Présentation TLS

TLS

SSL -> TLS

-> Cryptage de données

-> Server | TLS | Réseau

- Suppression de fonctionnalité dangereuse (SHA-1 / MD5 / …)

- Amélioration de la sécurité

- Ajout de protocole de cryptage.

Attaques TLS

POODLE

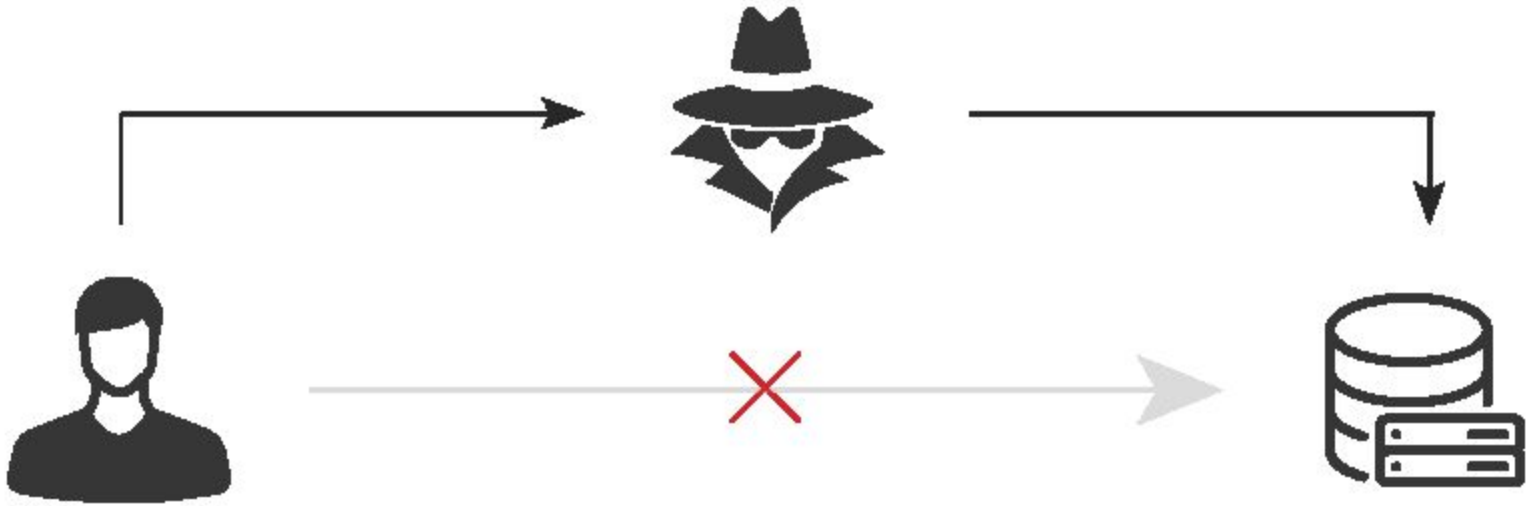
This POODLE bites: exploiting the SSL 3.0 fallback

CVE-2014-3566

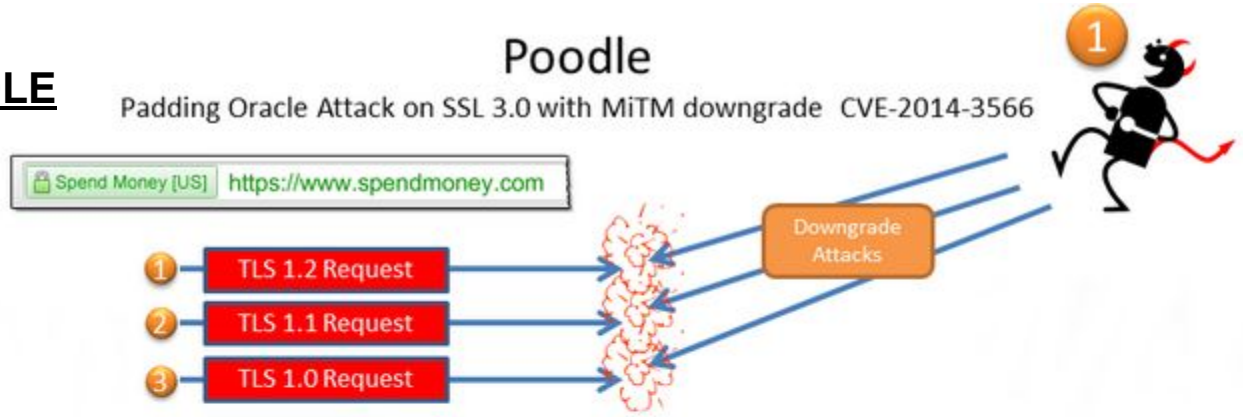
POODLE

- Padding Oracle On Downgraded Legacy
- 14 Octobre 2014
- Adam Langley, Bodo Möller, Thai Duong et Krzysztof Kotowicz
- Google Security Team
- MITM
- SSL3 (Secure Socket Layer v3.0)
- TLS

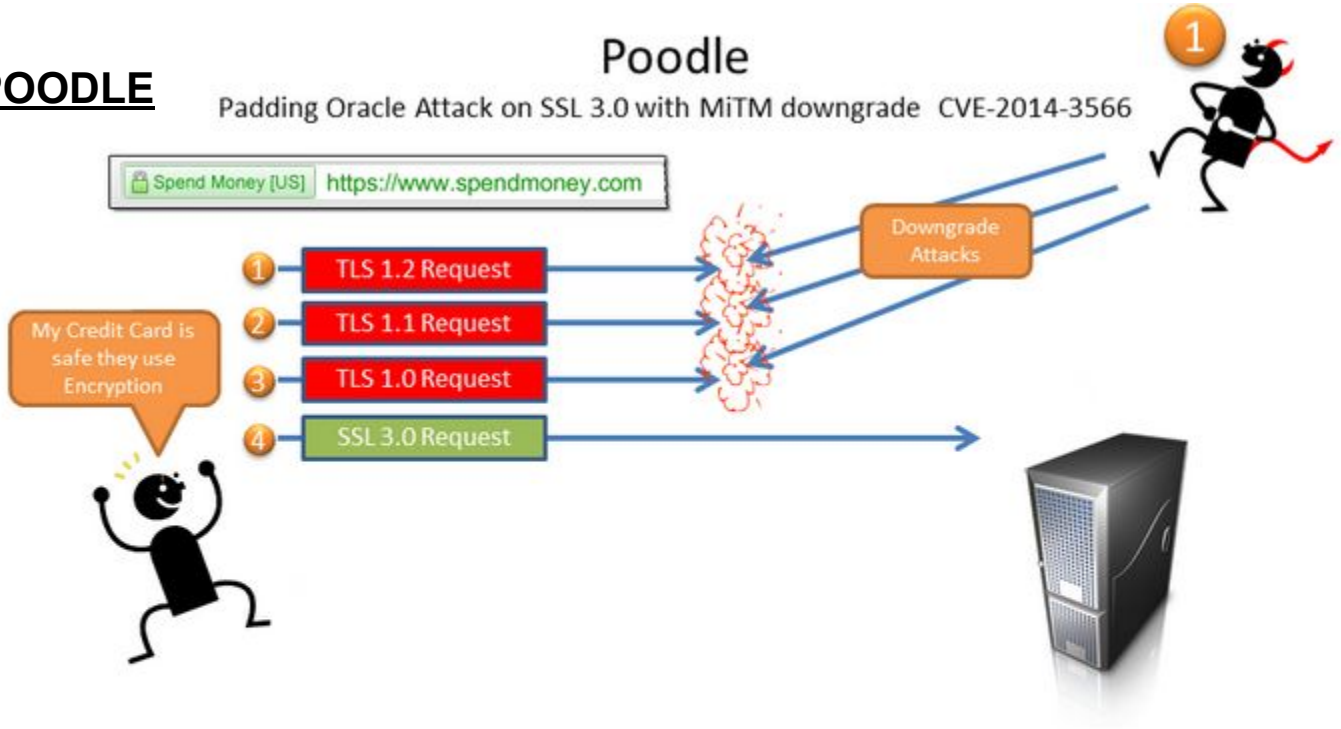
Attaque POODLE



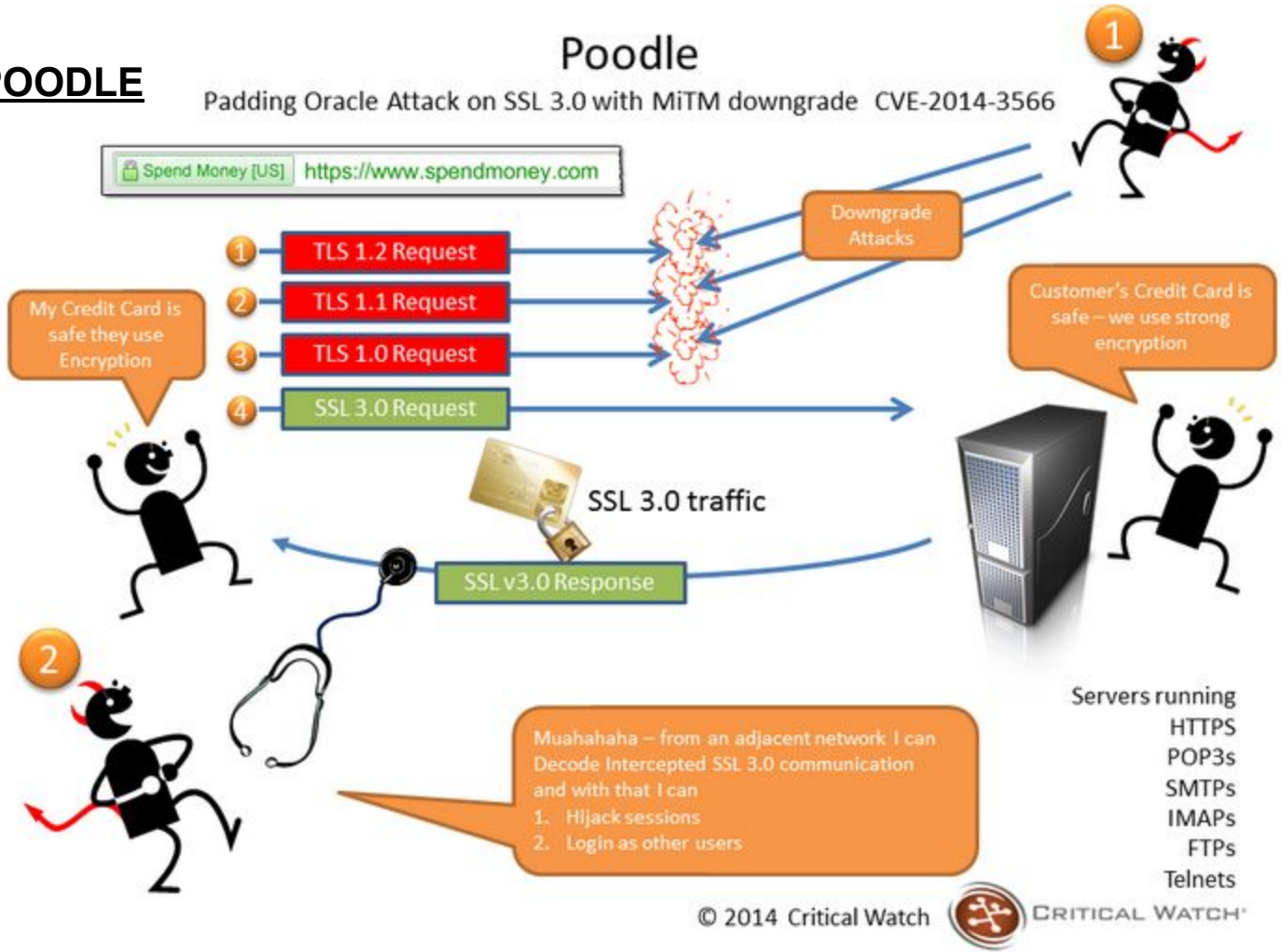
Attaque POODLE



Attaque POODLE

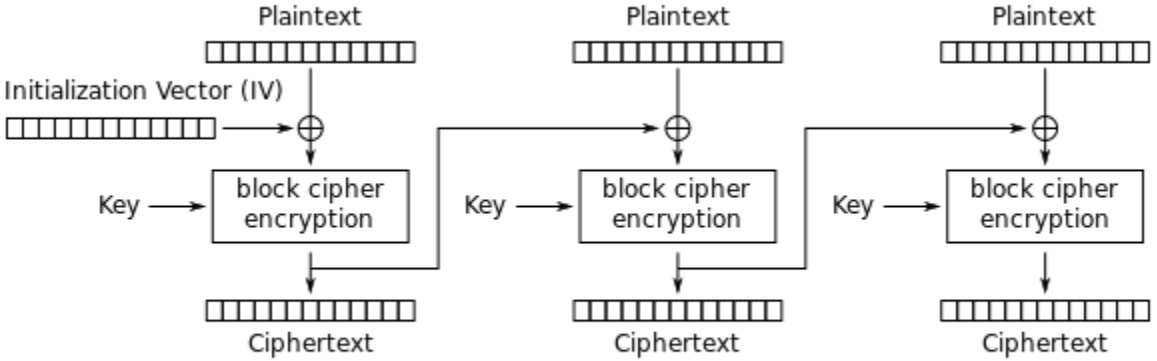


Attaque POODLE



Petit rappel sur le chiffrement

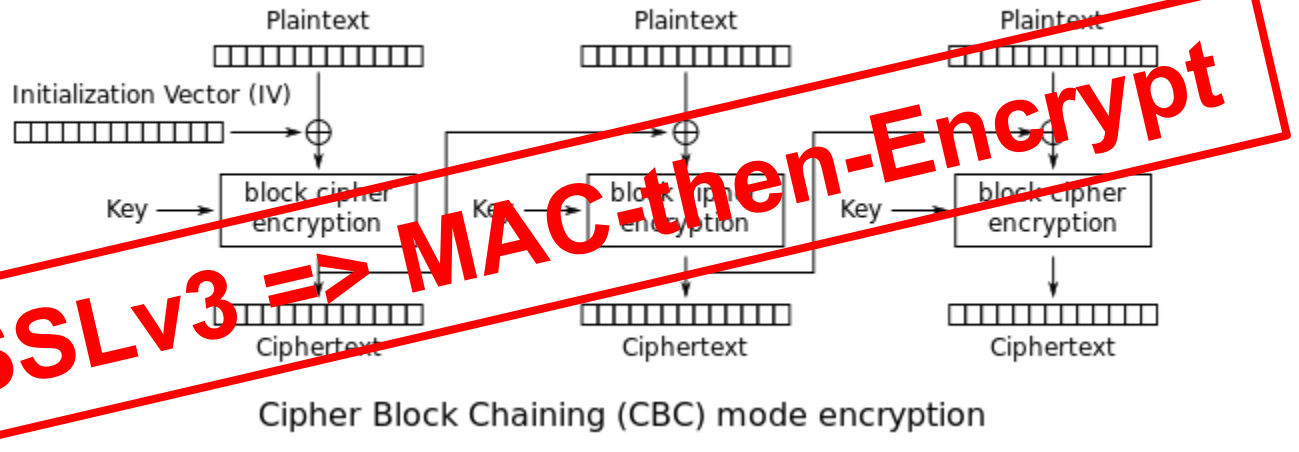
- Suites de chiffrements pour crypter la communication
- Vulnérabilité POODLE exploite
  - chiffrement symétrique avec des chiffrements par blocs
  - longueurs de blocs fixe
  - valeur de chaque bloc dépend du bloc précédent



Cipher Block Chaining (CBC) mode encryption

Petit rappel sur le chiffrement

- Suites de chiffrements pour crypter la communication
- Vulnérabilité POODLE exploite
  - chiffrement symétrique avec des chiffrements par blocs
  - longueurs de blocs fixe
  - valeur de chaque bloc dépend du bloc précédent



SSLv3 ⇒ MAC-then-Encrypt

Le Rembourrage(Padding)

- Données soient des multiples de la taille du bloc
  - Taille du bloc = 8 => les données doivent avoir 64, 80, etc octets
  - Pas un multiple => rembourrage (padding)
- Dernier octet du bloc = longueur du remplissage
  - pour 4 octet de rembourrage => bloc: `xx-xx-xx-yy-yy-yy-yy-04` où `yy` représente le rembourrage
  - valeur de rembourrage = taille de rembourrage => bloc: `xx-xx-xx-04-04-04-04-04`
  - longueur données = multiple de taille de bloc => bloc supplémentaire de rembourrage => `07-07-07-07-07-07-07-07`

Le Rembourrage(Padding)

SSL ne vérifie pas les octets de rembourrage

(ici tant que le dernier octet est compris entre 00 et 07 => OK)

xx-xx-xx-12-34-56-78-04

PO pour Padding Oracle

- L’attaquant sait ou peut deviner ce que contiennent les données
- L’attaquant modifie les données est les renvois au serveur
- Réponse du serveur : soit Mac erroné soit rembourrage incorrect

A : Attaquant S: Serveur
N: Navigateur de la victime

L’attaque POODLE

- But : voler un cookie de session

1. A incite N à lancer un script JavaScript (qui permet d’exe l’attaque)
2. Code JS incite N à envoyer plusieurs requêtes à S (cookies session contenu dans ces requêtes)
3. Code JS modifie l’URL de connexion
    - ajout de caractères => longueur des données soit un multiple de la taille du bloc
    - comme dit plus tôt le dernier bloc contiendra uniquement du rembourrage
4. A sait quels bloc de données contiennent le cookie de session (ici le $3^{ème}$ et le $4^{ème}$)

A : Attaquant S: Serveur
N: Navigateur de la victime

L’attaque POODLE

5. A copie l’intégralité du 3ème et l’envoie à S plusieurs fois en modifiant quelque chose dans l’URL pour que le MAC soit différent

6. Message accepté (max 256 fois)

7. A connaît maintenant le dernier octet et peut le combiner avec les locs précédents avec XOR => obtenir le dernier octet réel du 3ème bloc

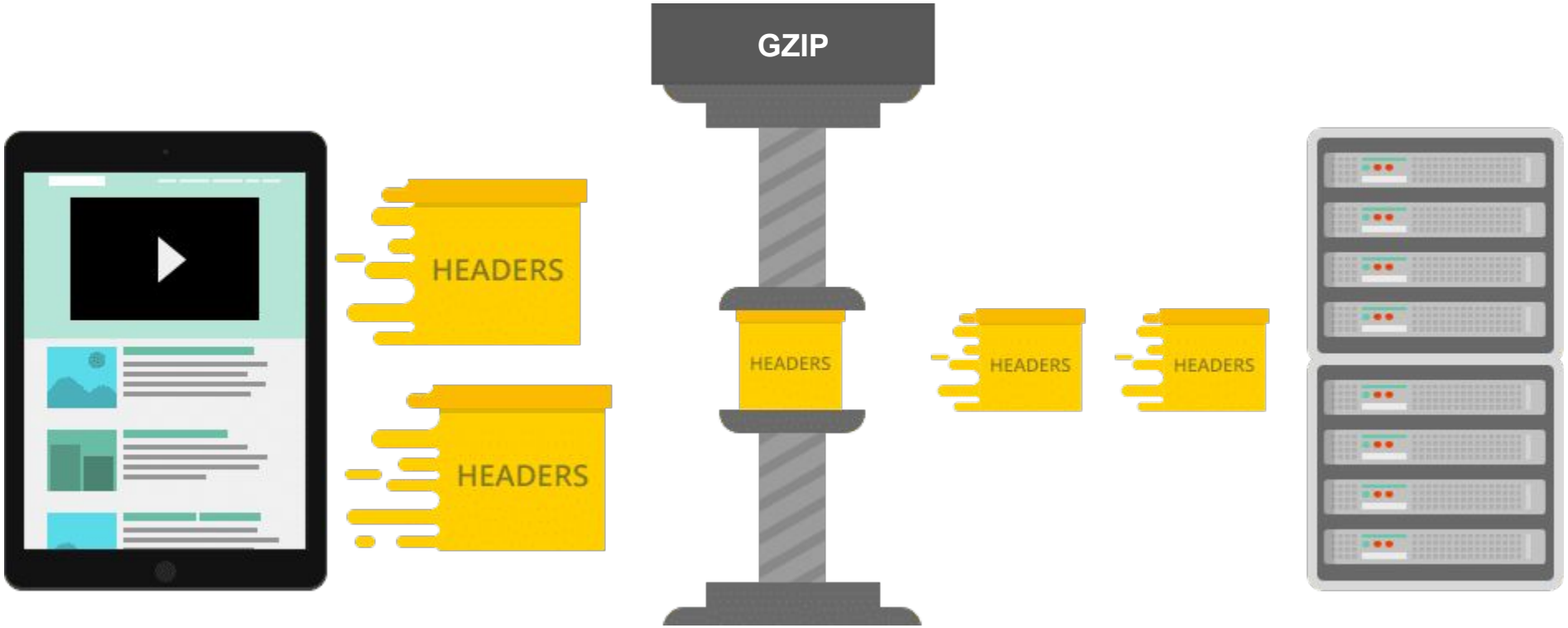
8. A peut allonger l’URL d’1 octet et répéter les opérations précédentes

9. Si longueur du cookie = 16, A connaitra le cookie après max 4096 requêtes (~ quelques min)

CRIME / BREACH

CVE 2012-4929 / CVE 2013-3587

Compression HTTP



Crime / Breach

Pourquoi ?

Récupérer des données sensibles, tel que des jetons de sessions, des mots de passes stockés dans les requêtes HTTP...

Comment ?

En se servant de la compression des pages web et en comparant leur taille on peut deviner le contenu de ces variables.

BREACH et CRIME marchent de la même manière sauf qu'avec CRIME c'est la compression TLS contre la compression HTTP, qui est beaucoup plus répandue, pour BREACH.

Explication fonctionnement :

La compression permet de gagner de la place sur les redondances dans la requête, en minimisant les messages répétés.

Ainsi en ajoutant à la requête un nouveau paramètre on peut voir si la taille de la requête augmente beaucoup (cas de non compression donc pas de redondances) ou alors la taille augmente peu (cas de compression donc redondance).

En faisant de nombreuses requêtes avec des valeurs différentes pour ce paramètre on peut retrouver le contenu de la variable contenue dans la requête, et ce malgré le chiffrement.

Exemple

Requête de base que le pirate cherche à attaquer

POST / HTTP /1.1
Host: testing.com
User -Agent: Mozilla /5.0 (Windows NT 6.1; WOW64; rv :14.0) Gecko /20100101 Firefox /14.0.1
Cookie: secretcookie =5db98j64wa23pq1cb4cb0031ln481nx1

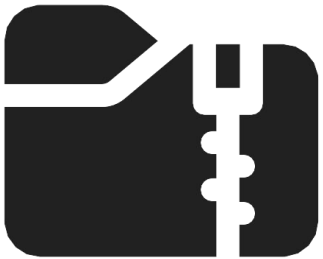
Requête que le pirate envoie pour tester le contenu de secretcookie

POST /secretcookie =0 HTTP /1.1
Host: testing.com
User -Agent: Mozilla /5.0 (Windows NT 6.1; WOW64; rv :14.0) Gecko /20100101 Firefox /14.0.1
Cookie: secretcookie =5db98j64wa23pq1cb4cb0031ln481nx1

L’attaquant va multiplier les requêtes comme celles de droites avec des valeurs différentes pour secretcookie.
A ‘secretcookie=5’ la compression sera plus performante et l’attaquant saura qu’il s’agit du premier caractère.
En faisant cela nombreuses fois il pourra brut-force la valeur du cookie.

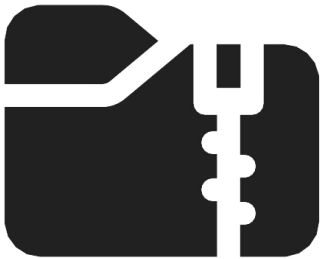
Explication fonctionnement :

secretcookie = 0



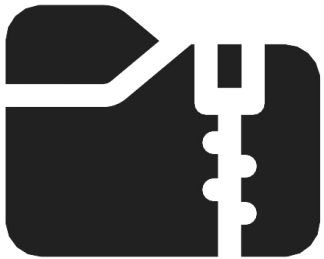
500 bits

secretcookie = a



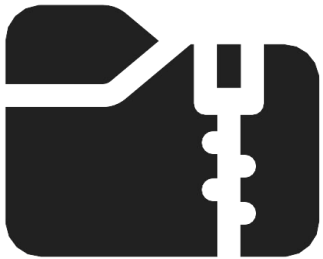
500 bits

secretcookie = 5



498 bits

secretcookie = 9



500 bits

Solutions

- Désactiver la compression HTTP
- Séparez les secrets de l'entrée de l'utilisateur
- Randomiser les secrets par requête
- Masquer les secrets (effectivement randomiser par XORing avec un secret aléatoire par requête)
- Protégez les pages contre le CSRF
- Masquer la longueur (en ajoutant des nombres aléatoires d'octets aux réponses)
- Limiter le taux de demandes pour éviter le brut-force

MERCI