Université de Bordeaux
Faculté de Sciences et Techniques
Professeur referent : Emmanuel Fleury

Études et implémentations d'attaques contre SSL/TLS

Grahek Kévin, Puygrenier Martial, Tremblain Rémi

Bordeaux, 27 avril 2015

Sommaire

Introduction

Le protocole TLS (Transport Layer Security) et son prédécesseur SSL (Secure Sockets Layer), sont des protocoles de sécurisation des échanges sur Internet. Le protocole SSL était développé à l'origine par Netscape et l'IETF [1] en a poursuivi le développement en le rebaptisant Transport Layer Security (TLS). On parle parfois de SSL/TLS pour désigner indifféremment SSL ou TLS. Il y a actuellement eu quatre versions différentes du protocole qui ont été utilisées, SSLv2 en 1995, SSLv3 1996, TLSv1.0 en 1999, TLSv1.1 en 2006 et TLSv1.2 en 2008. Chaque changement de version est dû à une mise a jour du protocole pour apporter ou combler des failles de sécurités. Le protocole permet de satisfaire aux objectifs de sécurité suivants :

1. L'authentification du serveur
2. la confidentialité des données échangées (ou session chiffrée),
3. l'intégrité des données échangées,
4. (optionnellement) l'authentification du client (souvent géré par le serveur).

L'utilisation la plus courante du protocole SSL/TLS est la sécurisation des échanges entre un client et un serveur distant, empêchant pour entité de lire les données qui transitent entre les deux parties. Dans ce document nous allons étudier trois attaques contre SSL/TLS visant à montrer les faiblesses du protocole.
Les attaques étudiées dans ce document sont :

— Padding Oracle On Downgraded Legacy Encryption (POODLE attack) - septembre 2014
— Heartbleed attack - mars 2014
— Browser Exploit Against SSL/TLS (BEAST) attack - septembre 2011

Pour chacune des attaques, nous avons défini les hypothèses de départ, le fonctionnement, la faisabilité ainsi que les contre-mesures mises en place. Pour compléter nos recherches, une implémentation de chaque attaque a été réalisée en Python.

1. Internet Engineering Task Force : groupe d'individu participant à l'élaboration de standards Internet.

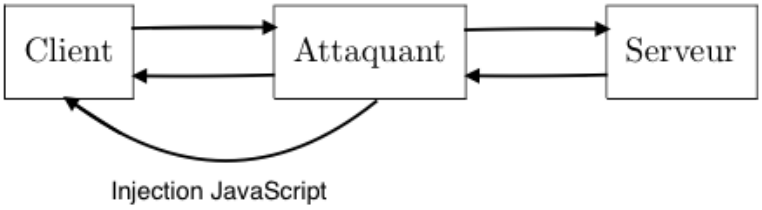
Chapitre 1

POODLE

1.1 Introduction

L'attaque POODLE (Padding Oracle On Downgraded Legacy) découverte le 14 septembre 2014 par l'équipe de securité google[3] est basée sur l'exploitation d'une vulnérabilité du protocole SSLv3 via l'utilisation d'un "Man in the middle" entre le serveur web et le navigateur web de la victime. L'attaque permet de déchiffrer des informations telles que des cookies de session ou encore les tokens HTTP Authorization header contents, cela constitue donc une atteinte à la confidentialité des données. Les systèmes touchés sont ceux qui disposent d'une implémentation du protocole SSLv3 ou ceux qui utilisent les navigateurs employant SSLv3.

Dans notre attaque, nous utilisons la configuration "Man in the middle" suivante :



Le but de l'attaquant va être de déchiffrer les requêtes partant du client vers le serveur.
L'ensemble du code réalisé ainsi que son explication est disponible à l'adresse suivante :
`https://github.com/mpgn/poodle-exploit`

1.2 Hypothèses de départ

Dans le cadre de cette attaque, nous devons mettre en place plusieurs hypothèses afin de rendre cette exploitation de faille réalisable :

- Le client et le serveur sont capables d'utiliser le protocole SSLv3.
- Le mode d'opération pour traiter les données est CBC (Cypher Bloc Chaining)
- L'attaquant se place en "Man in the Middle". Dans notre cas, l'attaquant contrôle un proxy qui intercepte les données et les transmet à l'attaquant.
- L'attaquant peut faire en sorte que le client génère des requêtes de son choix vers le serveur.
- L'attaquant peut intercepter, lire et modifier les requêtes du client et du serveur sans qu'aucune des deux parties ne s'en rendent compte.

— L'attaquant se sert du serveur comme d'un oracle [1].
— Le serveur doit être en mesure d'accepter un grand nombre de requêtes venant du client.

1.3 Déroulement de l'attaque

Nous allons voir dans cette partie quelles sont les grandes étapes de la réalisation de cette attaque.

1.3.1 Prémices de l'attaque

L'exploitation de la vulnérabilité pour réaliser l'attaque POODLE est basée sur la négociation protocolaire entre le client et le serveur. En effet, à cette étape, le client et le serveur se mettent d'accord sur le protocole de sécurisation des échanges à utiliser par les deux parties. C'est à ce moment là que l'attaquant intervient en modifiant la requête envoyée par le client en spécifiant que le client ne supporte pas plus que le protocole SSLv3.[1]

Une des spécificités du protocole SSLv3 est sa mécanique de chiffrement et d'authentification des données de type *Mac-then-Encrypt*. C'est à dire que les données vont tout d'abord être hachées via une fonction de hachage, telle que SHA-256, créant ainsi le MAC qui permet d'authentifier les données.
Les données et le MAC vont ensuite être chiffrées. Or ce mode ne permet pas de garantir l'intégrité des données chiffrées, laissant donc la possibilité à un attaquant de les modifier.

Un des points important de l'attaque provient du besoin de CBC de diviser les données en bloc de taille fixe et équivalent. Le protocole SSLv3 a donc mis en place un système de padding qui permet de compléter le dernier bloc CBC, si besoin est. La spécificité de ce système est que le dernier octet du dernier bloc correspond toujours à la taille du padding qui sera au maximum égale à la taille d'un bloc.
Une autre spécificité est la génération du padding fait aléatoirement et sans contrôle. Étant donné que le MAC n'opère que sur les données, l'attaquant a une fois de plus la possibilité de modifier le contenu du padding.

La requête type utilisée par le client est de la forme suivante :

GET / HT	TP/1.1\r\n	Cookie :	VRHnogWg	\r\n\r\nxxxx	xxxxM	AC •••3

On peut remarquer que la requête est découpée en bloc de taille constante (ici 8 octets) et que le dernier octet du dernier bloc correspond à la taille du padding (•). La longueur d'un bloc n'est pas fixée et dépend du chiffrement utilisé AES, DES, 3-DES, etc). Dans cette optique et afin d' amorcer l'attaque, la première étape sera donc de déterminer la taille d'un bloc.

Remarque : L'attaque fonctionne quelle que soit la fonction de chiffrement par bloc utilisé (DES, 3DES ou AES). Dans l'implémentation que nous avons faite de l'attaque, le chiffrement est AES

1. Entité qui retourne des informations sur une requête envoyée.

avec une clé de 128 bits, cela produit donc des blocs de taille 16 octets.

1.3.2 Détermination de la taille d'un bloc

Dans cette phase, l'attaquant va intercepter une requête émise par le client à destination du serveur et ainsi récupérer une première taille R_0. Il va ensuite redemander au client de regénérer **cette** requête en incluant un caractère supplémentaire dans le chemin de la requête, comme dans la trame suivante[2] :

GET /A HT	TP/1.1\r\n	Cookie :	VRHnogWg	\r\n\r\nxxxx	MAC data	●●●●●●●7

Il récupérera ainsi une nouvelle taille R_i de la requête. L'attaquant va réitérer cette opération jusqu'à ce que la taille R_i soit différente de la taille R_0, ainsi l'attaquant pourra effectuer une simple soustraction R_i - R_0 pour récupérer la taille d'un bloc et par la même occasion obtenir un dernier bloc complet de padding.

1.3.3 Altération et déchiffrement

Pour comprendre comment l'attaquant va réussir à déchiffrer un octet d'information, il faut regarder le fonctionnement du déchiffrement du mode d'opération CBC.

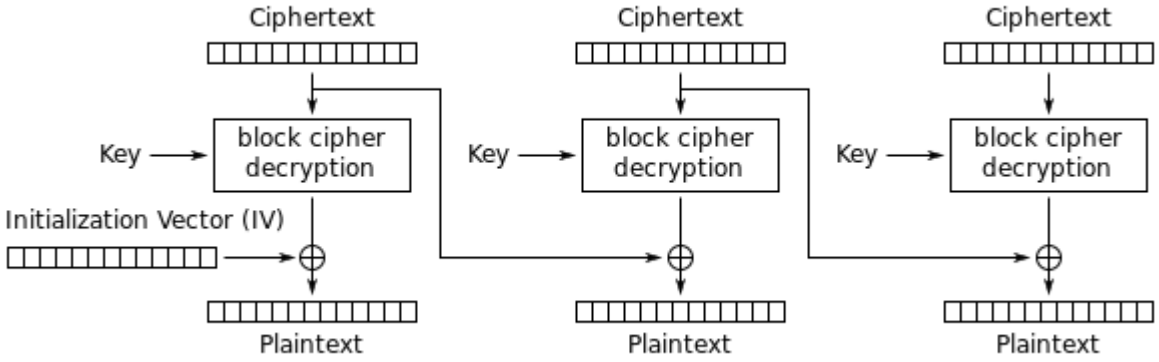


FIGURE 1.1 – Processus de déchiffrement CBC

Comme nous le montre l'image ci-dessus, lorsqu'un bloc d'information est déchiffré, il s'en suit une opération de type XOR qui est appliquée au bloc chiffré précédent. Cela nous permet d'obtenir un bloc en clair correspondant au bloc chiffré du départ.
Rappelons qu'il n'y a pas de contrôle du padding, dans cette perspective si un attaquant remplace le dernier bloc de padding par le bloc chiffré de son choix et envoie la requête modifiée au serveur, il aura une chance sur 256 que la requête soit acceptée par le serveur, il va donc attendre la réponse de ce dernier et analyser le résultat :

— Réponse négative : dans ce cas, le serveur renvoie une erreur de MAC car le dernier octet du dernier bloc ne correspond plus à sa valeur initiale, le padding a donc une taille

différente. L'attaquant va alors répéter cette manipulation de remplacement autant de fois qu'il sera nécessaire pour obtenir une réponse positive. Avant chaque requête, l'attaquant fait déconnecter le client du serveur puis le reconnecte. Cette opération est obligatoire pour que le chiffrement de la requête soit modifié.

— Réponse positive : Le serveur a réussi à déchiffrer la requête, cela implique que le dernier octet du bloc déchiffré appliqué à une opération XOR avec le dernier octet chiffré du bloc précédent est égale à la longueur du padding, soit la taille d'un bloc moins un. Prenons pour exemple des blocs de taille 8, et n représente le nombre de bloc total.

$$P_n[7] = D_k(C_n)[7] \oplus C_{n-1}[7] \tag{1.1}$$

L'attaquant sait aussi que le dernier octet du bloc i qu'il cherche à déchiffrer est égal à :

$$P_i[7] = D_k(C_i)[7] \oplus C_{i-1}[7] \tag{1.2}$$

Dans ces deux équations, l'attaquant ne connait pas : $P_i, D_k(C_i), D_k(C_n)$ mais il sait que $C_i = C_n \Rightarrow D_k(C_i) = D_k(C_n)$ et $P_n[7] = 7$ puisqu'il a lui-même altéré la requête du client et il connait la longueur d'un bloc. Il va donc poser l'équation suivante :

$$\begin{aligned} P_n[7] &= D_k(C_n)[7] \oplus C_{n-1}[7] \\ P_n[7] &= D_k(C_i)[7] \oplus C_{n-1}[7] \\ 7 &= D_k(C_i)[7] \oplus C_{n-1}[7] \\ D_k(C_i)[7] &= C_{n-1}[7] \oplus 7 \\ P_i[7] \oplus C_{i-1}[7] &= C_{n-1}[7] \oplus 7 \\ P_i[7] &= C_{i-1}[7] \oplus C_{n-1}[7] \oplus 7 \end{aligned} \tag{1.3}$$

Il obtient une formule qu'il appliquera à chaque fois qu'il voudra déchiffrer un octet pour un bloc de longueur 8.

$$P_i[7] = C_{i-1}[7] \oplus C_{n-1}[7] \oplus 7$$

Généralisée par la formule suivante, où L représente la longueur d'un bloc :

$$P_i[L-1] = C_{i-1}[L-1] \oplus C_{n-1}[L-1] \oplus L - 1$$

Une fois que l'attaquant obtient un premier octet, il réitère l'altération de la requête pour obtenir le reste du bloc, c'est à dire en modifiant de façon analogue à la méthode de détermination de la taille du paquet (cf. 2.3.2) le chemin de la requête, en y ajoutant un caractère afin de décaler la localisation de l'octet qu'il désire déchiffrer. Il doit aussi penser à enlever un octet dans les données de la requête pour garder une taille constante.

Remarque : l'attaquant est obligé de décaler les octets car il ne connait qu'un octet en clair : le dernier octet du dernier bloc qui correspond à la longueur du padding.
Exemple :

GET /AA H	TTP/1.1\r	\nCookie	:VRHnogW	q\r\n\r\nxxx	MAC data	●●●●●●●7

L'attaquant peut ainsi déchiffrer chaque octet de chaque bloc, sauf pour le premier bloc puisqu'il dépend du vecteur initialisation qu'il ne possède pas.

1.4 Faisabilité

Pour réaliser cette attaque, l'attaquant doit pouvoir mettre en place et contrôler un proxy de type "Man in the Middle" [4]passif. Il peut être directement chez le client ou sur le réseau. Son rôle sera d'intercepter les requêtes entre le client et le serveur ainsi que de les transmettre à l'attaquant qui lui sera actif. L'attaquant doit aussi pouvoir faire en sorte que le client se connecte à une page web malicieuse au préalablement réalisée pour pouvoir injecter du javascript et faire générer des requêtes au client en direction du serveur sécurisé. En supposant que le client se soit déjà connecté au service proposé par le serveur, le navigateur va automatiquement ajouter le cookie à la requête. L'attaquant a donc seulement besoin de faire générer des requêtes vers le serveur. Le reste se déroule exactement comme décrit dans le section "Déroulement de l'attaque (1.3)".

1.5 Sécurisation

L'un des fondements de l'attaque repose sur la négociation protocolaire entre le client et le serveur visant à les obliger à utiliser le protocole SSLv3. Pour contrer cette attaque, la meilleure solution consiste à désactiver SSLv3 aussi bien du côté navigateur que du serveur. Les versions supérieures à ce protocole (TLS) utilisent un format où les octets du padding ne sont plus générés de manière aléatoire, il n'est donc plus possible de dupliquer un bloc de données dans le bloc de padding sans avoir d'erreur exploitable par l'attaquant.

Dans le cas ou SSL serait toujours actif, une mise à jour du protocole a été faite ajoutant une sécurité TLS (Fallback Signaling Cipher Suite Value) qui empêche une rétrogression du protocole utilisé entre le client et le serveur.

Note : Seul Internet Explorer 6 sous Windows XP utilise par défaut le protocole SSLv3 et reste vulnérable. L'utilisateur peut décider de passer à TLS mais doit activer l'option manuellement.

Chapitre 2

Heartbleed

2.1 Introduction

OpenSSL est une boite à outils open source maintenue à jour par une communauté mondiale dans le but de développer un outil robuste pour gérer des échanges chiffrés et sécurisés entre deux entités distantes. L'attaque Heartbleed découverte la société Codenomicon[5] et rendue publique le 7 avril 2014 ne concerne pas directement une vulnérabilité dans le protocole SSL/TLS mais dans l'implémentation qui en a été faite par OpenSSL. C'est l'une des attaques la plus importante de ces dernières années car OpenSSL est à ce jour utilisée par plus d'un demi-million de personnes dans le monde et la vulnérabilité est très facilement exploitable par un attaquant.

Le but de l'attaquant va être de récupérer des données sensibles contenues dans la mémoire du serveur par exemple (password, clé privée...). Contrairement à l'attaque Poodle vue précédemment l'attaquant n'aura pas besoin de mettre en place un proxy entre un client et le serveur, il agit lui même en tant que client et attaquant. En effet les données qu'il cherche à voler sont situées dans le mémoire du serveur, il ne cible pas une entité particulière.
Ici, nous prendrons l'exemple d'un schéma du type client/serveur, mais il faut savoir que les clients utilisant des logiciels implémentant OpenSSL sont aussi vulnérables) l'attaque.

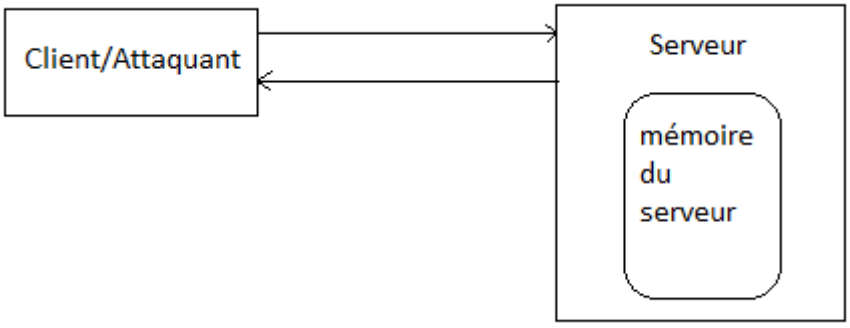


FIGURE 2.1 – L'attaquant va récupérer des données potentiellement sensibles contenues dans la mémoire du serveur.

2.2 Hypothèses de départ

Voici les deux hypothèses de départ nécessaires à une personne malveillante pour attaquer l'entité distante :

— L'entité visés utilise OpenSSL 1.01 à 1.01f
— L'entité malveillante dispose d'un outil pour exploiter la vulnérabilité

On constate que cette attaque est déjà beaucoup plus simple à mettre en place que Poodle.

2.3 Déroulement de l'attaque

2.3.1 Envoi d'une requête heartbeat

Pour permettre la transmission des données chiffrées entre les deux entités sans refaire une négociation protocolaire à chaque échange, OpenSSL a implémenté dans la partie SSL/TLS une extension nommée **heartbeat**. Cela va permettre de garder en vie une session d'échanges sécurisés entre le client et le serveur et ainsi gagner du temps dans l'échange des données. Heartbeat représente donc le "cœur" qui fait vivre la session, or le **heartbleat** implémenté dans OpenSSL possède une vulnérabilité dans son implémentation ce qui a donné le nom à l'attaque "Heartbleed".

Pour l'expliciter, nous allons nous baser sur le code qui a introduit le bug. Tout d'abord l'heartbeat possède la structure suivante :

```
1 struct {
2         HeartbeatMessageType type;
3         uint16 payload_length;
4         opaque payload[HeartbeatMessage.payload_length];
5         opaque padding[padding_length];
6 } HeartbeatMessage;
```

Le client va envoyer une *requête heartbeat* composée d'un payload qui contient une taille, une donnée et enfin d'un padding de maximum 16 octets. Lorsque le client envoie la requête, le serveur va traiter les informations comme il suit :

1. il va tout d'abord récupérer les données envoyées par le client
2. allouer la mémoire qui va contenir les données demandées par le client
3. copier les données depuis une zone mémoire source vers une zone mémoire destination
4. envoyer le tout au client

L'étape **1** correspond à ces quatre lignes :

```
1 /* Read type and payload length first */
2 hbtype = *p++;
3 n2s(p, payload);
4 pl = p;
```

Note : La macro `n2s` prend deux octets de `p` et les met dans la variable `payload`.

Comme on peut le voir sur la structure du heartbeat, la variable `payload` va finalement contenir la longueur rentrée par l'utilisateur du `payload_length` définie dans la requête heartbeat qu'il a fait

au serveur. La variable `p1` contiendra quand à elle les datas du payload. On peut déjà remarquer qu'il n'y a ici aucun contrôle sur la longueur rentrée par l'utilisateur par rapport à sa valeur réelle, il peut donc indiquer une valeur erronée sans qu'aucune erreur ne soit signalée.

On retrouve l'étape **2**, lecture des données, juste un peu plus loin dans le code :

```
1 buffer = OPENSSL_malloc(1 + 2 + payload + padding);
2 bf = buffer
```

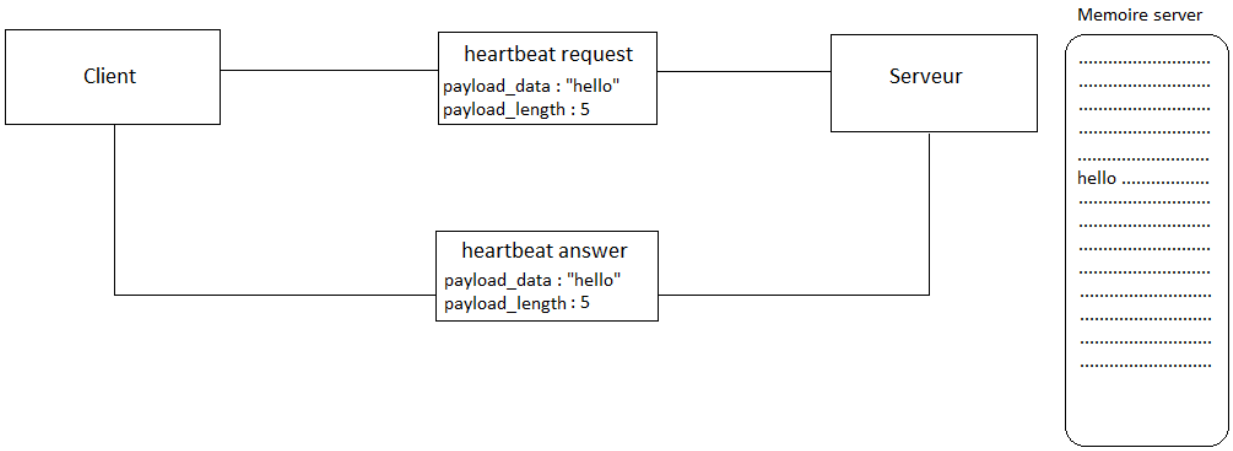
Ici on alloue un buffer qui contiendra les données retournées par le serveur. Encore une fois, comme le payload n'est pas contrôlé initialement, un utilisateur malveillant peut faire un malloc plus grand que la taille nécessaire.

L'étape **3**, lecture des données est introduite par :

```
1 *bp++ = TLS1_HB_RESPONSE;
2 s2n(payload, bp);
3 memcpy(bp, pl, payload);
4 RAND_pseudo_bytes(bp, padding);
5 r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer,
6                                          3 + payload + padding);
```

C'est ici la fonction `memcpy` qui va nous intéresser. Elle va copier les données du payload envoyée par le client dans le buffer avec la taille payload précisée par l'utilisateur.

Résumé en une image :

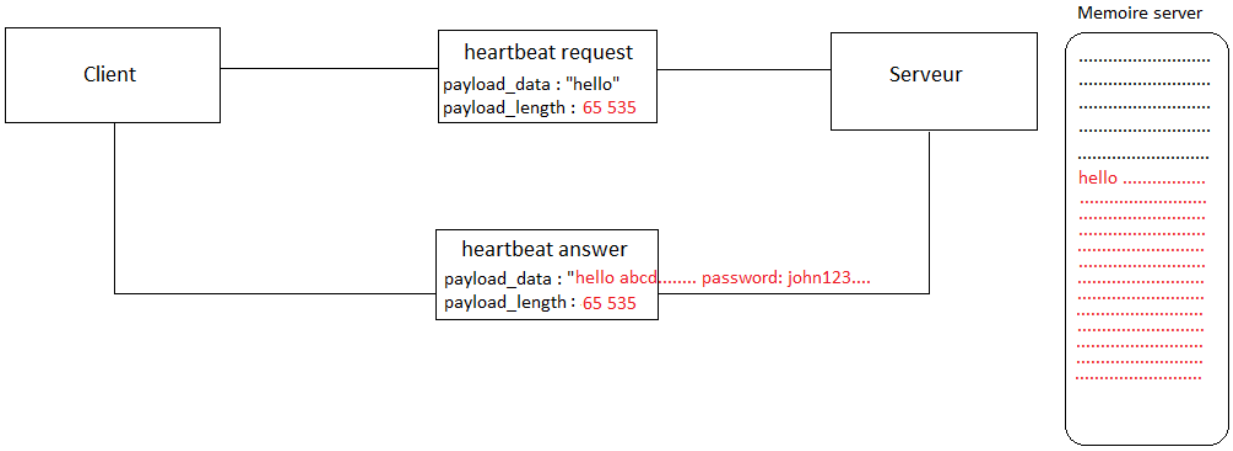


Remarque : Le git de OpenSSL nous indique que ce bug a été introduit le 31 décembre 2011 par Robin Seggelmann. Il y a donc eu une période de 2 ans durant laquelle cette attaque a pu être exploitée par diverses entités malveillantes.

2.3.2 Récupération et analyse des données

La vulnérabilité vient donc du fait qu'il n'y a aucun contrôle sur la longueur du `payload` entré par l'utilisateur. Si un attaquant rentre une longueur complètement erronée du payload alors le serveur va lire en mémoire plus qu'il ne devrait et retourner les informations à l'attaquant. Ainsi l'entité malveillante va faire une requête avec comme donnée du payload "hello" par exemple mais indiquer une taille de 65 535 octets au lieu de 5. Le serveur va donc allouer un buffer de taille 64k et grâce au `memcpy`, le serveur va copier 64k de données dans le buffer et les renvoyer à l'attaquant qui n'a plus qu'à analyser les données reçues et regarder s'il y des informations sensibles à l'intérieur.

Cela nous donne le schéma d'attaque suivant :



Note : la longueur est limitée à 65 535 octets qui correspond à la longueur maximale d'une requête heartbeat. Dans les données reçues, il faut enlever les données du padding (16 octets) + l'entête de la requête (3 octets).

2.4 Faisabilité

Pour mettre en place l'attaque Heartbleed, il suffit à une entité malveillante d'écrire un script python qui se connecte au serveur ciblé, de faire une demande d'heartbeat avec un payload connu en précisant une taille très grande et de regarder la réponse qui en a été faite. Une implémentation de l'attaque a été réalisée à l'adresse : `https://github.com/mpgn/heartbleed-exploit`

Dans notre exemple, l'attaque se fait sur le serveur *https ://www.cloudflarechallenge.com/* spécialement conçu pour démontrer la validité de l'attaque. Les données récupérées de la mémoire du serveur sont stockées dans un fichier out.txt. Nous avons envoyé une requête heartbeat de taille 16 ko alors que la taille réelle est de 4 otects et le serveur va bien nous renvoyer 16ko de données contenues dans sa mémoire au lieu de 4. On peut noter qu'ici nous avons fixé une taille de 16 384 octets à notre payload. On pourrait bien sûr monter à la valeur maximale 65 535 (64ko), le serveur nous retournera alors quatre paquets de 16 384 [1]. On peut ainsi récupérer des informations

1. "RFC6520 - The total length of a HeartbeatMessage MUST NOT exceed 2^{14}"

sensibles concernant des données utilisateur (passwords...) mais aussi la clé privée du serveur qui sert à chiffrer les données. Une fois en possession de la clé, l'attaquant peut déchiffrer toutes les données. Pour trouver une clé privée dans son intégralité, il faut en moyenne une dizaine d'heures d'après les attaques réalisées par la communauté relevant le challenge cloudflare.

Il faut aussi noter que le heartbleed peut même être fait pendant la phase de handshake quel que soit le résultat de celui-ci, cela implique que l'attaquant peut récupérer des informations sensibles sans même établir un connexion sécurisée avec le serveur. L'attaque Heartbleed a aussi la particularité de ne laisser aucune trace. En effet, comme il s'agit d'une erreur d'implémentation, le serveur répond tout à fait normalement et ne suspecte rien de malveillant.
Depuis la découverte de la faille, OpenSSL a rapidement été corrigé comme nous allons le voir dans la partie suivante.

2.5 Sécurisation

Le patch correctif et l'annonce de la faille sont apparus un mois après la découverte de la faille, OpenSSL étant hébergé sur Github, nous pouvons voir le code correctif qui à été appliqué [2] :

```
1 if (1 + 2 + 16 > s->s3->rrec.length)
2     return 0;
3 hbtype = *p++;
4 n2s(p, payload);
5 if (1 + 2 + payload + 16 > s->s3->rrec.length)
6     return 0;
7 pl = p;
```

Le patch se déroule en deux étapes et a été corrigé dans deux fichiers **sslt1_lib.c** et **ssld1_both.c**.

1. La taille de la requête n'est pas inférieure à la taille minimum. (ligne 1)
2. La taille de la requête n'est pas supérieure à la taille de la requête une fois que l'on a récupéré la longueur du payload. (ligne 5)

2. https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3#diff-2

Chapitre 3

BEAST

3.1 Introduction

BEAST (browser exploit against SSL/TLS) est une attaque développée et expliquée par Thai Duong et Juliano Rizzo dans un article intitulé "Here Come The Ninja"[6] sortit 13 mai 2011.
Il s'agit d'une attaque à clair choisi qui nécessite d'être en "Man On The Middle". Elle permet d'obtenir des informations sensibles présentes dans les requêtes https durant un dialogue entre le client et le serveur (cookie, session, token...). Dans cette attaque, le serveur n'a aucune implication, seul le client et le proxy (mitm) seront utiles.

3.2 Hypothèses de départ

- Le client et le serveur utilise le protocole SSLv3 ou TLS1.0
- Le mode d'opération pour traiter les données est CBC (Cypher Bloc Chaining)
- L'attaquant se place en "Man in the Middle"
- L'attaquant peut faire en sorte que le client génère des requêtes de son choix vers le serveur.(Attaque à clair choisi)
- L'attaquant peut intercepter, lire et modifier les requêtes du client et du serveur sans qu'aucune des deux parties ne s'en rendent compte.

3.3 Déroulement de l'attaque

3.3.1 Prémices de l'attaque

Pour réaliser cette attaque, il y a deux points important :

- L'utilisation du mécanisme de chiffrement par bloc CBC à la particularité de débuter son chiffrement par un vecteur d'initialisation et de découper les données en bloc de tailles constantes.
  On constate que le premier bloc chiffré et égale à : $C1 = E_k(IV \oplus P1)$

- Lors d'un échange sécurisé entre le client et le serveur de nombreuse requêtes sont envoyées, or générer un nouveau vecteur d'initialisation aléatoirement à chaque requêtes requière du temps. Pour palier à ce problème l'implémentation de TLS1.0 et versions antérieures a été fait de telle sorte que le vecteur d'initialisation de la seconde requête équivaut au dernier bloc chiffré de la précédente requête chiffrée. Ainsi un seul vecteur d'initialisation est tiré au hasard et les autres sont tous déterminés par rapport à la requête précédente.

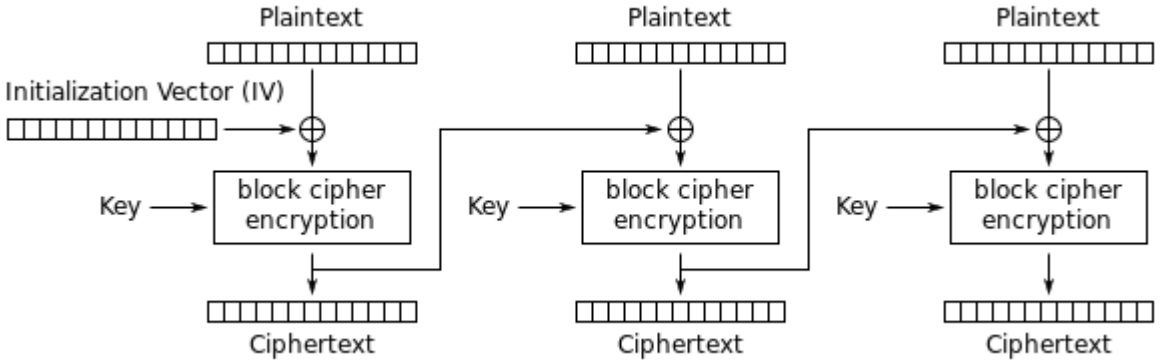


FIGURE 3.1 – Processus de chiffrement CBC

Requête 1 : IV : généré aléatoirement et on obtient des blocs chiffrés :$C1, C2, C3...Cn$
Requête 2 : IV : Cn et on obtient les blocs chiffrés suivant : $C'1, C'2, C'3..C'n$

3.3.2 Principe de l'attaque

L'attaquant va tout d'abord demander au client d'envoyer une requête vers le service ciblé au serveur (paypal, facebook etc). Il va ensuite déterminer la tailles des blocs de la même manière que pour l'attaque POODLE (Cette longueur dépend du chiffrement utilisé : AES, DES, 3-DES, etc.) Ayant intercepté la requête du client, il connaît ainsi son texte chiffré et émet une hypothèse sur l'emplacement de l'information qu'il l'intéresse dans le bloc du chiffré. En effet, les requêtes HTTPS sont toutes de la même forme :

GET /path HT	TP/1.1\r\n	Cookie :	VRHnogWg	\r\n\r\nxxxx	xxxxM	AC •••3

Ici, l'attaquant sera intéressé par le bloc 3 et 4 contenant le cookie. De plus, dans cette requête l'attaquant peut modifier certaines données par l'intermédiaire du client comme le *path* et les *data* représenté par des xxx. Pour le reste, le *cookie*, ou *sessionid* est secret et cette partie de la requête ne peut être connue ni modifiée par l'attaquant. Mais cette forme spécifique de la requête donne un renseignement sur l'emplacement des données sensibles. C'est à dire que l'attaquant va être capable de déterminer quel bloc chiffré contient les données que l'on veut déchiffrer.

Pour sa prochaine requête, l'attaquant va confectionner un message bien précis qu'il fera envoyer par le client au serveur afin que celui ci le chiffre et l'envoie au serveur. Le vecteur d'initialisation sera ici égale au dernier bloc de la précédente requête, il est donc connu de l'attaquant.

Posons :
— P_i : le bloc de texte clair que l'attaquant désire identifier.
— C_i : le bloc de texte crypté de P_i.
— P'_0 : le bloc de texte clair que l'attaquant envoie au client.
— C'_0 : le bloc de texte crypté de P'_0.

— x : la supposition de la valeur de P_i par l'attaquant.

En utilisant les propriétés du mode CBC et le fait que l'IV de la première requête soit connue, l'attaquant va confectionner un bloc de la forme :

$$\begin{aligned} P'_0 &= C_{i-1} \oplus x \oplus C_n \\ P'_0 &= C_{i-1} \oplus x \oplus IV \end{aligned} \tag{3.1}$$

Cette opération n'est pas du au hasard en effet quand le P'_0 sera envoyé, le chiffrement donnera :

$$\begin{aligned} C'_0 &= E_k(P'_0 \oplus IV) \\ C'_0 &= E_k(C_{i-1} \oplus x \oplus IV \oplus IV) \\ C'_0 &= E_k(C_{i-1} \oplus x) \end{aligned} \tag{3.2}$$

L'attaquant va alors comparer le chiffré obtenu. Si la valeur de x est bien équivalente à P_i, on aura :

$$\begin{aligned} E_k(P'_0 \oplus IV) &= E_k(P_i \oplus C_{i-1}) \\ E_k(P'_0 \oplus IV) &= C_i \\ C'_0 &= C_i \end{aligned} \tag{3.3}$$

L'attaquant aura donc déchiffrer le bloc contenant l'information sensible.

3.3.3 Altération d'un octet

Il serait impossible pour un attaquant de déterminer un bloc P'_0 entier même avec un nombre très élevé de requêtes car la probabilité serait de $(1/256)^n$ ou n représente la taille d'un bloc. Il va donc falloir qu'il modifie un peu la requête envoyée au serveur pour faire en sorte de décrypter un octet après l'autre et ainsi avoir un probabilité de 1/256 à chaque octets.

Pour se faire, l'attaquant va utiliser le fait que le le */path* est modifiable, et il va rajouter autant d'octets qu'il le faut pour pouvoir connaître tout les octets dans un bloc excepté le dernier.
Cela nous donne un schéma de ce type avec une taille de bloc de 8 octets :

GET /AAA	AAAA HTT	P/1.1\r\nC	ookie :V	RHnogWg\r	....

Dans cet exemple, l'attaquant ne connaît pas les blocs contenant les mots : *Cookie : VRHnogWg*, il va donc ajouter des octets tel qu'un "A" dans le chemin de la requête pour pouvoir mettre le premier octet qu'il ne connaît pas tout seul, dans notre exemple le C de Cookie avec d'autres octets qu'il connaît. L'attaquant va alors ajouter aux 7 octets qu'il connaît un octet sur les 256 possibles, nommons se bloc x.

Il va ensuite construire P'_0 faire en envoyer cette requête modifiée au client. Si il voit que le chiffrement $C'_0 = C_3$, alors il aura trouvé $P'_0 = P_3$. Comme l'attaquant connaît les 7 premier octets sur l'ensemble du bloc, il sait que c'est le $8^{\text{ème}}$ octet qu'il a bien deviné.

Ensuite, il lui suffit simplement de retirer un octet du */path* pour tout décaler et déterminer un autre caractère jusqu'à obtenir l'ensemble des octets du cookie.

GET /AAA	AAAA HTT	P/1.1\r\n**C**	ookie :V	RHnogWg\r	....

TABLE 3.1 – requête 1

$P'_0 = C_{i-1} \oplus P/1.1\backslash r\backslash n \oplus C_n$	....

TABLE 3.2 – requête 2

Où C_n est le dernier chiffré de la requête 1 qui correspond à l'IV de la requête 2. L'attaquant compare le chiffré de : P'_0 et "P/1.1\r\n**C**"

3.4 Résumé de l'attaque

L'attaque peut se résumer en 11 étapes :

1. Faire envoyer au client un requête contenant une information sensible vers un serveur
2. Intercepter la requête
3. Déterminer la taille des blocs
4. Déterminer dans quel bloc chiffré se situe l'information sensible
5. Faire envoyer au client des requêtes en jouant sur le */path* pour placer un octet du secret dans un bloc (voir Altération d'un octet)
6. Récupérer le dernier bloc de la dernière requête envoyée, se sera maintenant l'IV de la seconde requête que le client fera.
7. Construire : $P'_0 = C_{i-1} \oplus x \oplus IV$
8. Placer le résultat dans le premier bloc de la requête en faisant en sorte que le dernier octet du bloc soit celui que l'on cherche
9. Faire envoyer la requête au client
10. Analyser le résultat et regarder si le bloc 1 chiffré correspond au bloc chiffré de l'étape 4 sinon recommencer à partir de l'étape 6
11. Décaler les octets et recommencer à partir de l'étape 5

3.5 Faisabilité

Si la théorie expliquée précédemment pouvait s'appliquer à la pratique, cette à attaque serait redoutable. Malheureusement, il est impossible pour un attaquant de réécrire complètement le bloc 1 d'une requête car le *POST GET* sont des octets ajoutés par le navigateur et l'attaquant n'a aucun contrôle dessus. Il ne peut donc pas complètement remplacer le bloc 1 ce qu'il l'empêche de mener l'attaque à bien. Une solution corrigée maintenant a tout de même était trouvée par Thai Duong et Juliano Rizzo, consistant à utiliser le HTML5 WebSocket API et la fonction `send(...)` [7].

Nous avons quand même développé l'attaque pour montrer qu'elle est très facilement réalisable si il n'y avait pas ce problème d'octet non modifiable, c'est à dire que nous avons réécris le

bloc un dans sa totalité ce qui est impossible dans la pratique. `https://github.com/mpgn/BEAST-exploit`

3.6 Sécurisation

Pour palier à cette attaque il y a une seul contre mesure efficace : passer à la version TLS1.1 qui génère maintenant aléatoirement le vecteur d'initialisation de manière à rendre imprévisible la valeur d'un bloc crypté à chaque requêtes.

Pour contrer BEAST, on aurait pu utiliser le mode RC4 plutôt que CBC, c'est à dire un chiffrement par flot. Bien que cette mesure permet d'annuler l'attaque BEAST, RC4 est aussi sujet à de nombreuses attaques[8] et d'un point de cryptographie, il ne devrait pas être utilisé. Il y a en effet une probabilité plus grande de subir une attaque exploitant une faille sur RC4 qu'une attaque de type BEAST.

Bibliographie

[1] ANSSI, *CERTFR-2014-ALE-007*. 15 Octobre 2014.

[2] A. Langley, *POODLE attacks on SSLV3*. 14 Octobre 2014.

[3] K. K. Bodo Möller, Thai Duong, *This POODLE Bites : Exploiting The SSL 3.0 Fallback*. Septembre 2014.

[4] D. F. Franke, *How POODLE Happened*. Octobre 2014.

[5] D. S. Henson, *Commit : Add heartbeat extension bouds check*. 6 Avril 2014.

[6] T. Duong and J. Rizzo, *Here Come the Ninjas*. Mai 2011.

[7] T. Duong and J. Rizzo *Here Come the Ninjas*, pp. 9–10, 2011.

[8] N. Sullivan., *Killing RC4 : The Long Goodbye*. 7 Mai 2014.