1. Contexte

1.1 Contexte

Dans le cadre strict et légal d'une formation en cybersécurité offensive, cet exercice s'inscrit dans une approche pédagogique visant exclusivement la formation des étudiants aux techniques de simulation d'adversaires (Red Team). L'objectif est d'acquérir une compréhension approfondie des mécanismes de détection et des stratégies employées par les attaquants afin de mieux protéger les systèmes d'information.

Avertissement : Toute utilisation abusive des connaissances acquises dans un cadre illégal constitue une infraction pénale sévèrement réprimée par la loi. Cet exercice est réalisé sous supervision, dans un environnement contrôlé, avec des objectifs exclusivement pédagogiques et professionnels liés à la cybersécurité défensive et offensive dans un cadre légal.

1.2 Objectifs pédagogiques

Ce projet a pour but de permettre aux étudiants de :

- **Comprendre les mécanismes de détection des logiciels malveillants** : Apprentissage des méthodes utilisées par les antivirus, EDR, et sandboxes afin de mieux comprendre leurs limites et renforcer leur efficacité.
- **Développer des compétences avancées en analyse de malware** : Être capable d'identifier des techniques d'évasion lors d'une investigation forensic ou d'une analyse de compromission.
- **Renforcer les compétences en sécurité offensive pour les Red Teams** : Maîtriser les outils et techniques permettant de simuler des attaques réalistes dans un cadre autorisé et sous supervision, dans le but d'améliorer la posture de sécurité des entreprises.
- **Optimiser les stratégies de protection des systèmes d'information** : Apprendre à identifier les points faibles des solutions de cybersécurité et investir efficacement dans des technologies adaptées sans dépendre de solutions coûteuses inefficaces.

1.3 Cadre légal et éthique

Cet exercice est encadré par des professionnels de la cybersécurité et respecte les lois en vigueur. Il ne vise en aucun cas à encourager ou faciliter des activités illégales. Toute tentative d'application de ces techniques en dehors du cadre pédagogique et professionnel prévu dans cette formation est interdite et expose son auteur à des poursuites judiciaires.

2. Objectifs du projet

2.1 Construire un malware FUD modulaire

Vous devez élaborer un **malware flexible** permettant de facilement modifier ou patcher ses composants pour éviter une détection AV/EDR. L'architecture repose sur plusieurs **briques** :

1. **Le Runner (PE)**
    - Fichier exécutable **Windows** (Portable Executable), agissant comme **point d'entrée**.
    - Doit être **FUD** (détecté par un minimum d'antivirus).
    - Lance éventuellement un programme légitime (ou une fenêtre anodine) pour **ne pas attirer l'attention**.
    - Contient un mécanisme pour **exécuter le payload** malveillant (shellcode).
2. **PEB-based DLL lookup stub**
    - Permet de récupérer les fonctions de l'API Windows **sans les imports classiques** (afin de limiter la détection par signatures).
    - Se comporte comme une mini-bibliothèque interne, utilisable aussi bien par le runner que par le **shellcode**.
    - **Indispensable** pour tout shellcode Windows générique souhaitant être furtif.
3. **Le stageless (ou stager) payload**
    - Le **shellcode** principal (PIC – Position Independent Code).
    - Inclut directement (ou utilise) le **PEB-based DLL lookup stub** pour appeler la WinAPI.
    - Contient la **charge utile malveillante** (par exemple, un code offensif quelconque).
4. **L'anti-EDR honeypot**
    - Mécanismes de **détection d'environnement hostile** (sandbox, debug, machine virtuelle d'analyse).
    - Permet au malware de **s'auto-désactiver** pour éviter la détection si un environnement suspect est repéré.
    - Peut être présent **dans chaque brique** (runner, stub, payload) : les malwares avancés répliquent souvent ces checks partout.

2.2 Exigences fonctionnelles et techniques

- **Exécution conditionnelle** de la partie malveillante : doit contourner la surveillance en sandbox pour ne s'activer que dans un contexte "réel".
- **Bypass de l'analyse statique** (y compris par antivirus, Yara, etc.).
- **Bypass de l'analyse comportementale** (contrôle de la mémoire, hooking, Sysmon events).
- **Limitation des IOC (Indicators of Compromise)** sur le réseau.
- **Souplesse d'adaptation** : pouvoir modifier rapidement l'une des briques en cas de détection sans impacter tout le projet.

2.3 Pédagogie et méthodologie

Vous allez **décomposer** votre projet en **entités indépendantes** pour :

1. **Limiter la complexité globale** et isoler clairement les responsabilités (runner, stub, payload, anti-EDR).
2. Pouvoir rendre chaque composant **FUD** individuellement avant de les assembler.
3. **Tester et itérer** petit à petit, en validant chaque étape via votre VM Windows Defender et divers scanners en ligne.

3. Conseils détaillés pour chaque brique

3.1 Conseils pour le Runner

1. **Étapes initiales** :
    - Créer un `.exe` minimaliste, **sans comportement malveillant** ni shellcode, juste pour tester vos **options de compilation** (mingw, etc.) et éventuellement tenter des patchs binaires (avec Lief) afin d'obtenir un binaire **FUD**.
2. **Exécution d'un shellcode inoffensif** :
    - Introduire, dans un second temps, un **shellcode anodin** (par exemple, un pattern qui n'est pas louche ou qui fait simplement un `MessageBox`).
    - Vérifier avec **x64dbg** que l'exécution atteint bien la première instruction du faux shellcode.
    - S'assurer de rester **FUD** ou quasi en testant avec votre antivirus local (p. ex. Windows Defender) puis sur un service en ligne (VirusTotal, MetaDefender, etc.).
3. **Création du stub** :
    - Une fois un runner "correctement furtif" prêt, vous passerez à l'implémentation du **PEB-based DLL lookup stub** (dans le runner, ou sous forme de code partagé).
    - Tester encore avec un shellcode simple (du type `MessageBox`).
4. **Charge utile réelle** :
    - Quand tout est bien FUD, introduire votre **payload offensif**.

Astuce pour rester furtif :

- **Limiter la taille et l'entropie du binaire** : un exécutable trop "compacté" ou chiffré pourra être plus vite remarqué par des heuristiques.
- **Se rapprocher d'un "pattern standard"** d'exécutable Windows : utiliser un compilateur répandu (e.g. Visual Studio, mingw) peut aider, au prix d'une complexité de build plus élevée.
- **Règles Yara** : nombreuses sur Internet. Un binaire signé jouit parfois d'un "laxisme" relatif, mais ce n'est pas votre cas, donc prudence sur tout "pattern" suspect.

3.2 Conseils pour le stub (PEB-based DLL lookup)

1. **Évitez les classiques import tables** dans le runner pour que la résolution API se fasse en mémoire.
2. **Inspirez-vous** du code Metasploit (MSF) :
    - Ils ont beaucoup documenté leurs “offsets” (pour récupérer l’`InLoadOrderModuleList` du PEB, etc.) et la façon de minimiser la taille du stub.
3. **Attention au hashage** :
    - Les collisions, ou l’utilisation de hash déjà trop connus, peuvent causer des détections.
4. **Protocole d’appel** :
    - Respecter la convention d’appel Microsoft x64 (volatilité des registres, usage de `RCX/RDX/R8/R9` pour les 4 premiers arguments, etc.).
    - Éviter tout **pattern** trop classique d’obfuscation (souvent détecté par heuristique).
5. **Stratégies d’injection** :
    - Incorporer des “fausses instructions” pour tromper les analyses statiques (e.g., instructions NOP, jumps inutiles, etc.).

3.3 Conseils pour le payload (stageless/stager)

1. **Interactions avec le stub** :
    - Minimiser les appels suspects ou regrouper leur exécution dans une zone masquée.
    - Penser à l’**initialisation des chaînes** de caractère (strings).
2. **Exemples de stratégies** :
    - **call/pop** ou **push/pushRsp** pour récupérer l’offset d’une string.
    - Encodage/décodage à la volée (attention aux pages mémoire RX vs RW vs RWX).
3. **Anticipation de la détection** :
    - Ajouter des instructions “leurres” pour briser les patterns Yara.
    - Éviter d’utiliser des sections `.text` de taille anormale ou compressées.

3.4 Anti-EDR honeypot (détection d’environnement)

1. **Vérifier si on tourne dans une VM, un debugger, ou un sandbox** :
    - Détection de drivers suspects, de process typiques (`ProcMon`, etc.), de clés de registre, etc.
    - “Dormir” ou se neutraliser en cas d’environnement suspect.
2. **Répartition dans tout le code** :
    - Les malwares avancés disséminent ces checks dans **chaque brique** (runner, stub, payload) pour que l’analyse doive tout contourner.
3. **Exécution conditionnelle** :
    - Exécuter la charge malveillante uniquement si l’environnement paraît légitime, sinon faire croire à un fonctionnement normal et “anodin” (voire se terminer silencieusement).

4. Méthodologie de test

4.1 Utilisation d'une VM Windows Defender

- Toujours tester en local dans une machine virtuelle avec antivirus (Defender).
- **Évoluer par étapes** :
    1. Runner vide.
    2. Runner + petit shellcode "inoffensif".
    3. Ajout du stub.
    4. Payload plus offensif.
- Après chaque ajout, contrôler si vous restez FUD.

4.2 Soumission à des services d'analyse

- Outils en ligne (à utiliser de façon **raisonnée**, sous peine de contribuer à entraîner les AV) :
    - VirusTotal
    - Polyswarm
    - MetaDefender
    - Intezer
    - Hybrid Analysis
    - ANY.RUN
    - Triage
- **Attention** : ces services conservent parfois vos binaires. Les soumissions répétées d'un code très proche peuvent déclencher la création de nouvelles règles heuristiques (p. ex., Yara) et ruiner votre FUD.

4.3 Décomposition en briques et tests par élimination

- Exemple :
    - **Runner** seul (sans aucun comportement malveillant) → test FUD.
    - **Runner** + shellcode trivial → test.
    - **Stub** isolé (appelant des fonctions Windows "normales") → test.
    - **Payload** + quelques fonctionnalités → test.
    - **Anti-EDR** → test.
- Si vous identifiez une détection, vous savez **quelle brique** affiner en priorité.

5. Stratégies d’évasion (exemples concrets)

5.1 Gestion des pages mémoire (RW, RX, RWX)

1. **Exemple 1** :
    - Utiliser un **code automodifiant** ou encoder le malware via un encodeur MSF.
    - Problème : nécessite généralement une page **RWX** (exécution + écriture), ce que beaucoup d’EDR n’aiment pas.
    - Il faut alors s’assurer que l’encodeur lui-même n’est pas détecté.
2. **Exemple 2** :
    - **Encoder le payload à la compilation** (chiffrer le shellcode dans la section data du runner).
    - Le décoder au runtime **avant** de le copier dans une page RX (ou en modifiant les droits de la page).
    - Avantage : évite de laisser la page RWX.
    - Inconvénient : les **changements de droits mémoire** (VirtualProtect, etc.) peuvent aussi être détectés.
3. **Exemple 3** :
    - Avoir un **shellcode “FUD par nature”** pour qu’il s’exécute directement en RX.
    - Inconvénient : limite la possibilité de décoder des strings “à la volée” via `CALL/POP`.

5.2 Organisation et maintien de la furtivité

- **Bien séparer** le code :
    - Script de build paramétrable pour injecter différents shellcodes (ou options).
- **Ajouter/retirer** des briques rapidement.
- Chaque nouvelle fonctionnalité peut être testée **indépendamment** pour mesurer son impact sur la détection.

5.3 Exemples de détections Yara

- Règles Yara très nombreuses :
    - Exemple “gen_xor_hunting.yar”
    - Éviter les patterns (`xor`, `rol`, `add`, etc.) copiés-collés d’encodeurs publics.
- Ne pas négliger l’entropie globale des sections `.text`, `.rdata`.

6. Ressources et liens utiles

Sachez que la liste ci-dessous **n'est pas exhaustive**. Elle vous donne quelques points de départ :

- https://www.varonis.com/blog/malware-coding-lessons-people-part-learning-write-custom-fud-fully-undetected-malware
- https://hshrzd.wordpress.com/
- https://www.cnblogs.com/DirWang/p/16544561.html
- https://www.huntress.com/blog/hackers-no-hashing-randomizing-api-hashes-to-evade-cobalt-strike-shellcode-detection
- https://passthehashbrowns.github.io/dynamic-payload-generation-with-mingw
- https://www.zerodetection.net/blog/minimal-shellcode-loader-in-c-a-step-by-step-guide
- https://github.com/vxunderground/OCRMe
- https://github.com/vxunderground/VX-API
- https://github.com/Karneades/malware-persistence?tab=readme-ov-file#overview-of-difficult-to-detect-persistence-mechanisms
- https://global.ptsecurity.com/analytics/antisandbox-techniques
- https://www.apriorit.com/dev-blog/545-sandbox-evading-malware
- https://inria.hal.science/hal-01964222/document

7. Organisation des livrables

Vous devez rendre une archive `.tar.gz` structurée **exactement** comme suit :

1. `rapport.pdf`
    - Décrivant **clairement** :
        - Les différentes briques (runner, PEB-based stub, payload, anti-EDR).
        - Les dépendances et leur installation (mingw, lief, nasm, etc.).
        - La **méthodologie de test** : comment vérifier à chaque étape si l'exécutable est FUD.
        - Les **forces et faiblesses** de votre approche (techniques de dissimulation, limitations).
        - Votre **réflexion personnelle** sur vos choix d'architecture, d'encodage, etc.
2. **Dossier `./proj`**
    - Contenant **tous les fichiers** nécessaires à la compilation (code source).
    - Un **script exécutable** `build.sh` (appelé depuis la racine de l'archive via `./proj/build.sh`).
    - Ce script doit **afficher** dans la console où se trouve l'exécutable final à l'issue de la compilation.
3. **Fichiers de vérification VirusTotal**
    - `executable.exe` : la version finale (binaire Windows) produite par `build.sh`.
    - `screenshot-virustotal.png` : capture d'écran du **résultat** d'analyse sur VirusTotal pour **cet** exécutable précis.
    - `virustotal.txt` : contenant **le lien** du rapport en ligne.
    - **Le `sha256sum`** de `executable.exe` : pour prouver que c'est **la même version** que celle soumise en ligne.

Important : Toute **discordance** entre le binaire fourni et les preuves d'analyse (hash, capture) sera pénalisée.

8. Notation et examen final

8.1 Critères de notation du projet

1. **Qualité du rapport** :
    - Clarté, organisation, **exactitude** des informations (attention aux “inventions” hasardeuses).
2. **Pertinence et efficacité des techniques employées** :
    - Modularité, furtivité, cohérence globale.
3. **Organisation du code et facilité de compilation** :
    - Structure claire dans `./proj`, script `build.sh` fonctionnel, liberté vis-à-vis de dépendances lourdes.
4. **Maintenabilité à long terme** :
    - Possibilité d’adapter facilement votre code en cas de détection (composition en briques indépendantes, etc.).
5. **Compréhension approfondie** :
    - Montrez que vous **savez expliquer** vos choix techniques (résolution d’API par PEB, injection en mémoire, hooking, etc.).

8.2 Évaluation papier individuelle en fin de module

- Une **grosse évaluation écrite et sans Internet** aura lieu **à la fin** de ce module.
- **Attention** : le projet est **individuel** et vous devrez prouver votre compréhension complète des techniques que vous avez implémentées.
- **Double pénalité** si :
    - Vous rendez un projet très complexe / très furtif.
    - Et, lors de l’examen, vous **n’êtes pas capable** d’expliquer précisément vos mécanismes.
    - Plus vous prétendez maîtriser de techniques avancées, plus vous serez évalués sur leur compréhension.

Exemple concret :

- Vous livrez un malware avec un stub PEB-based très élaboré, un encodeur maison, diverses checks anti-EDR, etc.
- À l’examen, vous êtes incapables d’expliquer le fonctionnement des hashs, la convention d’appel, ou la façon dont vous déchiffrez vos chaînes.
- Résultat : **pénalité plus forte** que si votre projet était moins ambitieux, mais cohérent avec vos réelles connaissances.

9. Synthèse et conseils finaux

1. **Cohérence entre votre projet et votre niveau** : Viser très haut est valorisé seulement si vous maîtrisez vraiment ce que vous faites.
2. **Testez en environnement local** (VM Windows + antivirus) **avant** les soumissions en ligne.
3. **N'abusez pas** des plateformes publiques (VirusTotal, etc.) pour éviter de les "entraîner" contre votre code.
4. **Relisez votre rapport** : aucune **erreur technique** ne doit y subsister, sous peine de sanction.
5. **Exploitez les connaissances** des modules passés (reverse, exploitation binaire, hooking).
6. **Organisez votre code en briques** et validez-les progressivement.
7. **Un LLM peut être un assistant, pas une béquille. Si je tombe sur une réponse hallucinée, considérez que votre note sera tout aussi imaginaire.**

Bon courage à tous !

Vous avez désormais **tous les détails** pour mettre en pratique les techniques d'évasion dans un **projet complet** et modulaire. Prenez votre temps pour **tester chaque brique**, analyser les détections éventuelles et progresser **par itérations**. Le but n'est pas seulement de fournir un binaire furtif, mais de **comprendre** en profondeur comment fonctionnent les systèmes de protection, les API Windows et la résolution dynamique, ainsi que l'impact des différentes stratégies d'injection mémoire.

Rendez-vous en fin de module pour l'évaluation écrite, où vous devrez prouver votre **compréhension** de votre propre projet. Soyez rigoureux, testez méthodiquement et n'oubliez pas : un code parfaitement FUD mais mal compris ne vous fera pas obtenir les points que vous espérez.