); pB = NULL;
    return 0;
}
```

```
pA = 0x9a0c1a0
pA[0] = 42, pA[1] = 43, pA[2] = 44, pA[3] = 45
pB = 0x9a0c1a0
pB[0] = 39436, pB[1] = 0, pB[2] = 44, pB[3] = 45
```

Les addresses libérées peuvent être réutilisé. Si le pointeur associé à l'espace libéré n'est pas réinitialisé, il y aura donc 2 pointeurs vers la même adresse. C'est une vulnérabilité de type Use-After-Free.