HERVÉ SCHAUER CONSULTANTS
Cabinet de Consultants en Sécurité Informatique depuis 1989
Spécialisé sur Unix, Windows, TCP/IP et Internet

Le Web 2.0 : Plus d'ergonomie... et moins de sécurité ?

Journée Sécurité des Systèmes d'Informations
OSSIR
22 mai 2007

Renaud Feil
prenom . nom @ hsc . fr

Sommaire de la présentation

- Le nouveau modèle de développement du Web 2.0 et son impact sur la sécurité.
- Retours d'expériences sur les vulnérabilités fréquemment rencontrées dans les applications Web 2.0.
- Le rôle des outils de développement dans la sécurité du Web 2.0.
- Les solutions concrètes pour améliorer la sécurité dans les applications... et leurs limites.

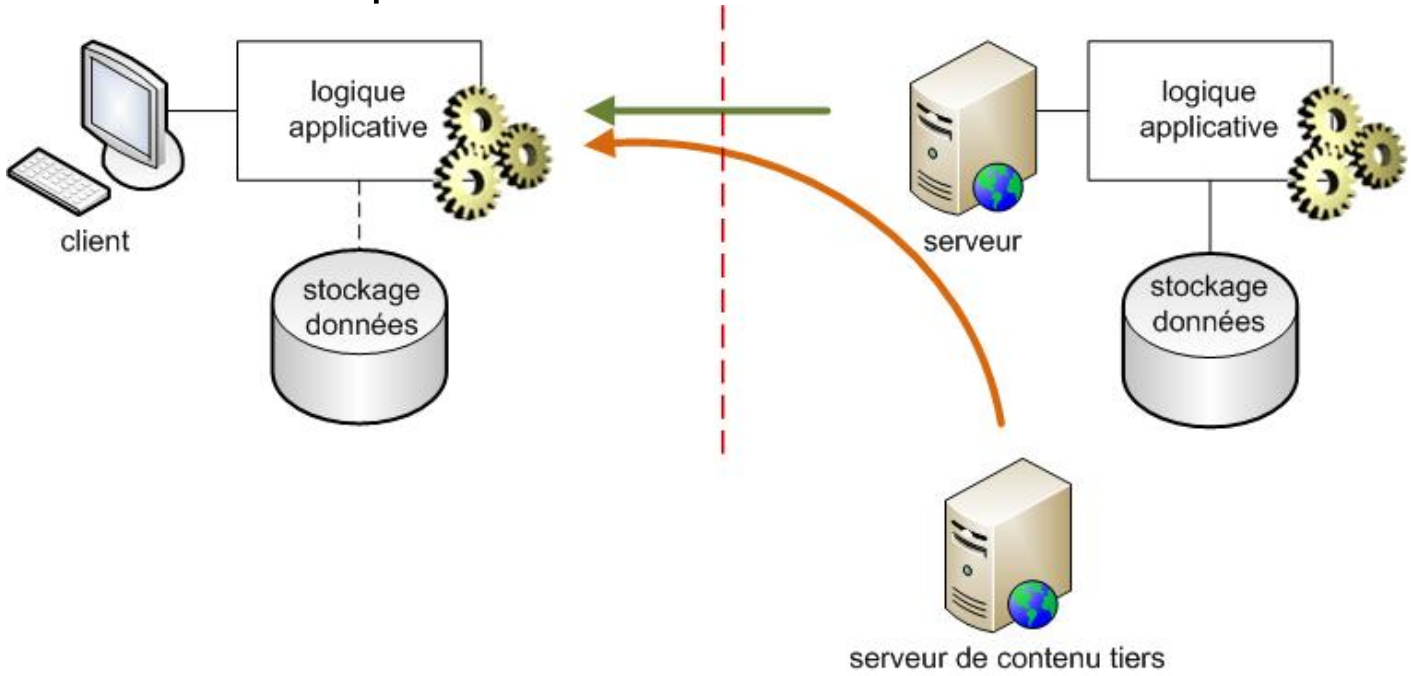
Partie 1 :
Le nouveau modèle de développement du Web 2.0 et son impact sur la sécurité

Les limites ergonomiques du Web 1.0

- Web 1.0 : Des limites ergonomiques intrinsèques
  - Pour chaque action : effacement de la page HTML en cours, réalisation d'une requête synchrone vers le serveur et affichage de la nouvelle page HTML.
- Ne permet pas de concurrencer les applications clients lourds en terme d'ergonomie.
  - Mais les applications clients lourds ont d'autres inconvenients (déploiement, maintenance, sécurité, etc...).
  - Il faut passer au Web 2.0 !
- Web 2.0 :
  - Mêmes briques de base que dans le Web 1.0 : HTML, CSS, *Javascript*.
  - Ajout des XMLHttpRequest (depuis *Internet Explorer 5*).
  - Paradigme de développement différent : « Avec le Web 2.0, on a enfin compris comment combiner efficacement HTML, les CSS et *Javascript* ».

Le Web 2.0 : un nouveau modèle de développement

- Amélioration de l'ergonomie en déplaçant une partie de la logique applicative vers le navigateur :
  - Traitements côté client en Javascript.
  - Stockage de données persistentes : *globalStorage* sous *Firefox*, *userData behavior* sous *Internet Explorer*.
- Intégration de nombreux formats multimédias.
- Agrégation de contenu provenant de serveurs tiers.



L'impact de ce nouveau modèle pour la sécurité

Constats	Risques
Déplacement de traitements et de données vers le navigateur	Modification des traitements et observation des données par un attaquant
Architecture applicative et interactions entre les composants plus complexes	Vulnérabilités dues à des erreurs de conception dans l'architecture applicative
Support de nombreux formats de données	Vulnérabilités dans le navigateur et ses extensions (surface d'exposition élevée)
Impossible de désactiver Javascript pour naviguer sur les sites Web 2.0	Scripts hostiles utilisant les fonctionnalités Javascript standards
Syndication de contenu provenant de serveurs tiers	Compromission en cascade si le contenu syndiqué est compromis
Interfaçage facile d'applications internes avec des services tiers	Externalisation d'informations sensibles « sans le savoir »

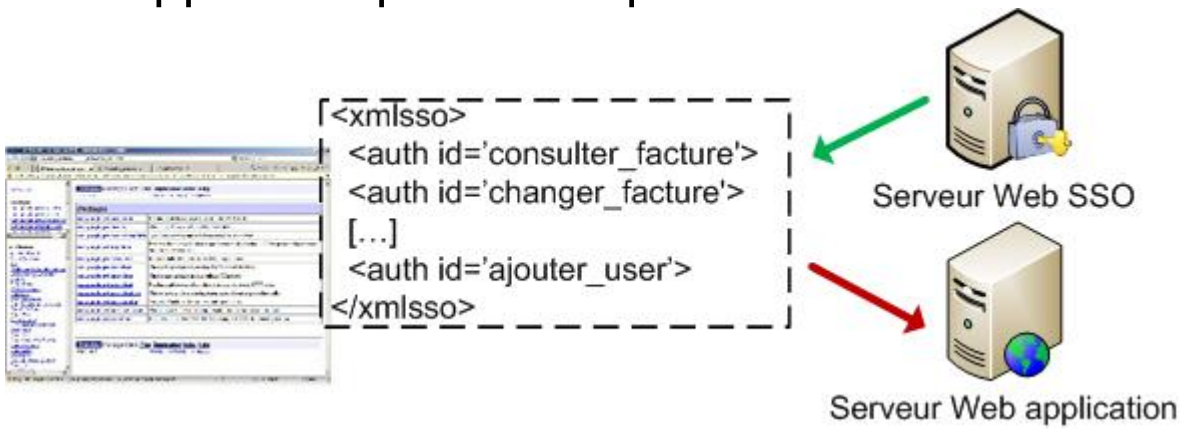
Partie 2 :
Retours d'expériences sur les vulnérabilités fréquemment rencontrées dans les applications Web 2.0

Causes des vulnérabilités propres au Web 2.0

- Volonté d'ergonomie :
  - Déplacement de traitements et de données sensibles vers le navigateur.
  - Suppression de certains contrôles de sécurité « pour ne pas diminuer la réactivité de l'interface ».
- Mauvaise utilisation des formats et protocoles :
  - Possibilités d'injection dans les nouveaux formats de données.
  - Manque de protection des *Web Services*.
- Exploitation facilitée et impact plus important avec le Web 2.0 :
  - *Cross-Site Scripting* (XSS).
  - *Cross-Site Request Forgery* (CSRF) (cf. présentation SSTIC 2007).

Vulnérabilités causées par la volonté d'ergonomie du Web 2.0

- Déplacement de traitements et données sensibles vers le client :
  - Récupération des habilitations de l'utilisateur sur un serveur Web et envoi aux autres serveurs applicatifs par un script côté client.



- Suppression de certains contrôles de sécurité « pour ne pas diminuer la réactivité de l'interface » :
  - Choix de ne pas authentifier ni journaliser les requêtes AJAX.

```
if(code.startsWith("ajax"))
    retour = "OK";
else
    retour = controleRequete(requete);
```

Vulnérabilités dues à une mauvaise utilisation des formats et protocoles

- Possibilités d'injection dans les nouveaux formats de données :
  - *XML Poisoning*.

```
<identification>
  <login>Reno</login>
  <location>Paris</location>
  <group>Users</group>
</identification>
```

```
location = Paris</location>
<group>Admins</group>
<location>Paris
```

```
<identification>
  <login>Reno</login>
  <location>Paris</location>
  <group>Admins</group>
  <location>Paris</location>
  <group>Users</group>
</identification>
```

- Manque de protection des Web Services :
  - Possibilité d'énumération WSDL.
  - Absence d'authentification des requêtes.
  - Injections dans les paramètres (XPATH, SQL, LDAP, shell...).
  - Impact souvent critique : contournement d'une partie de la logique applicative permettant de réaliser des actions non autorisées voire « impossibles » avec l'interface Web standard.

Vulnérabilité « classique », mais dont l'exploitation et l'impact augmentent

- XSS 2.0 : exploitation facilitée.
  - Injection à l'intérieur de balises <script> déjà ouvertes, ce qui permet de contourner certains filtrages.

```
<script>
[..]
Couleur[<%=i%>] = "<%=prod.getCouleur()%>";
Nombre[<%=i%>] = "<%=prod.getNombre()%>";
[..]
</script>
```

couleur=noir";alert(document.cookie);//

```
<script>
[..]
Couleur[2] = "noir";alert(document.cookie);//";
Nombre[2] = "666";
[..]
</script>
```

  - Autorisation du *cross-frame scripting* (par modification de la propriété *document.domain*) sur un domaine trop large.
  - Flux hostile provenant d'un serveur tiers : « C'est toujours l'autre qui s'occupe de nettoyer les données ».
- XSS 2.0 : impact plus important.
  - Le Javascript peut permettre à l'attaquant d'effectuer des actions dans l'application avec les droits de l'utilisateur.

Partie 3 :
Le rôle des outils de développement dans la sécurité du Web 2.0

Présentation de quelques outils de développement

- GWT (Google Web Toolkit) : 
  - Génération de contenu HTML / Javascript à partir de classes Java.
- Echo2 (NextApp) : 
  - Paradigme similaire aux applications client lourd : envoi d'événements par le client et traitements applicatifs sur le serveur.
- Apollo / Flex (Adobe) :  
  - Génération de pages AJAX ou Flash.
- ASP.NET AJAX (Microsoft). 
- Backbase. 
- OpenLazlo, DWR, script.aculo.us, Dojo, Prototype, Ruby On Rails, Visual WebGUI, YUI,...

Démonstration des risques liés à l'utilisation d'outils de développement

- Démonstration d'attaque d'une application Web développée avec GWT :
  - Utilisation de l'outil FireBug : débuggeur pour Firefox.
  - Méthodologie similaire à celle utilisée pour auditer les applications clients lourds (paradoxe pour un navigateur !) : observation des communications réseaux et des traitements effectués (mode *debug*).
- Vulnérabilités :
  - Envoi d'informations non autorisées : le serveur envoie toutes les données de la classe (et un *Javascript* côté client affiche uniquement celles correspondant au profil de l'utilisateur).
  - Déport de traitements sensibles côté client (algorithme de chiffrement).
- Difficultés liées à la gestion des habilitations :
  - Si l'on veut que ce soit ergonomique, il faut le faire côté client.
  - Si l'on veut que ce soit sécurisé, il faut le faire côté serveur.
  - ... moralité : gestion des habilitations à prendre en compte côté client et côté serveur.

Partie 4 :
Les solutions concrètes pour améliorer la sécurité dans les applications... et leurs limites

Respecter les bonnes pratiques

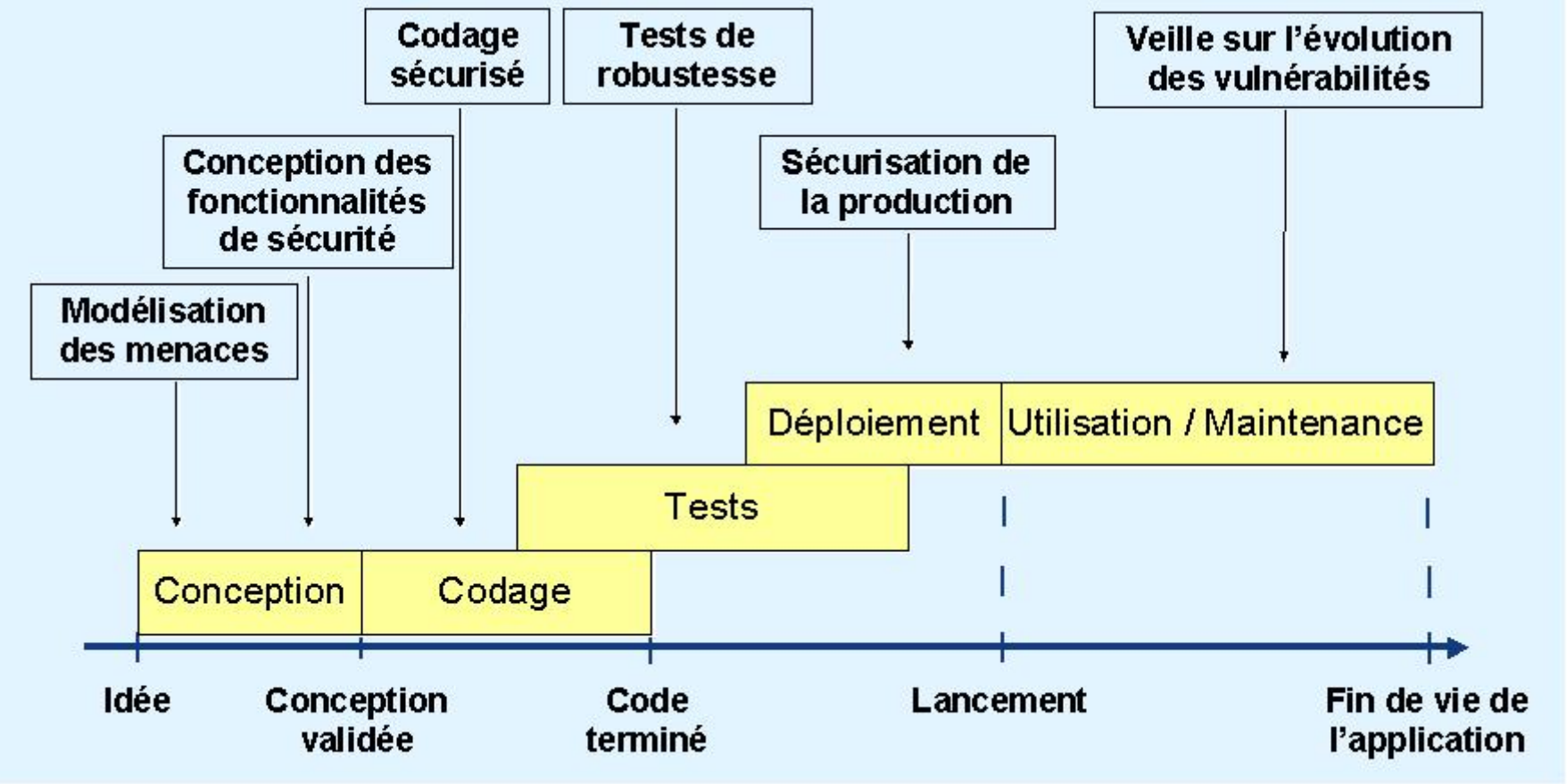
- « Il n'y a pas de sécurité côté client » :
  - Ne jamais déplacer les traitements sensibles sur navigateur.
  - Ne jamais envoyer des données sensibles sur le navigateur sans les chiffrer et / ou les signer sur le serveur.
- Choisir un *framework* de développement éprouvé et le maîtriser.
- Protéger tous les points d'entrée de l'application (requêtes AJAX, *Web Services*,...).
- Renforcer le filtrage contre le XSS et les injections dans les nouveaux formats de données (liste blanche et transformation en entités HTML).
- Filtrer les données provenant de serveurs tiers, même si ces serveurs sont de confiance : « La confiance n'exclut pas le contrôle ».
- ... mais suivre des recettes toutes faites sans les comprendre n'offre qu'une protection limitée...

Formation des concepteurs et développeurs

- Une formation doit à la fois présenter :
  - Les techniques d'attaques : « penser comme un attaquant ».
  - Les bonnes pratiques pour construire une application sécurisée : « penser comme un architecte ».
- La formation est indispensable :
  - « Vous ne savez pas ce que vous ne savez pas. »
  - « Une fonctionnalité de sécurité n'est pas forcement une fonctionnalité sécurisée. »
  - « Une poignée de personnes compétentes est plus efficace qu'une armée d'outils. »
- Mais pas toujours suffisante :
  - Toujours des erreurs dans le code source produit après un telle formation.
  - Pour obtenir des résultats pérennes, la formation doit s'accompagner d'une volonté de sécurisation dans les projets de développement.

Mise en place d'un processus de développement sécurisé

- SDLC (« Secure Development Life Cycle ») : processus permettant d'assurer un développement et une implémentation sécurisée.



Utilisation d'outils d'analyse automatique de code source

- Apports :
  - Pour le développeur : détection des portions de code potentiellement vulnérables au cours du développement.
  - Pour l'auditeur : repérage de zones de risques qui devront être creusées par analyse manuelle.
- Limites :
  - Ne remplacent pas une analyse « manuelle ».
  - Résultat d'un duel « homme contre outil » sur 460 lignes de code d'un filtre J2EE utilisée dans une application réelle.

Nb vuln	Analyse manuelle	Fortify v3.1.1
Critiques	3	1
Moyennes	2	0
Faibles	4	0
Faux positifs	-	3

Réalisation d'audits de code source avant la mise en production

- Apports :
  - Permet de détecter les principales vulnérabilités avant la mise en production.
  - Rôle de « gendarme » pour les développeurs, ce qui peut favoriser la qualité du code produit :-).
- Limites :
  - Difficulté pour un intervenant externe de s'approprier rapidement le fonctionnement d'une application complexe.
  - En cas de découverte de vulnérabilités dans l'architecture globale, la décision de retarder la mise en production est souvent impossible.
- Réflexion sur les meilleures pratiques pour l'audit de code source sur les projets importants :
  - Réalisation d'ateliers de travail et de relecture avec les développeurs.
  - Découpage du périmètre d'audit en modules indépendants pour faciliter un travail en parallèle.
  - S'assurer que les scénarios de risques sont pertinents.

Mise en place de mécanismes de sécurité externes à l'application

- Les « jokers » : relais inverses HTTP, pare-feux XML, etc.
  - Classique : « Nous sommes protégés par le *reverse proxy*. »
  - En pratique, la définition de règles de filtrage efficaces est soit impossible, soit supposerait d'avoir déjà réalisé un audit du code source pour connaître les paramètres dangereux...
  - Valable pour assurer une rupture protocolaire, ou dans une optique de défense en profondeur.
  - Parfois la seule solution pour protéger une application qu'on ne peut pas modifier.
- Utilisation de connexions de type *Citrix / Terminal Server* pour accéder à un navigateur permettant d'accéder à l'application...
  - Et le client léger dans tout ça ?
  - Et les risques d'actions non autorisées par des utilisateurs internes ayant accès au navigateur ?

Conclusion

- AJAX : la façon la « moins pire » d'améliorer l'ergonomie et la présentation du Web.
  - Mieux que les contrôles *ActiveX* ou les *applets Java* se connectant directement aux bases de données :-).
- Mais les risques sont importants :
  - Types de vulnérabilités directement liées à la conception des standards du Web (par exemple le CSRF) ou au modèle de développement du Web 2.0 (déport d'une partie de l'application côté client).
  - Attaques connues et faciles à exploiter.
  - Avec de nombreux *frameworks*, la sécurité reste la responsabilité des développeurs.

Remerciements :

Jérôme Boehm
L'équipe HSC

Merci pour votre attention !

Des questions ?

Renaud Feil
prenom . nom @ hsc . fr