Pratique

En mémoire aux timing attacks

Stavros Lekkas, Thanos Theodorides

Degré de difficulté

Le but de cet article est de ramener sur le devant de la scène le cas de l'analyse de l'exécution de synchronisation de chemins avec les daemons UNIX. Ce cas a été à ses débuts adressé par Sebastian Krahmer, un employé de SuSe, qui a également édité un article en 2002 expliquant les résultats possibles.

Dans cet article, nous décrirons comment exécuter l'analyse de synchronisation au-delà du chemin d'exécution d'un programme afin d'identifier les noms d'utilisateurs valides des services. UNIX (et pas seulement).

Nous proposerons d'abord quelques méthodes pour éliminer de telles possibilités sur un système. L'analyse du temps des tâches en informatique est une matière sur laquelle la recherche s'est appliquée pour plusieurs aspects. En général, il s'agit de calculer le temps de complexité d'un algorithme dans le but d'extraire des fonctionnalités spécifiques en résultat. De tels résultats peuvent être le temps nécessaire dont le programme a besoin pour produire le rendement d'une entrée spécifique d'une certaine plate-forme matérielle ou *Worst Case Execution Time* (Temps d'Execution le plus Mauvais).

Ces détails sont cruciaux pour les systèmes temps réel qui exigent de temps de réponse précis et rapide aussi bien que pour les systèmes intégrés et même pour le PC. Cependant, l'analyse au niveau du temps n'est pas exécutée seulement par des développeurs logiciels. Les fournisseurs *hardware* par exemple utilisent pour leurs processeurs graphiques ou non les méthodes d'analyse du temps de réponse puisque c'est l'un des points essentiels qui les rendra ou non compétitifs face à leurs concurrents.

La *Synchronisation Analysis* (analyse du temps de réponse) devient de plus en plus populaire dans la branche IT de la sécurité. Les chercheurs ont trouvé des méthodes sophistiquées pour détecter les portes dérobées (*backdoors*) au niveau du kernel via des *Synchronisation Analysis*, basé sur le principe que les programmes infectés avec du code

Cet article explique...

- Comment faire des suppositions valides en réalisant des analyses de synchronisation pendant l'exécution du chemin d'un programme.
- Comment identifier des noms d'utilisateurs valides.

Ce que qu'il faut savoir...

- Les bases de la programmation C.
- Statistiques de base.

malicieux exécute plus d'instructions – même si cela crée un temps plus complexe qu'une simple version du même programme.

En outre, l'analyse du temps de rendement d'un programme pour différents types d'entrées (tels que l'entrée d'un utilisateur existant et l'entrée d'un utilisateur inexistant) peut révéler des informations sensibles sur la configuration du système aux attaquants possibles fournissant des vecteurs d'attaque. Le but de cet article est d'expliquer comment une attaque de synchronisation peut être effectuée afin d'indiquer l'information sensible mentionnée ci-dessus et faire des conjectures dessus si un nom d'utilisateur (*username*) dans le système est valide. En plus de cela, concernant la partie pratique un prototype de programme de requêtes sera codé.

Qu'est-ce qu'une Execution Path Synchronisation Analysis ?

Un programme sur ordinateur est défini comme une suite organisée d'instructions, qui, une fois exécutées font que l'ordinateur agit d'une manière déterminée. Pour être plus précis, un programme est une procédure qui, quand elle reçoit des données en entrée aura le même type de données en sortie. Le temps nécessaire pour produire les sorties est connu comme time complexity (complexité de temps) du programme et cette complexité est similaire pour les programmes qui s'exécutent dans le même environnement et ayant les mêmes entrées (*input*)...

Comme nous le savons tous, le flux d'exécution de chaque programme est perpétuellement changeant dans le but de gérer correctement les entrées. Les instructions de programmation comme : `if-then-else (si-alors-sinon)` et `switch (selon cas)`, créer artificiellement – mais c'est possible- des carrefours affectant l'ordre d'exécution des instructions. Chaque manière différente de terminer le programme avec un résultat valide crée un chemin d'exécution. Naturellement, si le programme se compose de million de lignes de code, il peut y avoir des milliards de chemins d'exécution différents.

Chacun de ces chemins d'éxécution a son propre ensemble d'instructions a éxécuter. Ainsi il exige une quantité d'heures spécifique à éxécuter. La section de la science du développement d'algorithmes qui se doit de calculer cette quantité de temps s'appelle l'analyse de synchronisation d'exécution de chemin (EPTA), afin de comprendre EPTA, considérer le code d'un mécanisme imaginaire d'authentification comme à la Figure 1. Le mécanisme donne trois chances à un utilisateur qui veut s'authentifier sur le système, fournissant un mot de passe. La fonction: `valid_user()` retourne `0` si l'utilisateur n'existe pas et `1` si l'utilisateur existe. Les deux formes géométriques différentes sur le schéma 1 représentent l'ensemble des différentes instructions exécutées quand un utilisateur est valide ou non. Par exemple si l'utilisateur n'existe pas, le bloc d'instructions A est exécuté et ensuite la boucle for (boucle pour) exécute les instructions 1 à n fois. Dans le bloc A, les actions comme les tentatives de syslogging non réussies d'ouverture ou la désactivation de comptes utilisateurs peuvent avoir lieu si l'ouverture du login venait à échouer. Si l'utilisateur existe, le bloc d'instructions B est éxécuté et des actions comme : le binding (liaison) sur un *shell*. Cependant si un mauvais mot de passe est rencontré, la boucle for s'exécute 1 à n fois etc.

La Figure 2 illustre le graphe control-flux du code à la Figure 1.

Listing 1. *L'implémentation de calc_time()*

```
/* 0: */ long calc_time(char *username)
/* 1: */ {
/* 2: */    int n;
/* 3: */    struct timeval tvalue1, tvalue2;
/* 4: */    struct timezone tzone1, tzone2;
/* 5: */
/* 6: */    CLEAR(wBuf);
/* 7: */    gettimeofday(&tvalue1, &tzone1);
/* 8: */    snprintf(wBuf, sizeof(wBuf) - 1, "USER %s\r\n", username);
/* 9: */    write(socket_fd, wBuf, strlen(wBuf));
/* 10: */  CLEAR(rBuf);
/* 11: */  n = read(socket_fd, rBuf, sizeof(rBuf) - 1);
/* 12: */  gettimeofday(&tvalue2, &tzone2);
/* 13: */  return (tvalue2.tv_usec - tvalue1.tv_usec);
/* 14: */ }
```

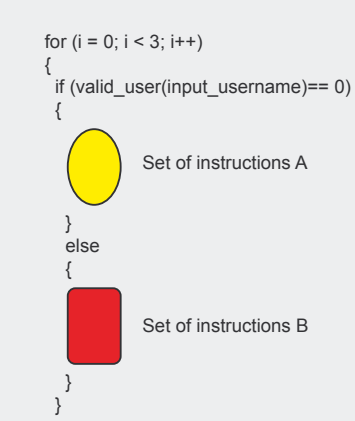

Figure 1 *Code Imaginaire d'authentification*

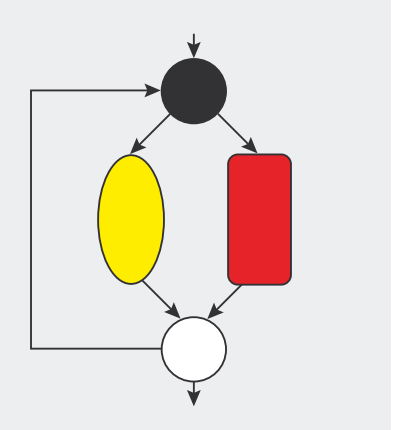

Figure 2. *Graphe Control-Flux*

Tableau 1. *Résultats d'éxécution*

Execution Case	A set of instructions	B set of instructions
First execution	30	50
Alternated execution	20	40
Consecutive executions	10	30

Une combinaison de l'ensemble des chemins d'éxécution est présentée à la Figure 3.

Considérez les événements suivants participant à un scénario de cas utilisant le mécanisme d'authentification que nous avons mentionné ci-dessus :

- Mécanisme d'authentification : il demande un nom d'utilisateur et nous entrons un utilisateur invalide. L'Auth-mécanisme réclame le mot de passe. Nous entrons un mot de passe aléatoire (n'importe lequel puisque l'utilisateur est interdit et l'authentification échouera de toute façon) et nous obtenons encore l'affichage du nom d'utilisateur (`i = 1` dans la boucle `for`).
- Nous écrivons un nom d'utilisateur valide et nous obtenons un message demandant un mot de passe. Nous entrons un mot de passe non valide pour ce nom d'utilisateur. L'authentification échoue et nous obtenons le message de demande encore une fois de l'utilisateur (`i = 2` dans la boucle `for`).
- On entre un nom d'utilisateur valide et un mot de passe valide. L'authentification réussit. Combinant le schéma 3 et le tableau 1, nous pouvons éxécuter l'analyse du temps du chemin d'éxécution (voir le schéma 4). Un scénario différent exigera un chemin différent d'éxécution ainsi l'analyse de temps résultera dans une valeur différente de temps. Le pire qui puisse se produire dans un vrai scénario est qu'un possible attaquant peut reconstruire le chemin précis d'exécution, recueillir des statistiques sur le temps de réponse et procéder à une attaque de synchronisation.

Qu'est-ce que l'Attaque de Synchronisation ?

Une attaque de synchronisation est une méthode pratique sous

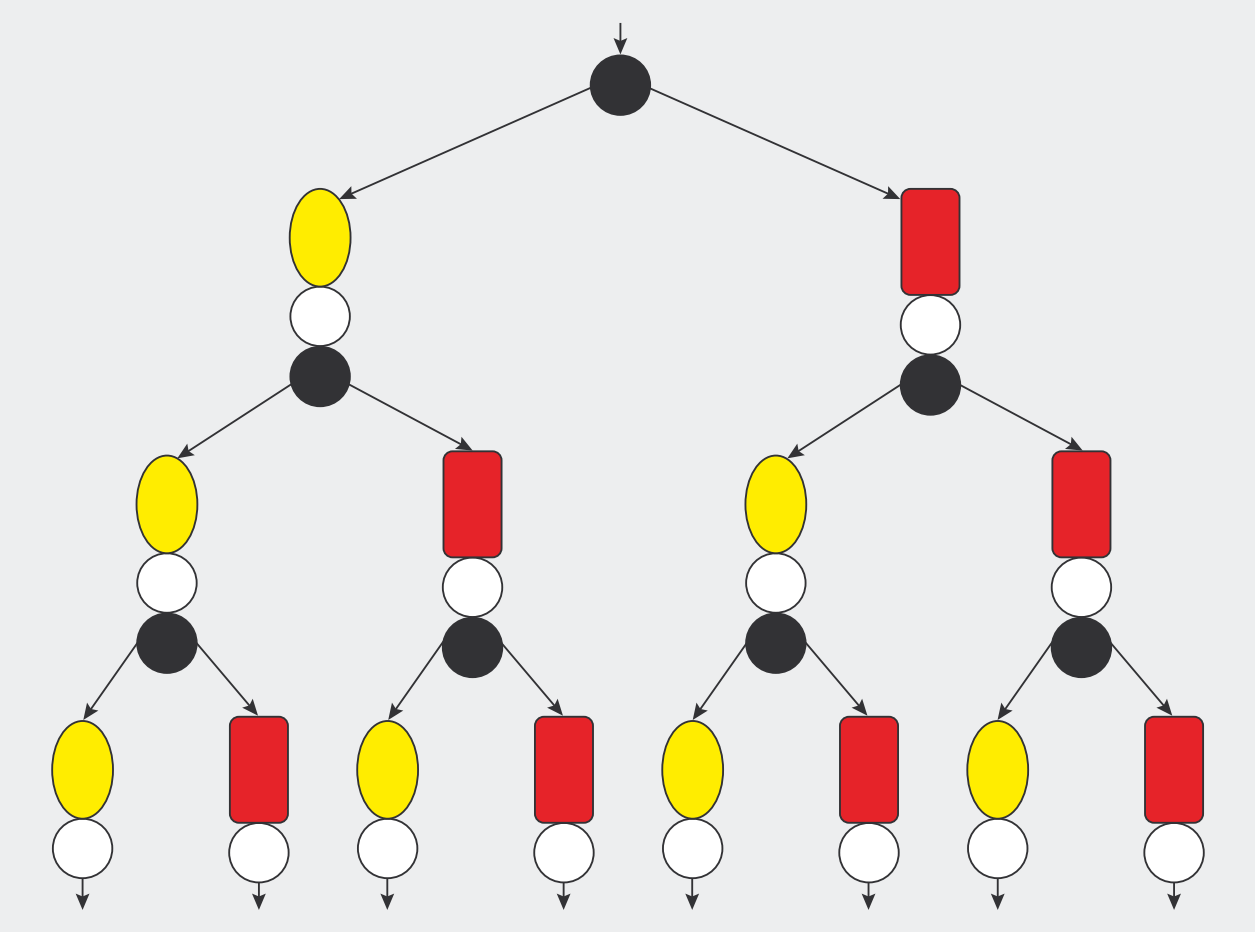

Figure 3. *Exposition des chemins après trois itérations (Pire des Scénarios)*

laquelle l'attaquant essaye d'extraire l'information en analysant le temps pris (jusqu'à une bonne précision) pour exécuter des parties spécifiques d'un algorithme. L'efficacité de cette attaque réside sur le fait que chaque opération dans un ordinateur prend du temps à s'éxécuter. La fuite d'informations d'un système peut être rendue possible au travers du temps pris par le système pour répondre à certaines requêtes.

Notez que si un algorithme est mis en application de manière que chaque sous-programme prend le même temps pour renvoyer des résultats, une attaque de synchronisation est impossible. En réalité, c'est quasiment infaisable car la plupart des implémentations sacrifient la sécurité de l'algorithme afin d'avoir des temps de réponse plus rapides en moyenne (qui est plutôt

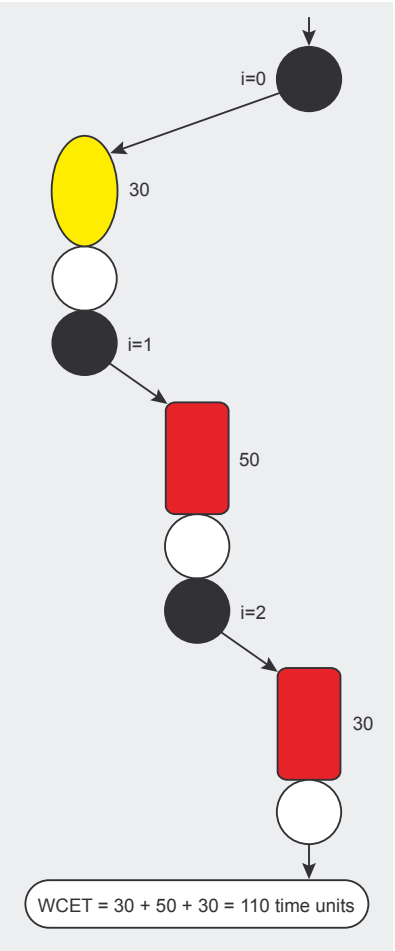

Figure 4. *Analyse du temps du mécanisme d'authentification*

Listing 2. *La prise de décision*

```
$ValidUser_Avg = check_user($valid_user, $host);
$GuessUser_Avg = check_user($check_user, $host);
$Factor = $ValidUser_Avg/$GuessUser_Avg;
if($Factor > 1.2) {
   print "[+] User ", $check_user," does not exist!\n";
} else {
   print "[+] User ", $check_user," exists!\n";
}
```

souhaitable pour des fournisseurs de logiciels). Les services comme : ftp, telnet, OpenSSH et probablement chaque service qui les module et que l'on peut relier à des authentifications d'utilisations (PAM), qui ne sont pas mis en application avec la possibilité d'attaque de synchronisation, sont vulnérables. Il faut maintenir à l'esprit que plusieurs de ces services sont bloqués jusqu'à un point (validation correcte d'entrée, gestion efficace de mémoire), la vulnérabilité, le cas échéant, d'une attaque de synchronisation, réside dans l'exécution.

Une Attaque de Synchronisation peut être réellement pratique quand on essaye de découvrir l'existence d'un utilisateur sur une machine hôte. Selon Krahmer (Voir le travail d'évaluation de Krahmer sur EPTA), des services comme ceux mentionnés ci-dessus classent le type de login comme :

- *Login valide* : Si le nom d'utilisateur et le mot de passe sont valides, l'utilisateur s'authentifie et les instructions additionnelles, par exemple un shell est exécuté ou un 2eme mécanisme d'authentification apparaît.
- *Login valide avec restrictions* : L'utilisateur existe mais bien que le username et le mot de passe soient corrects, l'utilisateur n'est pas autorisé à s'y logger. C'est possible si son compte a été suspendu, a expiré ou parce qu'il est énuméré dans un fichier d'interdiction (*deny file*). Par exemple, il peut être autorisé d'employer le ftp mais pas le sshd.
- *Login invalide* : L'utilisateur n'existe pas. Cependant, le service exige toujours un mot de passe de sorte qu'un attaquant ne sache pas que l'utilisateur est en fait invalide.
- *Login Special* : Le super-utilisateur entré, quelques dispositifs additionnels ont été éxécutés (*temps global additionnel*).

Les classes ci-dessus des logins ne sont pas manipulées de la même manière par chaque service. Par exemple, quelques serveurs de ftp peuvent exécuter avec plus de code pour un login invalide plutôt que pour un valide, alors que quelques serveurs de ssh peuvent faire l'inverse. Afin de faire des hypothèses correctes, une séquence spéciale d'essais est nécessaire. Cet ordre, la plupart du temps, est suffisant :

- Essayer de se logger avec un username valide pendant quelques fois. Pour chaque essai de login, essayez de mesurer le temps écoulé avant l'affichage d'un mot de passe.
- Refaites la même opération mais avec un mot de passe bidon. Par exemple : honorificabilitudinitatibus.
- Calculer la statistique moyenne ($\frac{\sum_{i=1}^{n} X_i}{n}$ for valid login et $\frac{\sum_{i=1}^{n} Y_i}{n}$

Listing 3. *L'implémentation de forget()*

```
void forget() {
   unsigned int time_slice;
   srand( time(time_t *)NULL) );
   time_slice = rand() % 31337+1;
   usleep(time_slice);
   return;
}
```

Tableau 2. *Resultats comme à la Figure 10*

User to guess	Class of user	Times probed	Accuracy of result
necro	Valid	10	100%
honorificabilitudinitatibus	Invalid	10	100%
root	Valid (super user)	10	100%
hakin9	Invalid	10	100%

Pour un qui est non valide, où les classes X et Y représentent le temps de réponse pour chaque essai valide et invalide respectivement) le temps de réponse pour un *login valide* et *invalide*.

- Essayer de vous logguer avec le username, don't vous voulez connaître l'existence, essayez de mesurer le temps de réponse et calculer la moyenne statistique. $\frac{\sum_{i=1}^{n} Z_i}{n}$ où Z est le temps de réponse de chaque essai).
- Si le temps de réponse moyen pour l'username que vous avez essayé est plus près de la moyenne d'ouvertures valides, il est presque sûr (aussi honnête que les statistiques peuvent l'être) que l'username testé est valide aussi. Si c'est probablement plus près de la moyenne des non valides alors il doit être invalide. Si le nombre est loin des 2 autres, alors peut-être l'utilisateur est classé dans une autre catégorie (par exemple compte expiré) ou un facteur externe a changé vos résultats.

Des facteurs externes pourraient être le traffic produit par un processus en tâche de fond, une perte soudaine de bande passante ou une perte de paquets sur le réseau, des temps d'attente ou des temps de chargement CPU excessifs. Pour éliminer l'exposition des facteurs externes comme ces derniers, assurez-vous du bas-latence d'utilisation, gardez la même charge d'ordinateur et tuez les processus inutiles fonctionnant en tâche de fond.

Évaluer les travaux de Krahmer sur l'EPTA

L'EPTA des daemons Unix, comme décrit par Sebastian Krahmer (dans ses travaux), peut être considéré comme de grande valeur au sens où il a décrit beaucoup de vecteurs d'attaques contre des ordinateurs. Disons simplement, qu'il explique le fait que tout ce

que nous pourrions faire, aurait lieu sur une flèche du temps et donc qu'on peut y tracer les évènements relatifs. Cela a été l'une des premières tentatives pour expliquer comment prendre des empreintes de structures de fichiers distants en mesurant le temps des événements corrélés qu'ils produisent pendant leur exécution. Assurément, cet article a établi la situation actuelle dans l'analyse de synchronisation de chemin d'éxécution des daemons UNIX.

Attaquer un service PAM

Il est temps de mettre en oeuvre la pratique. Pour couvrir l'aspect pratique de cet article, les auteurs ont décidé d'effectuer une attaque de synchronisation (*synchronisation attack*) contre le logiciel largement utilisé de serveur FTP : ProFTPD 1.3.0. Des versions plus récentes de ce daemon ont souffert de problèmes de synchronisation. Ses concepteurs ont présenté un nouveau module, appelé `mod_delay`, pour fixer leur serveur sur ce type d'attaques. Cependant, quand nous avons configuré le daemon dans nos essais,

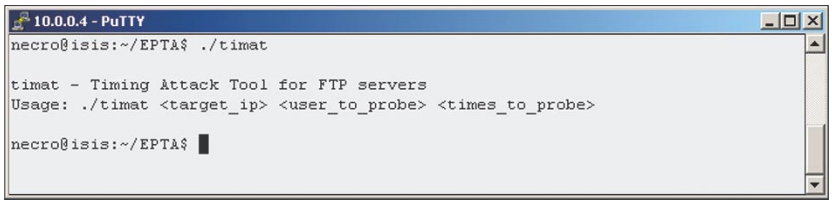

Figure 5. *Comment utiliser le timat*

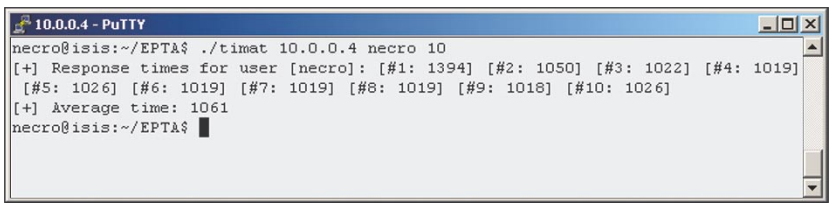

Figure 6. *Demande de l'utilisateur necro (valid)*

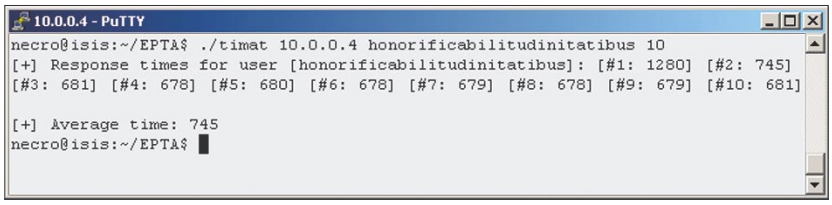

Figure 7. *Demande de l'utilisateur honorificabilitudinitatibus (invalid)*

Note 1

Le serveur fonctionnait sur une installation 10.1 Slackware Linux par défaut (version du Noyau: 2.4.29) utilisant un Pentium II 334Mhz (Intel) avec un cache de : 512Kb et le 256MB de RAM. La connexion réseau était un standard Ethernet 10Mbit sous un environnement LAN.

nous avons désactivé ce module pour que le concept ait lieu correctement ; nous sommes certains qu'il reste des versions plus anciennes et vulnérables du serveur qui subsistent. Afin de démontrer l'attaque, un outil prototype appelé timat, a été développé. Il s'agit d'une implémentation d'une liste d'étapes décrites dans : Qu'est-ce qu'une attaque de synchronisation (*timing attack*) ? La fonction qui exécute les instructions importantes est : `calc_time ()` (voir la Listing 1).

Aux lignes 3 et 4, nous créons 2 références aux structures `timeval` et `timezone` étant donné que l'on doit conserver 2 valeurs de temps. La première valeur `tvalue1` possède la valeur de timestamp initial (juste avant que nous demandions le username) et la deuxième : `tvalue2`, possède la valeur de timestamp final (juste après que le serveur FTP réponde avec le Mot de passe: à la ligne 11). L'appel de la fonction `gettimeofday()` aux lignes 7 et 12 sont responsables de la sauvegarde des 2 valeurs de timestamp. La fonction `calc_time()` retourne (ligne 13) la différence entre les 2 valeurs en microsecondes. Le troisième argument que l'outil reçoit définit combien de fois la fonction calc_time devrait être appelée depuis une boucle for. Rappelez-vous, nous devons demander le nom d'utilisateur (*username*) plusieurs fois afin d'avoir quelques valeurs significatives pour pouvoir calculer la moyenne statistique. À ce moment là, la seule chose que nous avons à disposition est un outil qui calcule cette valeur moyenne de temps. Comment cette valeur va-t-elle nous aider à découvrir si un utilisateur existe ou pas ? Des suppositions peuvent être faites en comparant la période moyenne d'un utilisateur valide et la période d'un utilisateur non valide (problème de classification binaire), mais ceci prend pas mal de temps particulièrement dans le cas où nous souhaitons découvrir beaucoup d'utilisateurs. Pour simplifier ce procédé, un script Perl appelé *pr0ber.pl* a été codé pour faire les suppositions à notre place.

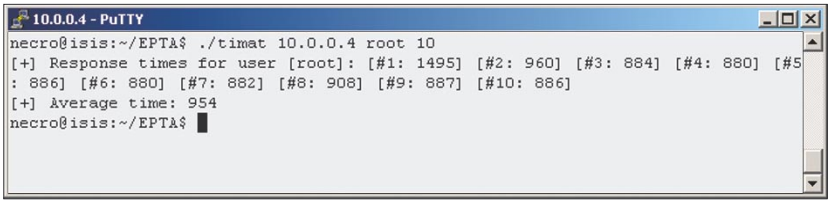

Figure 8. *Demande du : super-user root (obviously valid)*

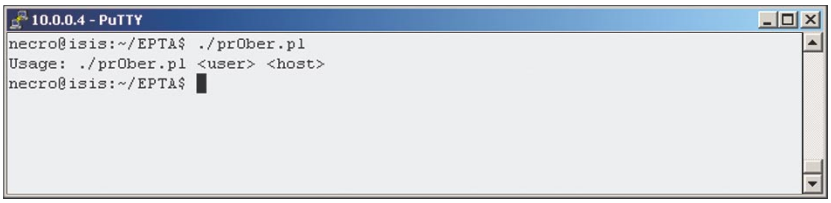

Figure 9. *Comment utiliser : pr0ber*

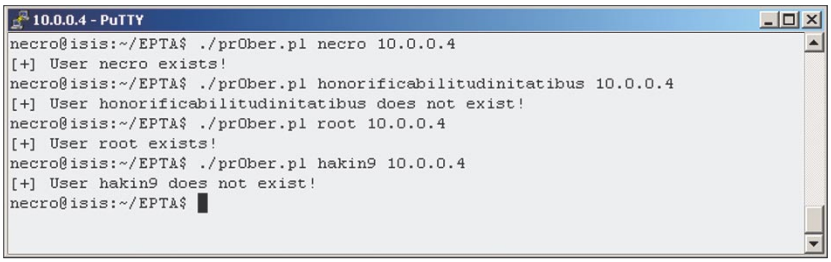

Figure 10. *Suppositions pour les utilisateurs: necro, honorificabilitudinitatibus, root et hakin9*

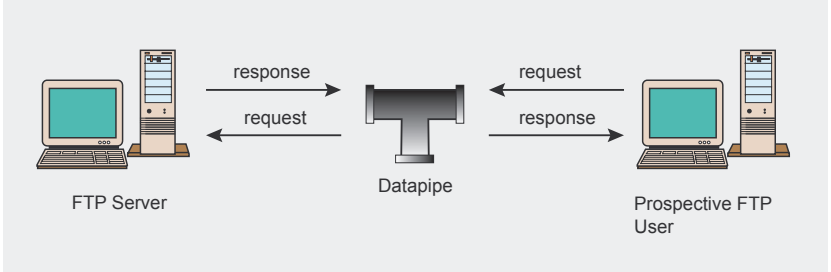

Figure 11. *Un Data pipe entre un service et un utilisateur*

`pr0ber` utilise l'outil timat pour calculer les temps moyens. Il demande un utilisateur valide par défaut (*root* devrait être Ok) et vérifie l'existence d'un utilisateur invalide. Après avoir obtenu les moyennes de temps du timat, il en calcule le facteur. $\frac{ValidUser_{Average}}{ValidUser_{Average}}$ ProFTPD 1.3.0 a besoin de moins de temps pour gérer un utilisateur invalide plutôt qu'un utilisateur valide, alors nous savons que si le facteur est plus grand que 1 (`GuessUser_Average<ValidUser_Average`) alors l'utilisateur que nous avons testé est certainement invalide. Pour réduire au minimum les chances d'une hypothèse se révélant fausses, nous avons augmenté le seuil du facteur à 1.2 (qui a renvoyé un succès

de 100% pendant l'essai). Une partie du code de Perl est montrée dans le Listing 2.

La procédure `check_user()` utilise: timat pour avoir le temps moyen pour chaque utilisateur pendant que : `$valid_user` est initialisé à : *root* et `$check_user` est le premier argument du script. Pour voir les outils en action, regardez ci-dessous les résultats d'attaques.

Méthodes de mesure des périodes de temps

Je vous recommande de faire attention temps : des minutes qui peuvent prendre des heures...

Philip Dormer Stanhope – *4th Earl of Chesterfield*, *Letters to His Son*

Généralement il y a au moins deux manières possibles de mesurer combien de temps certaines instructions ont besoin pour s'exécuter. La première est la fonction `gettimeofday()`. Elle fournit une grande flexibilité et précision au niveau des microsecondes, décrivant le délai de manière tout à fait précis que nous, humains, ne pourrions à peine déceler. Il est tout aussi simple de garder 2 valeurs de temps renvoyés par la fonction `gettimeofday()` puis de calculer leur différence.

La deuxième méthode est d'utiliser les : time-ticks. Sebastian Krahmer, dans son article (EPTA des daemons UNIX), définit les time-ticks comme le nombre d'appels à `read()` jusqu'à ce que la réponse soit lue. Pour avoir une vision plus claire, ce qui suit est un extrait de code du patch de Sebastian Krahmer pour OpenSSH qui emploie les time-ticks (au lieu de `gettimeofday()`)pour calculer le délai.

```
while (read(peer, dummy,
      sizeof(dummy)) < 0) {
   ++reads;
}
return reads;
```

reads est une variable entière t incrémentée constamment jusqu'à ce que la fonction `read()` soit mise à 1, ce qui signifie que nous sommes incités à entrer un mot de passe. Évidemment, un utilisateur invalide produit des nombres différents de read que celui valide, ainsi on peut considérer cela comme une mesure concurrentielle alternative. En conclusion, Krahmer est tellement clair dans son article qu'il n'y a rien d'autre à ajouter dans cette section, gentiment décrit par lui comme : Choosing the right clock, dans son article.

Résultats d'attaque

Comme décrit en Figure 5, timat requiert 3 paramètres en entrée. Le premier est l'adresse IP du hôte qu'on a à l'esprit, qui dans notre test était mis à : 10.0.0.4 ; Le deuxième est un utilisateur à tester et au final un entier non signé pour définir le nombre de demandes.

Les Figures 6-8 montrent l'outil en action, demandant l'utilisateur : necro, honorificabilitudinitatibus et root qui sont respectivement valide, invalide et valide. Il est clair que les utilisateurs invalides ont besoin de moins de temps pour être gérés que le super-utilisateur ou un utilisateur valide qui prennent un peu plus de temps, mais avec une différence significative. Noter que nos mesures de temps se font en micro-secondes (1 microsecond = 1 × 10-6 seconds).

Le schéma 8 présente notre outil de paquetage, *pr0ber.pl* (cf. attaquer un service fait sur commande de PAM), qui met en application le processus décisionnel concernant l'existence d'un utilisateur. Le premier paramètre de `pr0ber` est l'utilisateur que nous voulons deviner. Le schéma 10 montre l'outil en action pour différents utilisateurs valides et invalides.

Comme vous pouvez le voir dans la Tableau 2, il y a 100% d'efficience pour l'ensemble des utilisateurs, ce qui est une excellente performance si vous considérez que nous faisons les suppositions sur des éléments statistiques. En fait, même pour moins de 10 demandes (requêtes), l'exactitude des résultats restait de 100%. Cependant dans des scénarios réels, hors de l'environnement d'un essai, les choses pourraient paraître différentes. Les deux choses qui garantiront des résultats précis, dans ce cas, sont de plus grands échantillons pour chaque sondage (par exemple plus de 10 par utilisateur) et un calcul efficace du seuil de facteur dans *pr0ber.pl*.

Contre-mesures

Il est facile d'exécuter des attaques de synchronisation, mais il est également tout à fait possible de se protéger contre elles. Généralement il y a deux méthodes de faire cela. Tandis que la première est plutôt théorique, l'autre est beaucoup plus pratique. La manière théorique a l'inconvénient d'affecter l'exécution globale de l'application et est plus difficile à mettre en application. Le concept principal est que le code qui gère l'authentification et les instructions, qui affectent le flux, devraient être équilibrés. Ceci assurerait, à un certain moment, que le programme réponde en même temps pour chaque classe d'entrée. Cependant cet équilibre 1 à 1 exige des compétences en programmation que personne n'a.

Délais aléatoires

Introduisant des délais aléatoires comme time-patches est une autre technique qui produit les mêmes résultats. Vous implémentez la partie authentification de la même façon que vous feriez normalement.

Note 2

Le modèle de data pipe présente des overheads (temps passé par un système à ne rien faire d'autre que se gérer), celui de l'exécution de ses propres instructions. Cependant sur une base théorique, ces overheads ont lieu sur un temps constant et ne posent donc aucun problème.

Ensuite, vous calculez le temps de réponse exact pour chaque classe de login (cf : Qu'est-ce qu'une attaque de synchronisation ?). Vous gardez le temps de réponse le plus élevé comme borne supérieure et vous forcez les sous-programmes qui manipulent les autres classes en entrée à atteindre cette limite. Ceci peut être facilement fait en utilisant la fonction `usleep()` de libc, bien que la partie la plus dure soit de calculer les temps de réponse avec précision. Résultat ? Celui escompté. Chaque classe d'entrée a besoin du même délai, pour se produire.

La pratique est tout à fait similaire à la théorie mais elle est plus difficile et abstraite. L'idée est simple, vous entrez un pseudo aléatoire avec un délai confortable avant chaque réponse, de manière à ce que cela soit impossible pour un attaquant de deviner les *time-patches*. L'ensemble aléatoire du générateur de nombre peut prendre des valeurs de la fonction `gettimeofday()` de libc. Alternativement, */dev/urandom* rassemblera les éléments de l'environnemnt des modules de gestion de périphérique et créera un ensemble entropique satisfaisant afin de créer des nombres aléatoires. Le nombre aléatoire généré sera utilisé comme paramètre pour la fonction usleep() pour créer le délai. Gardez à l'esprit que l'inactivité sur un certain période de temps affectera dramatiquement la performance d'une application. Si vous prévoyez d'utiliser PAM, ne faites pas confiance à votre imagination pour produire des délais aléatoires. Vous pouvez utiliser la fonction pam_fail_delay() implémenté dans : *security/pam_modules.h*. Soyez sûr d'utiliser cette fonction avant chaque réponse de votre programme ou autrement les attaques par synchronisation resteront possibles, au moins quand il s'agira d'avoir des utilisateurs valides. Pour votre confort, les patches de sécurité kernel comme `GrSecurity` sont assez sophistiqués pour inclure des mécanismes de sécurité pour les authentifications.

Les Data pipes (Données en Tube)

Choisir la méthode mentionnée ci-dessus de délais aléatoires pour un accompte, peut fournir nombreux différents types d'architectures. L'un d'entre eux pourrait inclure le modèle de réseau d'un Data pipe. Un *Data pipe* (Données en tube) est un programme qui réside entre un utilisateur et un service, comme vous le voyez sur le schéma 11. Son rôle est de transmettre des données reçues de l'utilisateur directement au service et vice-versa. Il joue donc un rôle intermédiaire et il peut ainsi contrôler les synchronisations relatives de transmission de données (conseil !).

Bien que la pipe de données puisse être installée sur un ordinateur tiers, on suggère qu'il devrait fonctionner sur le même ordinateur que le service dans le cadre de la politique stricte suivante. Le ftp server est réorienté à un port autre que 21 et ce port doit être filtré par un firewall de sorte qu'il soit complètement invisible de l'Internet. La pipe de données devrait fonctionner sur le *port original de service* (imitant son comportement) et doit avoir accès à l'Internet et au service.

Évidemment, tous les utilisateurs éventuels ignorent son existence et pensent qu'ils communiquent directement avec le service. L'élément clé de ce concept est de retarder la réponse finale à l'utilisateur de sorte que l'ensemble de la session ne puisse pas être sujettée à une analyse précise de synchronisation. Ceci peut être réalisé en utilisant la fonction du Listing 3.

Cette fonction devrait être appelé juste avant d'envoyer la réponse en retour à l'utilisateur.

Conclusion et remarques

EPTA est une technique valable qui aide beaucoup de réalisateurs pour fournir des solutions optimales. Il est également possible d'aider le côté obscur, car quelqu'un pourrait l'employer pour faire de la rétro-ingénierie d'un objet (pour en déterminer le fonctionnement) jusqu'à un certain point (par exemple s'étendre sur l'exposition d'un utilisateur valide jusqu'à la corruption totale d'un système de cryptographie asymétrique). Afin de contrecarer et surmonter de telles attaques, de bonnes compétences en programmation sont nécessaires.

Einstein a mentionné que le temps peut être une quatrième dimension, une dimension avec des propriétés différentes que celles de l'espace, évidemment. Le temps est une illusion, quelque chose de totalement inventé par les êtres humains juste parce que c'est une manière facile d'identifier des changements dans notre montre visiblement spatiale. Illusion ou pas, nous pouvons sans risque soutenir que cette entité invisible contient beaucoup d'informations descriptives sur des événements et est de fait un compagnon utile dans notre réalité. ●

À propos des auteurs

Stavros Lekkas, originaire de Grèce, est un étudiant MPhil à l'Université de Manchester (autrefois connu comme : UMIST). Ses passions incluent : la cryptographie, la sécurité des informations, le data mining (extraction et étude de données), les mathématiques (logique, théorie des nombres et l'algèbre linéaire) et la complexité informatique. Il travaille actuellement sur sa thèse qui traite de L'Évolution des Systèmes Intelligents de Détection d'Intrusions.

Thanos Theodorides, également de Grèce, étudie les Ordinateurs, Réseaux et l'ingénierie des Telecommunications à l'Université de Thessaly, Grèce. Fervent passionné de la sécurité informatique depuis son plus jeune âge, ses passions incluent le développement et la sécurité Web, les réseaux sans fil et les systèmes d'exploitation réseau, parmi d'autres. Pendant son temps libre, il prend plaisir à créer des travaux d'art digitaux.