# BEAST - Browser Exploit Against SSL/TLS

**Livrables :**

- Description de BEAST avec vos mots.
- Votre méthodologie d'attaque.
- Défis et solutions.
- Captures d'écran.
- Le cookie déchiffré.

**Code Source :**

- Vos scripts et programmes.

## Livrables

## Description de BEAST avec vos mots.

BEAST (Browser Exploit Against SSL/TLS) est une attaque ciblant le protocole TLS 1.0, exploitant une vulnérabilité dans le mode de chiffrement CBC (Cipher Block Chaining). Cette faille, connue sous la référence CVE-2011-3389, se base sur l'utilisation de vecteurs d'initialisation (IV) prévisibles, compromettant ainsi la confidentialité des communications sécurisées. Elle a été patchée dans la version TLS 1.1 en 2006. Cependant, la majorité des serveurs web n’avaient pas fait la mise à jour nécessaire et étaient toujours sur la version obsolète.

### Un peu de contexte

Dès 2002, le cryptographe Phillip Rogaway avait mis en évidence des faiblesses théoriques dans l'utilisation du mode CBC au sein de TLS 1.0, notamment en raison de l'usage de IV prévisible (plus de détails dans le fonctionnement de l’attaque). Ce léger problème a conduit à la publication de TLS 1.1 en 2006, qui introduisait des IV aléatoires pour chaque message. Cependant, l'adoption de cette nouvelle version a été lente, laissant de nombreux systèmes exposés. Malgré tout, ces flemmards n’étaient pas pressés

de faire la mise à jour car pour casser le chiffrement par bloc, il y avait 256^8 combinaisons possibles (supposons ici un CBC avec des blocs de 8 octets.)

En septembre 2011, les chercheurs Thai Duong et Juliano Rizzo ont présenté à la conférence Ekoparty une démonstration concrète de l'attaque BEAST. En exploitant la vulnérabilité du mode CBC, ils ont réussi à déchiffrer des cookies de session HTTPS, notamment ceux de PayPal, en temps quasi réel. Leur méthode permettait de récupérer les données sensibles octet par octet, réduisant considérablement la complexité de l'attaque.

### Fonctionnement de l'attaque

Pour comprendre la dangerosité de BEAST, il faut savoir que dans TLS 1.0, le chiffrement en mode CBC utilise le dernier bloc chiffré du message précédent comme IV pour le message suivant. Pour rappel, l'IV est supposé être une valeur complètement aléatoire. Or cette prévisibilité permet à un attaquant, positionné en tant que Man-in-the-middle, d'injecter des données spécifiques dans le flux chiffré et d'observer les réponses du serveur pour déduire le contenu des messages. En répétant cette opération, l'attaquant peut progressivement reconstituer des informations sensibles, telles que des cookies de session. Pour donner un ordre d'idée, cette vulnérabilité a diminué le nombre de combinaisons possibles de 256^8 à 50 combinaisons en moyenne, ce qui est un gain de temps considérable. Ce proof of concept a démontré la dangerosité de l'utilisation TLS 1.0 en montrant qu'il était devenu très facile à exploiter.

### Réactions et mesures d'atténuation

La démonstration de BEAST a provoqué (enfin) une réaction rapide des entreprises qui ont mis en place plusieurs mesures temporaires pour atténuer la vulnérabilité, comme par exemple :

- Record splitting 1/n-1 : Cette technique consiste à fragmenter les messages TLS en insérant un petit bloc vide avant le message principal, perturbant ainsi le schéma de chiffrement prévisible. Elle a été implémentée dans des navigateurs comme Chrome et Firefox, ainsi que dans les bibliothèques OpenSSL.
- Utilisation de RC4 : Certains ont recommandé l'utilisation du chiffrement RC4, un algorithme de flux non vulnérable à BEAST. Cependant, RC4 a été

par la suite considéré comme peu sûr et son usage a été déconseillé.

Le temps que tout le monde fasse la mise à jour vers TLS 1.1 ou 1.2 qui corrigent la vulnérabilité en utilisant des IV aléatoires (vrai de vrai jvous jure) pour chaque message, cependant leur adoption a été lente en raison de problèmes de compatibilité avec certains systèmes.

**Conséquences**

Bien que la vulnérabilité ait été identifiée et corrigée dans les versions ultérieures de TLS, de nombreux systèmes ont continué à utiliser TLS 1.0 pendant plusieurs années, maintenant ainsi une surface d'attaque exploitable. Ce n'est qu'en mars 2021 que TLS 1.0 et 1.1 ont été officiellement dépréciés par l'IETF, encourageant les organisations à migrer vers des versions plus sécurisées.

## Votre méthodologie d'attaque.

1. Mise en place d’environnement
    a. docker file donné
    b. wireshark
2. récupération des POCs BEAST
3. capturer les paquets wireshark
4. script javascript (pour déchiffrer le cookie ??)
5. mise en place du script python

Avant d’établir une méthodologie d’attaque, nous avons mis en place l’environnement suivant:

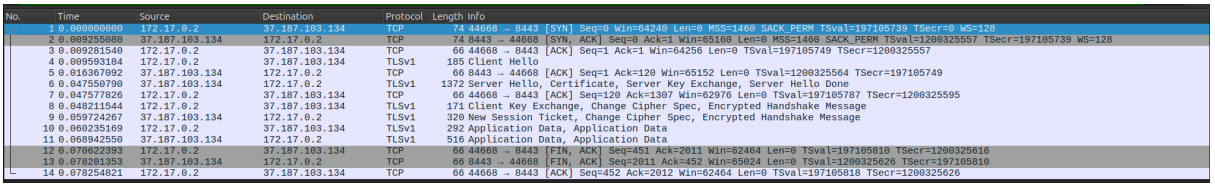
- Un docker file donné, qui jouera le rôle du client et de la victime et est spécialement conçu pour utiliser TLS 1.0.
- Notre PC principal qui jouera le rôle de l’attaquant et du Man-in-the-Middle.
- Un site web (déjà fait).

Nous avons également recherché sur internet plusieurs POCs de BEAST afin de pouvoir les tester lorsqu’ils seront nécessaires. Voici une petite liste des POCs que nous avons trouvé:

- https://github.com/yaoyi2008/BEAST-exploit
- BEAST POC: https://gitlab.cs.ui.ac.id/cis-fasilkom-ui/kelompok-8/-/tree/bc92a1145f23626ef227683fb3a1d9f103a9672f/BEAST-PoC
- https://github.com/mpgn/BEAST-PoC/blob/master/BEAST-poc.py
- https://github.com/ckasidis/tls-attacks-poc

Ils nous serviront plus tard lors des tests.

Après s'être assuré que tout fonctionnait, on a lancé Wireshark sur le PC de l'attaquant (MITM) afin de voir les échanges qui se font entre le client et le serveur web. On remarquera dans la capture suivante les échanges (filtrés avec 'tls') entre la machine docker et le serveur web:



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000000 | 172.17.0.2 | 37.187.103.134 | TCP | 74 | 44668 → 8443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=197105739 TSecr=0 WS=128 |
| 2 | 0.009255080 | 37.187.103.134 | 172.17.0.2 | TCP | 74 | 8443 → 44668 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=1200325557 TSecr=197105739 WS=128 |
| 3 | 0.009281540 | 172.17.0.2 | 37.187.103.134 | TCP | 66 | 44668 → 8443 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=197105749 TSecr=1200325557 |
| 4 | 0.009593184 | 172.17.0.2 | 37.187.103.134 | TLSv1 | 185 | Client Hello |
| 5 | 0.016367092 | 37.187.103.134 | 172.17.0.2 | TCP | 66 | 8443 → 44668 [ACK] Seq=1 Ack=120 Win=65152 Len=0 TSval=1200325564 TSecr=197105749 |
| 6 | 0.047550790 | 37.187.103.134 | 172.17.0.2 | TLSv1 | 1372 | Server Hello, Certificate, Server Key Exchange, Server Hello Done |
| 7 | 0.047577826 | 172.17.0.2 | 37.187.103.134 | TCP | 66 | 44668 → 8443 [ACK] Seq=120 Ack=1307 Win=62976 Len=0 TSval=197105787 TSecr=1200325595 |
| 8 | 0.048211544 | 172.17.0.2 | 37.187.103.134 | TLSv1 | 171 | Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message |
| 9 | 0.059724267 | 37.187.103.134 | 172.17.0.2 | TLSv1 | 320 | New Session Ticket, Change Cipher Spec, Encrypted Handshake Message |
| 10 | 0.060235169 | 172.17.0.2 | 37.187.103.134 | TLSv1 | 292 | Application Data, Application Data |
| 11 | 0.068942550 | 37.187.103.134 | 172.17.0.2 | TLSv1 | 516 | Application Data, Application Data |
| 12 | 0.070622393 | 172.17.0.2 | 37.187.103.134 | TCP | 66 | 44668 → 8443 [FIN, ACK] Seq=451 Ack=2011 Win=62464 Len=0 TSval=197105810 TSecr=1200325616 |
| 13 | 0.078201353 | 37.187.103.134 | 172.17.0.2 | TCP | 66 | 8443 → 44668 [FIN, ACK] Seq=2011 Ack=452 Win=65024 Len=0 TSval=1200325626 TSecr=197105810 |
| 14 | 0.078254821 | 172.17.0.2 | 37.187.103.134 | TCP | 66 | 44668 → 8443 [ACK] Seq=452 Ack=2012 Win=62464 Len=0 TSval=197105818 TSecr=1200325626 |

> Encrypted Application Data [truncated]: 30d96360468308c5b01664c15e9625856c56246c735b92b840a6523f61eef057ad88a22db6b12e1e83461d81e23e44f64ca8670a7c02782adf0412f6da33b3125bc955461b944824a3e43863f3bbe083925861053ab5a8ea113f89b2f1c0416730815eb

Nous avons ensuite décidé de tester les différents POC trouvés (cf liste), cependant, aucun d'eux n'a fonctionné pour notre cas. Malgré la modification des codes pour essayer de s'adapter n'ont pas été fructueuses. Ainsi, on a décidé d'essayer de développer notre propre script python afin d'exploiter la vulnérabilité TLS 1.0.

En parallèle, nous essayons de rédiger le script javascript que l'attaquant enverra à partir du navigateur du client (victime).
Ce script effectuera exactement ce que le POC https://github.com/mpgn/BEAST-PoC/blob/master/BEAST-poc.py effectue mais en javascript.

## Défis et solutions.

Malgré la simplicité théorique de l’exploitation de cette vulnérabilité, il est nécessaire de réunir plusieurs conditions :

- Réussir à se positionner en tant que Man-in-the-Middle
- Réussir à injecter et exécuter un code malveillant en JavaScript sur le navigateur client.
- Ce code permet de mettre un octet que l’on connaît dans la requête TLS et en connaissant celui-ci nous pouvons trouver le cookie.
- Déchiffrer le cookie
- Coder un script python qui permettrait d’automatiser le processus (j’ai un doute)

### Captures d'écran.

### Le cookie déchiffré.

## Code Python POC pour le décryptage d’un octet

```
#!/usr/bin/env python

import os
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
import string

# Generate IV and key
first_iv = os.urandom(16)
key = os.urandom(16)

def encrypt_cbc(iv, key, plaintext):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_plaintext = pad(plaintext, AES.block_size)
    ciphertext = cipher.encrypt(padded_plaintext)
    return ciphertext

def decrypt_cbc(iv, key, ciphertext):
```

```
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_plaintext = cipher.decrypt(ciphertext)
    plaintext = unpad(padded_plaintext, AES.block_size)
    return plaintext

# -------------------------
# Ground truth
# -------------------------
real_plaintext = b"bbbbbbbA"
target_ciphertext = encrypt_cbc(first_iv, key, real_plaintext)

# -------------------------
# Brute-force F replacement
# -------------------------
base = b"bbbbbbb"
charset = string.printable.encode()

for guess_byte in charset:
    guess = base + bytes([guess_byte])
    test_ciphertext = encrypt_cbc(first_iv, key, guess)

    print("Guessing: " + chr(guess_byte) + " → " + test_ciphertext.hex())

    if test_ciphertext == target_ciphertext:
        print(f"[+] Found match! Character: {chr(guess_byte)}")
        print(f"[+] Plaintext is : {base.decode() + chr(guess_byte)}")
        break
else:
    print("[-] No match found.")
```