Travaux Pratiques : Lab – Pentest avec Python

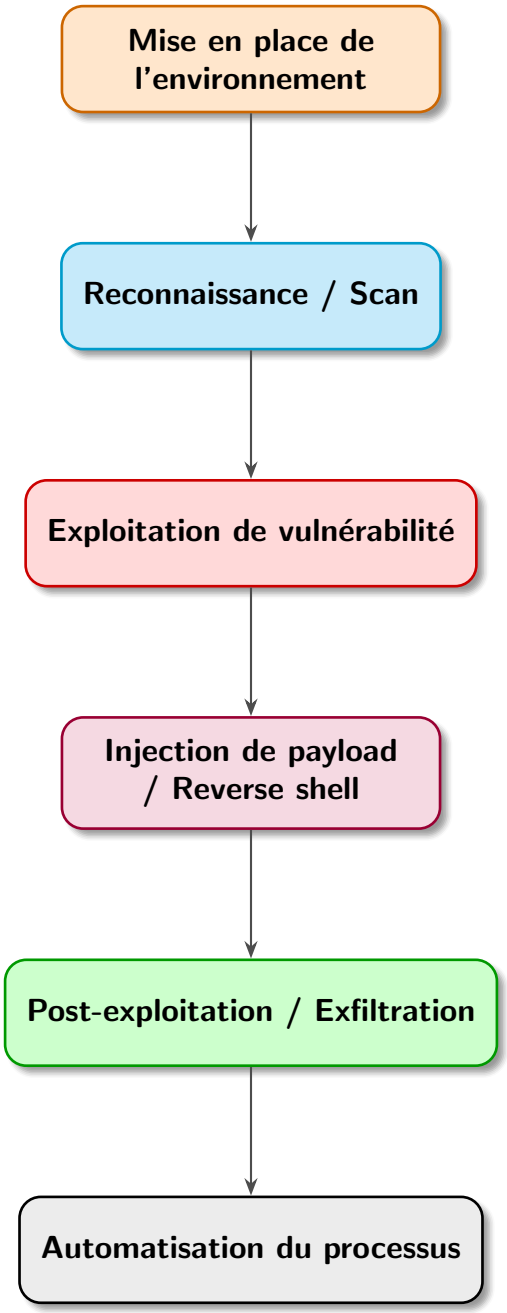
M1-SI – Module : Sécurité avec Python

Date : 09 Juillet 2025

Objectifs

L'objectif de ce TP est de concevoir un outil Python capable d'automatiser les différentes étapes d'une attaque de type *pentest* sur une machine vulnérable. L'ensemble du cycle d'attaque est abordé, depuis la reconnaissance initiale jusqu'à la post-exploitation.

Étapes du pentest :



Pré-requis

Avant de commencer, vous devez disposer de l'environnement et des connaissances suivants :

Machines virtuelles

- Une machine virtuelle **Kali Linux** comme machine **attaquante**
- Une machine virtuelle **Metasploitable 2** comme machine **cible vulnérable**
- Les deux machines doivent être configurées sur le **même réseau local** (mode `Host-Only` ou `Bridged`)

Environnement Python

- **Python 3** installé sur Kali
- `pip` pour installer les bibliothèques
- Création d'un environnement virtuel : `python3 -m venv venv`
- Bibliothèques à installer :
  - `requests`
  - `beautifulsoup4`
  - `paramiko`
  - `pwntools`

Compétences recommandées

- Bases en **Python** (fonctions, modules, sockets)
- Compréhension du fonctionnement réseau (IP, ports, TCP, HTTP)
- Notions de base en sécurité informatique et systèmes Linux
- Capacité à utiliser un terminal sous Linux et modifier des scripts

1 Étapes du projet

1.1 Structure du projet Python

Avant de développer les fonctionnalités de reconnaissance, d'exploitation ou de post-exploitation, il est important de structurer correctement le projet Python.

Arborescence du projet

```
redfox/
      main.py    (point d'entrée principal)
      scanner.py    (module de reconnaissance)
      exploit.py    (module d'exploitation)
      payloads.py    (reverse shell, stagers)
      post_exploit.py    (exfiltration, persistance)
      config.py    (cible, IPs, ports)
      utils.py    (fonctions partagées)
      requirements.txt    (bibliothèques Python à installer)
```

Instructions :

- Créez un dossier `redfox` dans votre répertoire utilisateur Kali
- Initialisez un environnement virtuel Python :
  - `python3 -m venv venv`
  - `source venv/bin/activate`
- Installez les bibliothèques :
  - `requests`, `beautifulsoup4`, `paramiko`, `pwntools`
- Utilisez `main.py` pour appeler les autres modules dans le bon ordre

Phase 3 – Exploitation de vulnérabilité

L'objectif est de comprendre comment exploiter une vulnérabilité détectée lors de la phase de reconnaissance dans le but d'exécuter du code arbitraire sur la machine cible.

À cette étape, on cible une faille spécifique présente sur l'interface web ou le service identifié précédemment. L'exemple utilisé ici est une **injection de commande système** (`Command Injection`) permettant d'exécuter des commandes shell sur le serveur.

L'exploitation consiste à :

- Repérer une URL vulnérable qui accepte des paramètres insérés dans une commande système (ex : `ping?host=127.0.0.1`)
- Injecter une commande supplémentaire à l'aide de séparateurs (ex : `127.0.0.1; whoami`)
- Observer si la sortie de la commande est renvoyée dans la réponse HTTP
- En cas de succès, enchaîner vers un payload plus avancé (ex : reverse shell)

Pré-requis :

- Une URL identifiée comme vulnérable depuis la phase 2 (par exemple `/dvwa/vulnerabilities/exec/`)

- Connaissance des séparateurs de commande sous Linux : `;`, `&&`, `|`
- Familiarité avec les requêtes HTTP GET et la structure d'URL

Instructions – Partie Exploitation

1. Créez un fichier `exploit.py` dans votre dossier de projet.
2. Implémentez un script Python qui :
   - Envoie une requête HTTP GET à une URL vulnérable
   - Injecte une commande (ex : `whoami`, `id`) via un paramètre GET
   - Affiche la réponse du serveur (code HTML ou sortie de la commande)
3. Testez votre script sur une URL vulnérable comme :
   - `http://192.168.X.X/dvwa/vulnerabilities/exec/?ip=127.0.0.1`
4. Vérifiez que la commande est bien exécutée côté serveur et que la sortie est visible dans la réponse HTML.

Phase 4 – Injection de Payload / Reverse Shell

Lorsqu'une vulnérabilité (comme une injection de commande) permet d'exécuter des commandes système, on peut injecter un **payload Python** ou Bash qui ouvre une connexion inverse ("reverse shell") :

- La machine cible (`Metasploitable`) lance une connexion vers la machine attaquante (`Kali`)
- L'attaquant écoute avec `netcat` sur un port défini (ex : 4444)
- Une fois connecté, il obtient un shell interactif à distance

Ce processus permet d'obtenir un accès direct sur le système distant et de poursuivre les actions en post-exploitation.

Pourquoi un reverse shell ?

- Permet une prise de contrôle à distance en mode interactif
- Facilite l'exécution de commandes complexes
- Ouvre la voie à la persistance, à l'élévation de privilèges ou au pivot réseau

Instructions – Partie Reverse Shell

1. Créez un fichier `payloads.py` dans votre dossier de projet.
2. Implémentez une fonction Python qui contient un payload de reverse shell :
   - Il doit se connecter à votre machine Kali (IP, port)
   - Il doit rediriger les entrées/sorties vers un shell Bash
3. Sur Kali, ouvrez un terminal et démarrez un écouteur Netcat :
   - `nc -lvnp 4444`
4. Injectez le payload via le script d'exploitation développé à l'étape précédente (ex : via `command injection`).
5. Vérifiez que vous obtenez une connexion entrante depuis Metasploitable.

Phase 5 – Post-Exploitation / Exfiltration

Une fois un shell obtenu sur la cible, l'attaquant peut :

- Explorer les fichiers du système (ex : `/etc/passwd`, fichiers de configuration, logs)
- Récupérer des données sensibles via des outils comme `SFTP`, `scp` ou un script personnalisé
- Installer une clé SSH pour persistance
- Créer un utilisateur malveillant

Dans ce TP, on se concentre sur une tâche typique : l'**exfiltration d'un fichier sensible** depuis la cible vers Kali, à l'aide du protocole SSH et de la bibliothèque Python `paramiko`.

Pourquoi cette étape est critique ?

- Elle simule ce que ferait un attaquant après avoir compromis un système
- Elle permet de démontrer l'impact réel de l'exploitation (preuve de compromission)
- Elle prépare les étapes futures : persistance, pivot, élévation de privilège

Instructions – Partie Post-Exploitation

1. Créez un fichier `post_exploit.py` dans votre projet.
2. Implémentez un script Python utilisant `paramiko` qui :
   - Se connecte en SSH à la machine cible (ex : `192.168.X.X`, utilisateur `msfadmin`)
   - Récupère un fichier local (ex : `/etc/passwd`) via SFTP
   - Le sauvegarde sur votre machine Kali dans un fichier local
3. Testez la connexion et la récupération de fichier.

Phase 6 – Automatisation du processus

Chaque module Python précédemment développé (scanner, exploit, payload, post_exploit) sera intégré dans un fichier central `main.py`, qui orchestre l'ensemble de la chaîne d'attaque.

L'exécution se fait de manière séquentielle ou interactive :

- Appel des fonctions de reconnaissance et extraction des informations
- Ciblage automatique ou manuel d'un vecteur vulnérable
- Injection d'un payload (reverse shell)
- Exécution d'actions post-exploitation

Cela permet de :

- Gagner du temps lors d'un audit répétitif
- Documenter et structurer la chaîne d'exploitation

Instructions – Partie Automatisation

1. Créez un fichier `main.py` à la racine du projet.
2. Importez tous vos modules précédents :
   - `scanner`, `exploit`, `payloads`, `post_exploit`
3. Organisez le déroulement du script :
   - Étape 1 : reconnaissance
   - Étape 2 : exploitation
   - Étape 3 : reverse shell (optionnel)
   - Étape 4 : exfiltration
4. Ajoutez des impressions claires dans le terminal (`[INFO]`, `[OK]`, `[ERREUR]`)
5. **p**roposez un menu interactif pour choisir quelles phases exécuter

Conclusion

Ce projet a permis de mettre en œuvre l'ensemble du cycle d'un test d'intrusion automatisé sur une cible vulnérable. En utilisant uniquement Python et des bibliothèques spécialisées, ce projet vous a permis de :

- Créer un environnement de test (Kali + Metasploitable)
- Concevoir un outil modulaire pour automatiser les étapes classiques d'un pentest :
  - **Reconnaissance**
  - **Exploitation de vulnérabilité**
  - **Injection de payload**
  - **Post-exploitation**
- Construire un projet Python structuré, documenté et réutilisable