**Rapport d'Évaluation de Sécurité :**

**Exploitation de la Faille BEAST**

Table des matières

[Partie 1 : Présentation et Compréhension de la Faille 2](#_Toc198205831)

[Partie 2 : Méthodologie Technique 4](#_Toc198205832)

[Partie 3 : Contre-mesures et protections efficaces face à l’attaque BEAST 7](#_Toc198205833)

[Partie 4 : Démonstration de l’attaque BEAST 8](#_Toc198205834)

[Partie 5 : Résultat de notre attaque 10](#_Toc198205835)

[Partie 6 : Défis Rencontrés et Solutions Apportées 11](#_Toc198205836)

[Partie 7 : Conclusion 12](#_Toc198205837)

[Partie 8 : Livrables 13](#_Toc198205838)

## Partie 1 : Présentation et Compréhension de la Faille

Nom de la faille : BEAST (Browser Exploit Against SSL/TLS)

**Description générale :**

BEAST est une attaque cryptographique révélée en 2011 par les chercheurs Thai Duong et Juliano Rizzo. Elle vise les connexions sécurisées utilisant le protocole TLS 1.0 (et, dans une moindre mesure, SSL 3.0) lorsqu'elles emploient un chiffrement par blocs en mode CBC (Cipher Block Chaining).

Bien que les algorithmes de chiffrement utilisés, comme AES ou 3DES, soient mathématiquement sûrs, la manière dont CBC est implémenté dans TLS 1.0 introduit une faiblesse. Cette vulnérabilité repose principalement sur la gestion du vecteur d’initialisation (IV), qui dans TLS 1.0 est défini comme étant le dernier bloc chiffré du message précédent. Cette prévisibilité permet à un attaquant en position d’interception (Man-in-the-Middle) d'exploiter cette propriété pour deviner des informations sensibles telles que des cookies de session.

**Qu’est-ce que TLS et le mode CBC ?**

TLS (Transport Layer Security) est un protocole cryptographique assurant la confidentialité, l’intégrité et l’authenticité des communications entre deux entités, généralement un client (navigateur) et un serveur.

CBC (Cipher Block Chaining) est un mode opératoire pour les algorithmes de chiffrement par blocs. Chaque bloc de texte clair est d'abord combiné (XOR) avec le bloc chiffré précédent avant d’être chiffré à son tour. Le premier bloc nécessite un vecteur d’initialisation (IV), qui est censé être aléatoire pour garantir que deux messages identiques produisent des chiffrés différents.

Dans TLS 1.0, le IV utilisé pour chiffrer un nouveau message n’est pas aléatoire, mais est défini comme étant le dernier bloc chiffré du message précédent. Cette particularité affaiblit la sécurité du protocole.

**Objectif de l’attaque :**

L’objectif principal de BEAST est de récupérer, octet par octet, des données sensibles chiffrées dans une session TLS, telles que les cookies HTTP de session. Pour cela, l’attaque repose sur les éléments suivants :

* Intercepter et analyser une session TLS chiffrée, généralement via une attaque de type Man-in-the-Middle (MitM).
* Deviner progressivement le contenu chiffré en injectant du contenu contrôlé dans les requêtes.
* Comparer les blocs chiffrés générés par ces requêtes à ceux contenant les données sensibles pour identifier les octets du message original.

La cible typique de l’attaque est le cookie de session contenant les jetons d’authentification.

**Compréhension technique détaillée : Fonctionnement de l’attaque BEAST**

1. Contexte : fonctionnement de CBC dans TLS 1.0

Dans TLS 1.0, le mode CBC est utilisé avec une particularité problématique : le vecteur d’initialisation (IV) d’un message est fixé au dernier bloc chiffré du message précédent. Cela rend le IV entièrement prévisible par un attaquant qui peut observer le trafic réseau. Contrairement à TLS 1.1 et TLS 1.2, qui utilisent un IV aléatoire et unique à chaque message, cette méthode viole le principe d’imprévisibilité essentiel à la sécurité du chiffrement CBC.

2. Problème : IV prévisible

Lorsque le IV est prévisible, il devient possible de manipuler le texte clair d’une requête de façon à ce qu’un bloc spécifique produise un chiffré identique à celui que l’attaquant cherche à deviner. Cela permet d'effectuer des tests de type bruteforce sur chaque caractère et de valider les hypothèses en comparant les résultats obtenus avec les blocs chiffrés observés.

3. Conséquence : attaque de type oracle

Dans une attaque par oracle, l’attaquant envoie une requête, observe la réponse (ou un bloc chiffré généré en réponse), puis utilise cette observation pour en déduire une partie du message clair. Dans BEAST :

* L’attaquant est en position MitM.
* Il injecte du contenu contrôlé juste avant une donnée secrète (par exemple un cookie).
* Il modifie ce contenu à chaque requête pour forcer une certaine structure et comparer les blocs résultants.
* Cette approche permet de déduire un caractère à la fois.

4. Déroulement de l’attaque

Étape 1 : Injection de JavaScript dans le navigateur de la victime  
L’attaquant héberge une page web non sécurisée (HTTP) contenant du JavaScript malveillant. Lorsque la victime visite cette page, le script peut envoyer des requêtes HTTPS vers un autre site (le site cible contenant des cookies).

Étape 2 : Génération de requêtes modifiées  
Le JavaScript insère du contenu contrôlé dans les requêtes (dans les en-têtes ou le corps), immédiatement avant la donnée secrète. À chaque itération, une nouvelle valeur est testée pour obtenir différentes sorties chiffrées.

Étape 3 : Alignement des blocs  
La donnée secrète (comme le cookie) est alignée de façon à commencer au début d’un bloc CBC, ce qui permet d’isoler et de cibler précisément un bloc.

Étape 4 : Observation et comparaison des blocs chiffrés  
L’attaquant, via le proxy MitM, intercepte les requêtes chiffrées de la victime. Il compare les blocs chiffrés générés à ceux qu’il connaît, et identifie les correspondances avec ses propres tests.

Étape 5 : Répétition du processus  
Cette procédure est répétée pour chaque octet, ce qui permet de récupérer toute la donnée chiffrée, caractère par caractère.

**Exemple pratique :**

Supposons qu’un cookie contienne la valeur suivante :

session=abc123xyz456

L’attaquant injecte des en-têtes comme :

X-Prefix: A

X-Prefix: B

X-Prefix: C

...

En observant le bloc chiffré contenant le cookie, il identifie à quel moment le bloc généré par une de ses requêtes correspond à celui qu’il cherche à deviner. Une correspondance indique que la lettre testée est correcte.

**Pourquoi cette faille est critique :**

* Elle montre qu’un protocole peut être compromis non pas à cause d’un algorithme faible, mais en raison de détails d’implémentation.
* Bien que AES ou 3DES soient sûrs en soi, la prévisibilité du IV dans TLS 1.0 permet une attaque pratique et efficace.
* Elle est applicable dans des scénarios réels, notamment sur des navigateurs web et des serveurs utilisant TLS 1.0.
* Elle a conduit à l’abandon progressif de TLS 1.0 au profit de TLS 1.2 et TLS 1.3, qui intègrent :
  + Des vecteurs d’initialisation aléatoires,
  + Des modes de chiffrement comme GCM (Galois/Counter Mode),
  + Une gestion renforcée des données sensibles.

## Partie 2 : Méthodologie Technique

**Environnement utilisé :**

Pour reproduire l’attaque BEAST dans un environnement contrôlé, une infrastructure simplifiée a été mise en place à l’aide de conteneurs et d’outils d’analyse réseau. Voici les éléments de l’environnement :

* **Système d’exploitation** : Conteneur Debian exécuté via Docker
* **Protocole TLS** : TLS 1.0 forcé avec chiffrement en mode CBC (suite AES128-SHA)
* **Navigateur simulé** : Script client.py imitant un navigateur envoyant des requêtes HTTP(S)
* **Outil d’interception réseau** : mitmproxy ou wireshark utilisé comme proxy pour observer le trafic TLS
* **Langage de script** : Python avec la bibliothèque requests pour simuler des injections et manipulations de requêtes

**Étapes réalisées :**

**1. Construction de l’environnement vulnérable**

Le but est de simuler un serveur web supportant uniquement TLS 1.0 avec chiffrement CBC. Pour cela :

docker build -t tls10-python .

docker run --rm -it --name tls10-container -v "$(pwd)":/app -w /app tls10-python

* TLS est explicitement configuré pour n’accepter que la version 1.0.
* Le serveur utilise la suite de chiffrement AES128-SHA.
* Les paquets nécessaires sont installés : openssl, libssl1.1, curl, requests.

Cela permet de reproduire un environnement cryptographiquement vulnérable et conforme aux conditions de l’attaque BEAST.

**2. Mise en place du proxy MitM (mitmproxy)**

mitmproxy est configuré pour intercepter les requêtes HTTPS envoyées par le navigateur simulé (via client.py). Ce proxy joue un rôle central :

* Il intercepte le trafic TLS de la victime.
* Il observe et enregistre les blocs chiffrés échangés.
* Il extrait automatiquement les cookies envoyés dans les en-têtes HTTP.

Ce positionnement en "Man-in-the-Middle" est crucial pour mener l’attaque BEAST, puisqu’il permet de suivre l’évolution des blocs CBC et de capturer les données ciblées.

**3. Observation du comportement CBC**

Une série de requêtes identiques ou légèrement modifiées est envoyée à travers TLS 1.0. Grâce à mitmproxy :

* Les blocs chiffrés associés aux cookies sont comparés d'une requête à l'autre.
* On observe que l’IV du message est bien le dernier bloc du message précédent.
* Cela confirme la vulnérabilité spécifique à TLS 1.0 : l’IV est prévisible.

Ce comportement est un prérequis indispensable pour que l’attaque par oracle fonctionne.

**4. Injection contrôlée de JavaScript côté victime**

Un script JavaScript est injecté dans une page HTTP servie en clair. Ce script agit comme une charge active côté navigateur :

* Il force l’envoi de requêtes HTTPS vers le site cible.
* Il modifie dynamiquement les en-têtes HTTP ou le corps des requêtes.
* Le contenu injecté est contrôlé entièrement, ce qui permet de manipuler l’alignement du cookie dans les blocs CBC.

Cette étape repose sur un scénario de type XSS ou injection dans une page non sécurisée chargée par la victime.

**5. Alignement des blocs**

Pour que l’attaque fonctionne, il faut que le cookie (ou autre secret) commence **exactement au début d’un bloc CBC**. Cela permet de cibler un bloc unique, facilitant la comparaison.

* Des données contrôlées sont insérées avant le cookie, décalant son emplacement dans la requête.
* Le script effectue des tests d’alignement jusqu’à trouver une position idéale.
* Une fois aligné, un bloc spécial est construit pour tester chaque caractère possible du cookie.

Ces "guessing blocks" sont soigneusement construits de manière à produire, lors du chiffrement, un bloc CBC que l’on pourra comparer à la version chiffrée réelle.

**6. Comparaison des blocs chiffrés**

L’attaquant, via mitmproxy, observe les blocs chiffrés générés à chaque tentative.

* Si un bloc chiffré généré correspond exactement à celui extrait du trafic réel, alors le caractère deviné est correct.
* Ce processus est répété pour chaque octet du cookie ou du secret, jusqu’à reconstruction complète.

L’efficacité repose sur le fait que TLS 1.0 génère toujours les mêmes blocs chiffrés lorsqu’un bloc clair est combiné avec un IV prévisible de manière identique.

## Partie 3 : Contre-mesures et protections efficaces face à l’attaque BEAST

L’attaque BEAST tire parti d’une faiblesse spécifique dans le mode CBC utilisé avec TLS 1.0. Pour contrer cette attaque, plusieurs approches techniques et organisationnelles ont été mises en place à différents niveaux (client, serveur, protocole). Ces mesures ont pour but de corriger la faille à la source ou de l'atténuer jusqu’à rendre l'exploitation irréaliste.

**1. Mise à jour du protocole TLS**

**a. Passage à TLS 1.1 ou TLS 1.2 (et TLS 1.3)**

La solution principale, radicale et définitive contre BEAST est d’abandonner TLS 1.0 (et SSL 3.0).

* **TLS 1.1** introduit un vecteur d’initialisation aléatoire pour chaque message, éliminant ainsi la prévisibilité de l’IV.
* **TLS 1.2** offre non seulement un IV aléatoire mais aussi la prise en charge de modes de chiffrement alternatifs comme GCM (Galois/Counter Mode), qui ne reposent pas sur CBC.
* **TLS 1.3** supprime entièrement le mode CBC, ne conservant que des suites modernes (basées sur AEAD).

**Conclusion** : L’adoption de TLS ≥ 1.1 corrige structurellement la faille exploitée par BEAST.

**2. Désactivation du support de TLS 1.0 et SSL 3.0**

Les serveurs web doivent refuser les connexions utilisant des versions vulnérables :

* Configurer les serveurs (Apache, Nginx, etc.) pour interdire SSL 3.0 et TLS 1.0.
* Autoriser uniquement TLS 1.2 et TLS 1.3.

**Exemple de configuration Nginx** :

ssl\_protocols TLSv1.2 TLSv1.3;

ssl\_ciphers 'ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256';

**3. Priorisation des suites de chiffrement modernes (GCM)**

Même si TLS 1.2 est utilisé, il est possible que CBC soit encore activé. Il faut donc :

* Prioriser les suites utilisant **GCM** (AEAD — Authenticated Encryption with Associated Data), comme AES-GCM ou ChaCha20-Poly1305.
* Éviter les suites AES-CBC ou 3DES.

GCM supprime complètement la notion de blocs chaînés, donc l’attaque BEAST ne peut pas s’appliquer.

**4. Protection côté client (navigateur)**

**a. Réglages et patchs dans les navigateurs**

Dès 2011-2012, les navigateurs ont introduit des contre-mesures contre BEAST, bien avant la désactivation massive de TLS 1.0 :

* **Blocage des suites CBC** côté client pour TLS 1.0.
* **Réordonnancement des préférences de chiffrement** pour favoriser les suites RC4 (à l’époque, considérées comme sûres contre BEAST).

RC4 est aujourd’hui considéré comme **insecure** et a été retiré. Cette solution était temporaire.

**b. Mise à jour des navigateurs**

Les versions modernes de Chrome, Firefox, Safari, Edge ont supprimé le support de TLS 1.0.  
Les utilisateurs doivent s’assurer d’utiliser un navigateur à jour pour éviter toute exposition à ce type d’attaque.

**5. Hardening des bibliothèques SSL/TLS**

**a. Patching d’OpenSSL et autres bibliothèques**

Les bibliothèques comme **OpenSSL**, **GnuTLS**, **NSS** ont été modifiées pour :

* Introduire des contre-mesures de type **record splitting** : découper les données en fragments pour empêcher la prédiction de blocs.
* Désactiver le support de TLS 1.0 par défaut dans les versions récentes.

**b. Record Splitting**

Une technique logicielle déployée pour casser l’attaque BEAST :

* Elle consiste à **envoyer le premier octet d’un message dans un record TLS séparé**.
* Cela empêche l’attaquant de placer un bloc manipulé immédiatement avant le secret.
* Implémentée automatiquement dans plusieurs bibliothèques et navigateurs entre 2011 et 2013.

**6. Protection côté application**

Même si BEAST est une attaque sur le protocole, des bonnes pratiques côté application réduisent son impact potentiel :

* Limiter la durée de vie des **cookies de session**.
* Marquer les cookies comme Secure et HttpOnly.
* Ne jamais exposer de secrets dans des URLs ou en clair.

**7. Détection et monitoring**

* Utilisation de **scanners de sécurité TLS** (ex : ssllabs.com, testssl.sh) pour identifier les suites CBC activées et le support de TLS 1.0.
* Surveillance des logs de connexion et alertes sur connexions utilisant des versions obsolètes.

## Partie 4 : Démonstration de l’attaque BEAST

**Objectif**

Illustrer le principe de l’attaque BEAST à travers une simulation contrôlée permettant de :

* Comprendre comment un attaquant peut intercepter et manipuler des requêtes,
* Observer la vulnérabilité de TLS 1.0 avec CBC,
* Reconstituer un cookie de session chiffré, octet par octet.

**1. Environnement de test**

**Composants utilisés** :

* Conteneur Docker avec TLS 1.0 activé
* Client simulé (client.py) utilisant curl
* Mitmproxy ou Wireshark pour capturer et manipuler le trafic
* Script Python pour l'automatisation de l'attaque (bruteforce)

**2. Dockerfile simplifié**

Cf annexe fichier Dockerfile

**4. Script client simulé (client.py)**

Cf annexe fichier client.py

Ce script envoie des requêtes HTTPS vers le serveur via mitmproxy ou wireshark, en ajoutant un en-tête X-Prefix contenant une lettre devinée.

**5. Script d’attaque Python**

Cf annexe dossier BEAST\_POC fichier BEAST.py ou alors le notre Beast.py

**6. Analyse via wireshark**

* Wireshark intercepte et affiche les blocs chiffrés des requêtes envoyées par client.py.
* L’attaquant exporte ces blocs (en hexadécimal) pour les comparer.
* Lorsqu’un bloc généré par le préfixe injecté est égal au bloc du cookie ciblé, le caractère est validé.

**7. Limites et contraintes**

* L’attaque nécessite un positionnement **MitM** fiable (souvent difficile à obtenir en pratique).
* Le trafic doit passer en **TLS 1.0 avec CBC**, ce qui est aujourd’hui rare.
* La présence d’un JavaScript injecté dans la page d’un site sécurisé est nécessaire pour manipuler les requêtes côté victime.

**Résultat attendu**

À la fin de l’attaque :

* Le cookie "SECRET123456" est reconstitué caractère par caractère.
* Le script Python peut l’exporter ou le réutiliser pour usurper l’identité de la victime (session hijacking).

**Conclusion de cette démonstration**

Cette attaque repose sur un alignement précis des blocs et la possibilité d’injecter des requêtes contrôlées. Bien qu’elle ait été atténuée dans la pratique par l’évolution des protocoles et des navigateurs, elle reste un exemple marquant d’exploitation cryptographique par **oracle CBC** dans un contexte web réel.

## Partie 5 : Résultat de notre attaque

**Cookie reconstitué :**  
Grâce à l’attaque par injection contrôlée et analyse des blocs CBC, le cookie a été récupéré avec succès : **token=SECRET123456**

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

## Partie 6 : Défis Rencontrés et Solutions Apportées

|  |  |
| --- | --- |
| **Défi rencontré** | **Solution apportée** |
| TLS 1.0 non supporté dans les navigateurs modernes | Mise en place d’un conteneur Docker configuré pour rétrograder OpenSSL et utilisation de curl forcé en TLS 1.0 |
| Cookies automatiquement envoyés par le navigateur, non manipulables | Création d’un JavaScript injecté (via HTTP) non sécurisé pour forcer les requêtes et contrôler le contenu |
| Alignement complexe des blocs CBC | Détection dynamique de l’offset du cookie en calculant l’offset exact, padding manuel pour forcer l’alignement du début du secret |
| Prévisibilité faible du bloc cible | Répétition de requêtes et injection de contenu contrôlé pour augmenter les chances de correspondance |
| Nombre de requêtes très élevé (brute-force par caractère) | Utilisation de multi-threading (ou asyncio) pour paralléliser les requêtes et mise en cache partielle |
| Python 3 ne peut pas accéder à TLS 1.0 | La majorité des versions modernes de Python (via requests, urllib3, etc.) désactivent TLS 1.0 pour des raisons de sécurité. Aucune solution native immédiate n'existe sans recompiler OpenSSL ou patcher les bibliothèques SSL. Donc utilisation d’un outil en ligne de commande (curl) compilé avec support explicite de TLS 1.0, ou utilisation d’anciennes versions de bibliothèques dans un environnement Docker contrôlé. |
| Le client (navigateur simulé) ne se connecte pas au serveur | TLS 1.0 étant obsolète et rejeté par les navigateurs modernes, il a été nécessaire d'utiliser un client simulé dans un environnement Docker, avec un script comme victim\_browser.sh utilisant curl --tlsv1 pour forcer la connexion. |
| Difficulté à capturer précisément les blocs chiffrés dans mitmproxy | mitmproxy étant un proxy applicatif HTTP(S), il déchiffre le contenu pour l’analyse, ce qui altère les blocs CBC originaux nécessaires à l’analyse cryptographique. La solution est de passer à Wireshark, qui permet une capture brute au niveau réseau. Avec Wireshark, il est possible de visualiser les blocs TLS sans altération, y compris les IV et les blocs CBC. Un filtrage fin sur les ports (443 ou personnalisé) a permis de repérer précisément les segments contenant les cookies. |

## Partie 7 : Conclusion

L’attaque BEAST, bien qu’ancienne, reste un cas d’école majeur dans le domaine de la cryptanalyse appliquée aux protocoles de communication sécurisés. La faille BEAST démontre qu’une mauvaise implémentation d’un protocole sécurisé (comme CBC dans TLS 1.0) peut rendre même les algorithmes les plus solides vulnérables à l’exploitation.

Ce projet a permis de mettre en lumière plusieurs enseignements fondamentaux :

**Apports techniques :**

* Compréhension du rôle crucial de l’IV (Initialisation Vector) dans la sécurité des modes CBC.
* Analyse détaillée du protocole TLS 1.0 et de ses failles structurelles.
* Exploitation concrète d’une faiblesse cryptographique via des moyens pratiques, combinant MitM, injection JavaScript, et alignement contrôlé.
* Construction d’un environnement sécurisé isolé, reproduisant des conditions réelles d’exploitation sans mettre en danger d’infrastructure externe.

**Enseignements en sécurité :**

* Un algorithme robuste (comme AES) peut devenir vulnérable à cause d’une mauvaise implémentation ou d’un protocole obsolète.
* L’évolution vers TLS 1.2+ et 1.3 est indispensable pour éliminer ces classes de failles (grâce au mode GCM et IV aléatoire).
* La nécessité de désactiver activement TLS 1.0/SSL 3.0 sur les serveurs encore en production.
* L’importance de l’aléa cryptographique dans la prévention des attaques par prédiction ou par oracle.

**Compétences renforcées :**

* Configuration de TLS dans un environnement personnalisé,
* Utilisation avancée de Docker pour la sécurité réseau,
* Analyse fine du comportement et fonctionnement bas niveau de CBC en TLS,
* Déploiement et utilisation d’outils de type Man-in-the-Middle (mitmproxy ou wireshark), avec une analyse du trafic réseau
* Développement de scripts d’attaque réseau automatisés.

**Ce projet m’a permis de comprendre en profondeur :**

* Comment une faiblesse dans un protocole sécurisé (comme l’IV dans CBC de TLS 1.0) peut mener à une faille critique.
* L’importance de l’IV imprévisible dans les algorithmes par blocs.
* Les subtilités du mode CBC et de son chaînage.
* Comment une faille cryptographique peut être exploitable en pratique par injection + MitM.
* L’importance de l’entropie et de l’aléa dans les mécanismes cryptographiques.
* Les implications réelles d’une attaque comme BEAST sur des connexions supposées sécurisées.
* L’intérêt de protocoles modernes comme TLS 1.2/1.3, qui corrigent ces problèmes structurels.

## Partie 8 : Livrables

Le projet s’accompagne des éléments suivants, disponibles en annexe ou en dossier joint :

**Rapport :**

* Ce document complet retraçant le contexte, l’approche, la méthodologie, l’expérimentation, les résultats et l’analyse de l’attaque BEAST.

**Scripts :**

* beast\_exploit.py : script Python principal effectuant l’attaque byte-par-byte.
* Client.py : script de simulation de requêtes HTTPS via curl avec injection de contenu contrôlé.
* Dockerfile : image Docker Debian configurée avec TLS 1.0 et environnement nécessaire (OpenSSL rétrogradé, Flask, curl, mitmproxy).
* Wireshark : montre les blocs TLS interceptés et utilisés pour l’analyse.

Beast.py

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Client.py

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Wireshark

Une image contenant texte, capture d’écran, logiciel, Page web

Le contenu généré par l’IA peut être incorrect.