

La recherche de failles aujourd'hui

De nos jours la découverte de failles se monnaie, le simple fait d'écrire « Oday » fait grimper les prix de façon vertigineuse. Dans un tel cadre de tension il peut sembler bon de voir le fonctionnement de cet art tant convoité qu'est la recherche de failles.

I. Introduction

Si on remonte quelques années en arrière, les hackers prévenaient les fabricants logiciels à la découverte d'une faille. Les efforts fournis par l'attaquant étaient donc gratuits et la seule reconnaissance possible de son travail se faisait par le fabricant logiciel. Celui-ci ne corrigeait d'ailleurs pas toujours ses vulnérabilités (souvenons nous de Microsoft et sa faille MS08-028 qu'il a mis **1138 jours** à corriger – soit plus de 3 ans).

Puis est venu le temps du fulldisclosure, entendez par là « publication d'une vulnérabilité au grand public sans prévenir le fabricant logiciel ». L'objectif premier était de faire pression sur les fabricants pour qu'ils corrigent la vulnérabilité en question.

Nous observons une nouvelle tendance, plus commerciale mais aussi moins éthiquement correcte. A la découverte d'une faille le hacker contact le fabricant pour lui proposer les fruits de son travail contre rémunération. Dans le cas où l'entreprise n'est pas d'accord avec la condition de l'attaquant nous risquons de voir quelques jours plus tard la faille publiée à une vente aux enchères à Oday.

II. Les différentes recherches

Comme pour toutes les méthodes de recherches nous avons plusieurs voies pour arriver au même résultat. Dans cet article

II.2. Par fuzzer

Les fuzzers sont des applicatifs servants à éprouver de façon automatique la robustesse d'un logiciel. Nous pouvons les diviser en deux grandes catégories, les locaux et les réseaux. Dans un premier temps nous allons voir comment fonctionnent ceux présent sur les réseaux, puis nous nous attarderons sur ceux en local.

II.2.a) En réseau

L'objectif est de reproduire le comportement d'une application en modifiant certains paramètres ou saisies et étudier le comportement de l'application distante. Nous aborderons ici Spike, un utilitaire d'Immunity, entreprise de renom dans le milieu de la sécurité informatique.

Cet outil permet l'écriture rapide d'une suite de transferts réseau et ajustant manuellement les données devant être modifiées. Un exemple sera surement plus parlant qu'une longue explication :

```
s_string("HOST ");
s_string_variable("10.0.0.10");
s_string("\r\n");

s_string("USER ");
s_string_variable("Testeur");
s_string("\r\n");

s_string("PASS ");
s_string_variable("MotDePass");
s_string("\r\n");
```

Ici nous avons placé les premières lignes de communication du protocole FTP. Nous utilisons deux fonctions, `s_string()` et `s_string_variable()`. La première permet de définir une chaîne de caractères fixe et non modifiable par spike. La deuxième définit une chaîne de caractères pouvant être modifiée par le fuzzer. Cette modification prend plusieurs formes, elle peut remplacer la chaîne par une longue suite de « AAA », des

« %x », des « ../.. » , « C:\ » , etc. Dans notre exemple cela nous donne :

```
HOST /././AAAAAAA[...]AAAAAAA  
USER Testeur  
PASS MotDePass  
  
HOST /.../.../.../.../.../  
USER Testeur  
PASS MotDePass  
  
HOST C:\  
USER Testeur  
PASS MotDePass
```

Chaque communication avec le serveur teste un type de vulnérabilité, la base de Spike propose des centaines de combinaisons, ce qui agrandit d'autant notre champ d'action.

Le net avantage de cette méthode est que nous pouvons lancer plusieurs batteries de tests en même temps (par exemple tester plusieurs serveurs simultanément). Chose impossible à faire manuellement. Nous gagnons ainsi en efficacité sur tous les points.

Nous avons parlé ici des fuzzers logiciel mais nous pouvons également attaquer les protocoles eux-mêmes ! Par exemple appliquer des générations aléatoires de code sur le protocole IP. Les conséquences d'une telle vulnérabilité peuvent être désastreuses, étant donné que nous toucherions une large gamme d'ordinateurs avec probablement la possibilité d'exécuter du code en noyau. Si une faille est trouvée nous obtiendrons des droits bien plus élevés sur le système que l'administrateur lui-même.

II.2.b) En local

Cet aspect est souvent sous-estimé et donc moins testé, mais les vulnérabilités en local sont une réalité et peuvent être bien plus grave qu'à première vue.

Notre approche sera similaire à celle sur les réseaux mais cette fois appliquée aux fichiers. Notre objectif sera de modifier le contenu d'un fichier pour engendrer des bugs (voir plantage) de l'application propriétaire.

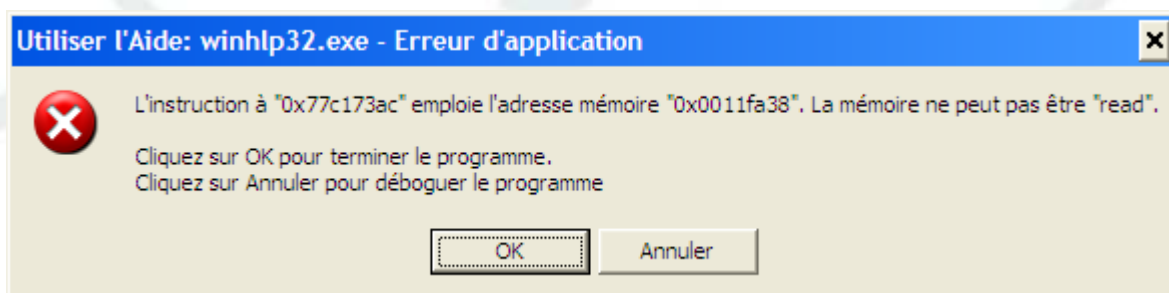
Nous allons encore une fois prendre un programme installé par défaut sur Windows, winhlp32.exe. Ce programme permet de lire des fichiers .hlp, ce sont des fichiers d'aide. Si nous ouvrons test.hlp nous pouvons voir que le début du code ressemble à ceci :

```
\x3F \x5F \x03 \x00 \xF3 \x03 \x00 \x00  
\xFF \xFF \xFF \xFF \x7A \x7F \x00 \x00  
\x6E \x0D \x00 \x00 \x65 \x0D \x00 \x00  
\x00 \xBE \x01 \x00 \x01 \xDA \x12 \x00
```

Ce sont les 32 premiers octets, appliquons maintenant une petite modification, par exemple remplaçons un octet par une valeur nulle (cette octet a été trouvée par un fuzzer). Ceci nous donnera alors le code suivant :

```
\x3F \x5F \x03 \x00 \xF3 \x03 \x00 \x00  
\xFF \xFF \xFF \xFF \x7A \x7F \x00 \x00  
\x6E \x0D \x00 \x00 \x65 \x0D \x00 \x00  
\x00 \xBE \x01 \x00 \x00 \xDA \x12 \x00
```

En lançant l'application winhlp32.exe nous obtenons le message ci-contre :



Nous avons alors au minimum un déni de service. Nous sommes en présence d'un heap overflow (débordement du tas), c'est une vulnérabilité difficile à exploiter, donc dans notre

cadre nous nous arrêterons à la simple découverte de cette faille par fuzzing.

II.3. Analyse comportementale

L'analyse comportementale se base elle sur les actions d'un programme. Elle utilise généralement la technologie des hooks (détournement de flux d'exécution lors d'un appel à une API) et est sensée étudier (ou tout du moins afficher) les différences entre les entrées aux fonctions et les sorties. Les fonctions à risques sont surveillées comme `sprintf`, `vsprintf`, `strcpy`, `wcscpy`, `strcat`, `wcscat`, etc. L'utilisation mal contrôlée de ces fonctions peut conduire à des vulnérabilités dans votre système. De plus il suffit de la négligence d'un seul développeur de dll pour mettre en péril la sécurité totale de l'application.

Je présenterai ici DumBug, un outil de phéonelite ayant déjà prouvé à maintes reprises ses performances. Son fonctionnement est relativement simple mais met bien en avant certains problèmes de programmation.

DumBug modifie toutes les adresses (de fonctions) à risque contenue dans l'IAT (Import Address Table). Ainsi quand un module désire utiliser une fonction exportée par un autre il passe par notre filtre. Ce filtre récupère les valeurs d'entrée et de sortie et les dumps dans un fichier. Il suffit donc de laisser l'application tourner avec un fonctionnement normal, puis d'analyser le fichier de sortie. Nous pouvons commencer la chasse aux failles. Dans ce fichier nous trouvons des informations très intéressantes comme l'exemple suivant :

```
7C80BA80 -> wcscpy(  
    wchar*      dest = [0012B3B8] = "",  
    wchar*      Src  = [00B160C8] = "/var/www/contact.php",  
0012B3B8 << wcscpy(  
    wchar*      dest = [0012B3B8] = "/var/www/contact.php" in  
stack of Thread,  
    wchar*      Src  = [00B160C8] = ""  
);
```

Nous voyons très clairement que la source est passée dans la destination. Nous pouvons aussi nous apercevoir que la destination est une adresse de la stack du thread. Nous avons

également la possibilité de modifier le nom du fichier à copier (l'application testée est Fillezilla version 2.2.24a). Nous avons alors de fortes chances d'être face à une vulnérabilité de type stack overflow, vulnérabilité critique pouvant amener à un contrôle à distance. Fort heureusement pour nous dans notre cas, le stack overflow (qui lui est bien réel) ne donnait accès qu'à un déni de service applicatif.

Nous pourrions améliorer cette technique en faisant ce qui s'appelle un hook inline. Un hook inline est le fait de réécrire une partie du code à exécuter d'une fonction pour la rediriger vers notre code. Ainsi même si la fonction est directement appelée (ce qui est une minorité des cas) nous filtrerons les entrées et sorties de celle-ci.

DumBug s'appuie sur la modification de l'IAT des processus (évoqué plus haut), mais il existe d'autres analyses comportementales. L'une d'entre elles se base sur la lecture des sources logicielles. Elle recherche les paramètres non vérifiés et émet une alerte si une variable non contrôlée est utilisée.

III. Conclusion

Nous avons fait un tour d'horizon sur les principales méthodes de découverte de vulnérabilités actuelles. Comme nous l'avons vu tout type de programme peut être vulnérable (des buffers overflow ont été trouvés dans php) et par conséquent cet aspect de la sécurité n'est pas à négliger. Vous connaissez maintenant différentes méthodes de recherche, libre à vous d'essayer celle qui vous convient le mieux.

Stéfan LE BERRE