gitlab.enssat.fr

Alexandre DOYEN / CTF Logiciels sécurisés · GitLab

7-9 minutes

Beta Management Server

1) Contexte

Beta Management Server est un logiciel de gestion de classes qui est une version plus évoluée et censée être plus sécurisée que sa version précédente, Alpha Management Server.

Comme pour la version précédente, nous avons pu exploiter ce logiciel afin d'ouvrir un shell interactif sur le serveur l'hébergeant, et ce, malgré les quelques sécurités supplémentaires ajoutées.

2) Différences par rapport à la version précédente

Quand on décompile le logiciel sous Ghidra, on remarque la présence de plusieurs sécurités mises en place par le développeur :

- La présence d'une protection de type "canary" à la fin de chaque fonction. En effet, au début de la fonction, une valeur aléatoire est stockée sur la pile, et à la fin de celle-ci, il y a une vérification permettant de savoir si la valeur stockée sur la pile est bien la même que celle qui était mise initialement. Si ce n'est pas le cas, cela signifie que la pile a été modifiée de manière illégale (Stack Buffer Overflow par exemple), et donc, le programme plante par sécurité.
- La sécurisation de toutes les copies d'entrées utilisateur dans des tableaux : Désormais, lors des appels à la fonction `strncpy()`, les bonnes valeurs de tailles de tableaux sont passées en paramètre.

Aussi, quelques ajouts ont été effectués :

- La présence d'une fonctionnalité permettant de mettre à jour un étudiant
- La possibilité de redémarrer le serveur

Et en définitive, un fonction importante dans le cadre de notre attaque n'existe plus : La fonction `hide()` qui réalisait un appel `system("/bin/sh")`.

3) Vulnérabilités trouvées

Cependant, la fonction de mise à jour d'un étudiant possède une vulnérabilité majeure : En effet, il n'y a aucun contrôle sur un éventuel nombre maximum d'étudiants, la seule vérification faite étant que le numéro d'étudiant doit être positif.

```
if (idEleve < 0)
{
```

```
    puts(s_L'index_doit_tre_sup_rieur_0._0804a078);
}
else
{
    // Modification de l'étudiant...
}
```

Ainsi, il devient assez naturel pour un attaquant d'utiliser le numéro de l'élève comme vecteur d'attaque, dû au fait que les instructions suivantes font de la copie d'entrées utilisateur dans la pile. Ceci est dû au fait que le tableau `classe` est stocké sur la pile.

```
printf(s_Pr_nom_:_>_0804a09c);
fgets(buffer,30,stdin);
buffEnd = strcspn(buffer,"\r\n");
buffer[buffEnd] = '\0';
if (buffer[0] != '\0') {
  strncpy((char *)((int)classe + idEleve * 90),buffer,30);
}
// Pareil pour les champs "Nom" et "City"
```

Également, on a `nom = (classe + idEleve * 90 + 30)` et `city = classe + idEleve * 90 + 60`. Donc, en posant `idEleve = (addrCible - classe) / 90`, on peut modifier n'importe quelle valeur inscriptible dans la mémoire (Les valeurs négatives étant accessibles en prenant le complément à deux de la différence `addrCible - classe` et en divisant le résultat par 90, ce qui permet de contourner la vérification de positivité du numéro d'élève).

4) Attaque

4.1) Point d'entrée pour attaquer une sauvegarde de la valeur du registre EIP

Donc, un point d'entrée trivial est d'attaquer la sauvegarde de la valeur du registre EIP étant dans le cadre de l'exécution de la fonction `update()`. Cependant, ceci est impossible au vu du fait que la fonction `strncpy()` remplit les n octets du tableau de destination si la chaîne source n'est pas vide, et donc, va écraser le canary se situant avant la sauvegarde du contexte du registre EIP.

Cependant, en allant attaquer le cadre d'exécution de la fonction `server()`, qui se situe avant celui de la fonction `update()` dans la pile, il est possible d'attaquer une sauvegarde de la valeur du registre EIP, et ce, sans toucher au canary de la fonction `server()`. En effet, lorsque l'élève `0x1e`[1] (30 en décimal) est choisi, les écritures vont se faire autour de la sauvegarde du registre EIP sur la pile. Et en modifiant le champ "City", il se trouve que l'on modifie la sauvegarde de EIP sans toucher au canary. Donc, d'un point de vue du programme, le retour de fonction est légitime.

Après, en utilisant un motif généré par GDB et Gef, on trouve que le décalage permettant de modifier la sauvegarde de la valeur de EIP en utilisant ce champ est de 4.

Une fois "l'élève" modifié, il suffit de sélectionner l'option 5 dans le menu (Redémarrage du serveur) pour exécuter le code pointé par la nouvelle valeur contenue à l'emplacement de sauvegarde de EIP dans la pile.

4.2) Lancement d'un schell interactif

En l'abscence de fonction permettant de lancer un shell au sein du programme, il faut effectuer une attaque de type "Return-To-Libc" pour réussir à ouvrir un

shell interactif. Cette attaque est une variante de l'attaque "ROP" (Return Oriented Programming), qui consiste à construire des charges utiles se basant sur des retours de fonctions.

Cette attaque se réalise en deux temps :

- Dans un premier temps, il faut calculer l'adresse de base de la libc de son système. En effet, à cause de l'ASLR (Address Space Layout Randomization), il est impossible de savoir à l'avance sur quelles adresses va être associée la libc (Attaque "Return-To-PLT").
- Dans un second temps, il faut lancer la ROP-Chain qui va servir à lancer le shell interactif

4.2.1) Fuite de l'adresse de base de la libc

Pour faire fuiter l'adresse de base de la libc, il faut utiliser une fonction pour faire afficher une adresse contenue dans la GOT (Global Offset Table) qui contient les adresses des fonctions de la libc utilisées par le programme au sein du programme. Ces adresses sont utilisées lors de l'appel d'une telle fonction. En effet, la PLT (Procedure Linkage Table) contient des micro-fonctions qui vont se charger d'appeler les véritables fonctions de la libc. Par exemple, lors de l'appel à `printf()`, le programme va faire appel à `printf@plt()`, qui va faire appel à la fonction `printf()` de la libc en utilisant l'adresse de `printf()` qui est contenue dans la GOT, et qui pointe vers l'adresse du code de la fonction dans le programme.

Ainsi, il faut donc construire une charge utile qui va faire appel à `puts()`, avec comme paramètre l'adresse contenant une fonction de la libc dans la GOT.

Une fois la charge utile lancée sur le serveur, il faut récupérer les quatre premiers octets, qui correspondent à l'adresse de la fonction de la libc dans le programme. Et finalement, une fois cette attaque terminée, il faut retourner sur une fonction valide dans le code, sans quoi le programme planterait, et l'attaque serait inutile au vu du fait que les adresses sont replacées à chaque exécution.

Après, il suffit de calculer la différence entre l'adresse de la fonction dans le programme, et l'adresse de la fonction dans la libc, pour avoir l'adresse de début de la libc dans le programme.

Avec cette adresse de base, il est possible d'avoir d'adresse de la chaîne de caractères `/bin/sh`, et l'adresse de la fonction `system()` au sein du programme, en analysant la libc directement.

Ainsi, voici la charge utile utilisée :

```
addr_puts_plt + addr_main + addr_printf_got
```

4.2.2) Attaque

Finalement, pour lancer un `system("/bin/bash")`, il faut lancer cette ROP-Chain :

```
addr_system + addr_main + addr_bin_bash
```

Une fois cette ROP-Chain lancée, on s'apperçoit que l'on est face à un shell interactif.

5) Notes

1. `(0xfffe63bc (&ret_server) - 0xfffe58f0 (&classe)) / 0x5a = 0x1e` ↩