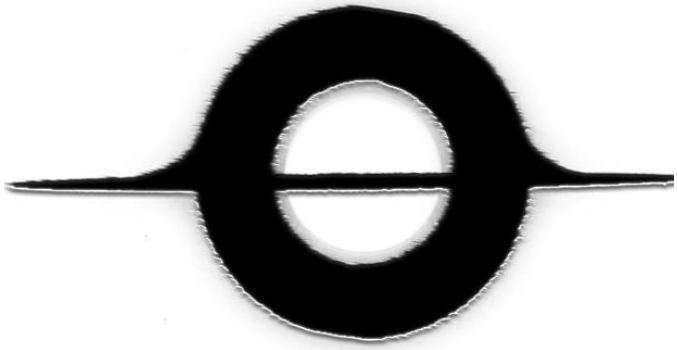
Blackhole
Pentest Toolbox



Documentation

MSI-CSA
Projet de fin d'année
FEVRE Tony – MULLER Florian

Sommaire

[1]

[1]

Présentation et objectif

Blackhole est un outil de test d'intrusion accompagnant les phases de reconnaissances actives et permettant de réaliser des attaques basiques afin de réaliser des économies de temps. Les informations obtenues sont ajoutées dans un rapport au format HTML.

Blackhole n'a besoin que d'une cible pour commencer à travailler. Des options au lancement sont disponibles afin paramétrer votre reconnaissance à votre besoin.

Installation

Python

Si ce n'est pas déjà le cas, installez l'interpréteur Python : https://www.python.org/downloads/

Blackhole a été développé sous Python **3.8.3 .**

Nmap

Nmap doit être installé dans votre environnement : https://nmap.org/download.html

Télécharger et installer la version correspondant à votre environnement.

Chrome webdriver

La recherche d'exploits utilise la librairie Selenium et Chromewebdriver. Ce dernier doit être téléchargé et ajouter la variable PATH de votre système, la manière la plus rapide est d'ajouter le chromewebdriver.exe dans %system32%.

Lien du téléchargement, faites correspondre avec votre version de Google Chrome.
https://chromedriver.chromium.org/downloads

La recherche d'exploit ne peut pas fonctionner sans Chrome.

Dépendances

Toutes les dépendances de librairies peuvent être installées à l'aide de ***pip*** et du fichier "requirements.txt" fourni. Exécutez la commande suivante dans un terminal (placé au préalable dans le bon répertoire) afin d'installer les dépendances avec les bonnes versions.

pip install -r requirements.txt

Utilisation

Seul un fichier est prévu pour l'interface utilisateur : ***main.py .***

Vous devez appeler ce fichier avec l'interpréteur Python et lui donner les arguments de ligne de commande dont vous avez besoin. La position des arguments n'a pas d'importance.

--help, -h (défaut)

S'affiche si l'on précise le flag ou si aucun argument n'est donné.

Affichage de l'aide comme ci-après :

```
usage: main.py [-h] [-p PORTS] [-j] [-b] [-s] [-v] target

Blackhole : Pentest Toolbox

positional arguments:
  target                IP address or domain to scan.

optional arguments:
  -h, --help            show this help message and exit
  -p PORTS, --ports PORTS
                        Take a range of ports to scan. (like 1-65535)
  -j, --json            Output result to blackhole_report.json file.
  -b, --bruteforce      Enable Bruteforce if applicable to the target. You will be prompt to enter some additionnals
                        info.
  -s, --subdomains      Enumerate subdomains of given target. Must be a domain name.
  -v, --verbose         Increase output verbosity.
```

Target (nécessaire)

Adresse de la cible à attaquer.

Il s'agît soit d'un nom de domaine ou d'une adresse IP. Pas de flag, renseigner simplement l'adresse en argument.

```
\blackhole_v1\main.py 172.16.80.128
```

-p, --ports [1-65535] (flag + valeur)

Par défaut, le scanner se contente d'énumérer les ports les plus utilisés. Si on souhaite énumérer davantage de ports, on peut préciser une étendue (de 1 à 65 535) de ports qui seront scanner par ***Blackhole***.

Ce paramètre nécessite la présence du flag **–p** ou –**ports** accompagner **obligatoirement** d'une valeur, sans quoi une erreur sera retournée.

```
\main.py -p 1-200 gruyere.corp
```

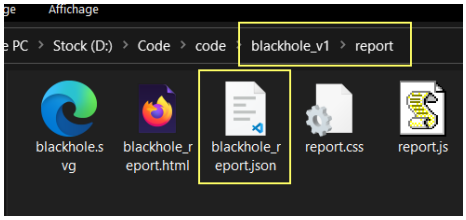
-j, --json (flag)

Active la sortie vers un fichier JSON dans le répertoire ***report***.

L'ensemble des données collectés par le programme sont stockés dans un dictionnaire Python, l'activation de ce flag permet d'écrire le contenu de ce dictionnaire dans un fichier au format JSON.

Utile pour réaliser du post-traitement.

```
\main.py -j gruyere.corp
```

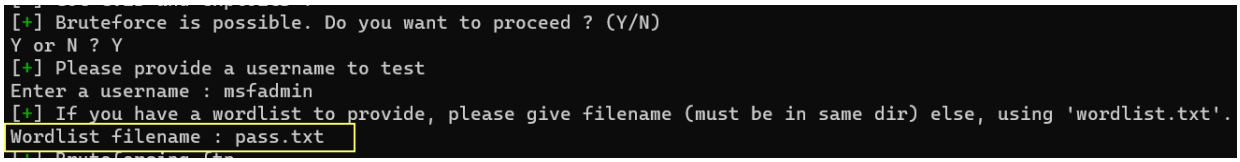


-b, --bruteforce (flag)

Active les fonctions de bruteforce.

Blackhole peut réaliser des attaques par dictionnaire **s'il détecte un service FTP ou SSH**, peu importe le port, si aucun service n'est détecté le bruteforce sera automatiquement désactivé (même si le flag est actif).

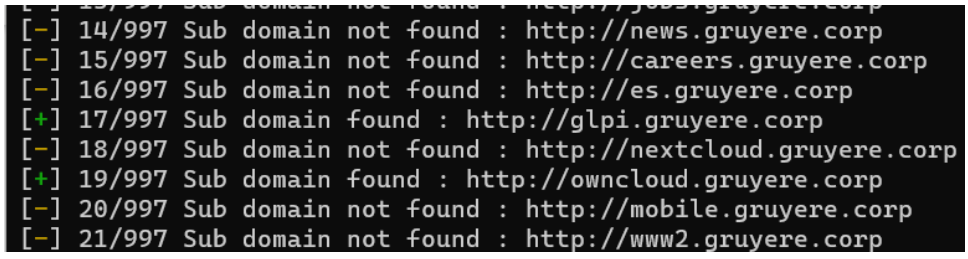
Le programme est livré avec une wordlist par défaut qui sera utilisé si aucune n'est précisé durant l'exécution. Si vous souhaitez rajouter des wordlists, vous devez les ajouter dans le dossier de ***Blackhole*** et donner le nom du fichier au lancement de l'outil de bruteforce.



-s, --subdomains (flag)

Active la fonction d'énumération des sous-domaines. Cette fonction s'active **uniquement si la cible est un nom de domaine**, dans le cas contraire le module renvoie une erreur et le programme continue le reste des opérations.

Cette option charge le fichier "subdomains.txt" et va réaliser des requêtes sur la cible en ajoutant chaque préfixe contenu dans le fichier à la cible et vérifier si une réponse est obtenue, si oui le sous-domaine est actif, il est ajouté à la liste des sous-domaines actif.



-v, --verbose (flag)

Augmente le niveau de détail des messages imprimés dans la console.

Quand ce paramètre est actif, des messages supplémentaires sont affichés dans la console pour tous les modules. Elle permet d'avoir une meilleure visibilité sur l'action que l'on mène mais au détriment de la lisibilité du terminal.

Code review

Main.py

C'est le point central du programme, c'est là que les différents modules s'intègre et sont appelés.

On import toutes nos librairies mais également nos modules.

```
ain.py > ...
    # Built-in #
    import argparse
    import json
    import re
    import os
    #############

    # Core #
    from cve import get_CVE
    from scan import get_services
    from report import generate_report
    from colors import info, error
    from subdomains import get_subdomains
    import searchsploit as s
    from bruteforce import bruteforce
    ########
```

On définit tous nos arguments, s'il sont présents, on créer des variables constantes en leur nom.

```
# Target
args_parser.add_argument(
    'target',
    help="IP address or domain to scan.",
)

# Ports range
args_parser.add_argument(
    '-p', '--ports',
    default=False,
    help="Take a range of ports to scan. (like 1-65535)"
)

# JSON output
args_parser.add_argument(
    '-j', '--json',
    action='store_true',
    default=False,
    help="Output result to blackhole_report.json file."
)

# Bruteforce
args_parser.add_argument(
    '-b', '--bruteforce',
    action='store_true',
    default=False,
    help="Enable Bruteforce if applicable to the target. Y
)
```

```
# args and flags #
TARGET = args.target
JSON_OUTPUT = args.json
VERBOSE = args.verbose
SUBDOMAINS = args.subdomains
BRUTEFORCE = args.bruteforce
PORTS_RANGE = args.ports
##################
```

Si les informations rentrées ne sont pas celle attendu, le retour help affiche la bannière d'aide.

On démarre le scan et on récupère les services souhaités. Voir la partie sur **scan.py** pour plus de détails.

```
services = get_services(TARGET, PORTS_RANGE, VERBOSE)
info('NMAP finished !')
```

On itère dans la liste des services retournées par get_services(). Afin de savoir si le bruteforce est possible, on vérifie la présence des services FTP et SSH.

```
for service in services:
    if service['name'] == 'ssh':
        bruteforce_services.append(['ssh', service['port']])
    elif service['name'] == 'ftp':
        bruteforce_services.append(['ftp', service['port']])
    else:
        SSH_PORT = False
        FTP_PORT = False
```

Pour chaque service, on fait une recherche de CVEs avec la fonction get_CVE (voir partie dédiée plus bas). Une fois les CVEs récupérés, on enchaîne avec la recherche d'exploits applicable à l'aide de **searchsploit_by_cve().** Enfin, on agrège les informations dans la liste services.

```
    service['cve'] = list()
    # Match services and applicable CVEs.
    cves = get_CVE(service)
    if not cves == 'Not enough enumeration info':
        for cve in cves:

            # Search for available exploits on EDB
            exploits = s.searchsploit_by_cve(cve['id'], VERBOSE)
            cve['exploits'] = exploits

            service['cve'].append(cve)

info("Got CVEs and exploits !")
```

Si le flag BRUTEFORCE est actif et s'il existe au moins un élément dans la liste **bruteforce_services,** alors on démarre le phase bruteforce. On commence par demander confirmation à l'utilisateur, puis on demande un nom d'utilisateur à utiliser pour les tentatives de connexions et s'il souhaite utiliser une wordlist différente de celle par défaut (wordlist.txt).

On stock les connexions réussites dans la variable **credentials.**

On ajoute nos dernières informations à la liste **services** :

- La cible en début de liste
- Un item dictionnaire supplémentaire pour le bruteforce
- Un item dictionnaire supplémentaire pour les subdomains (si applicable)

Toutes nos données sont stockées dans **services,** on peut maintenant générer le rapport avec **generate_report()**. Une fois terminée il ne nous reste plus qu'à ouvrir le fichier **report/blackhole_report.html**

Si le flag JSON_OUTPUT est actif, la variable service est également dump dans un fichier JSON situé dans le répertoire **report/ .**

```python
# Add TARGET to first index of list
services.insert(0, TARGET)

# Add found credentials
services.append({"bruteforce": credentials})

# Add subdomains enumeration (if exist)
if SUBDOMAINS:
    if subdomains_list:
        services.append({"subdomains": subdomains_list})
    else:
        services.append({"subdomains": 'No subdomains enumeration'})

info('Creating report...')
generate_report(services)

if JSON_OUTPUT:
    json_path = FILE_DIR + "report/blackhole_report.json"

    with open(json_path, "w") as f:
        json.dump(services, f, indent=4)
        info("Output to JSON file !")
```

Scan.py

Réalise un scan nmap et extrait les données nécessaires aux traitements suivant.

On utilise la librairie **python3-nmap** qui permet tout simplement de manipuler le nmap installé sur notre poste depuis Python.

Homepage : https://pypi.org/project/python3-nmap/

On commence par créer les arguments à passer à la commande nmap, si l'utilisateur à utiliser le flag –**p** alors on ajoute la liste des ports aux arguments. Le reste des arguments reste inchangé car nmap doit être en mesure d'énumérer les services détecter (**-sV**), le timing (**-T4**) n'a pas besoin d'être changé.

```python
def get_services(target, ports_range, verbose):
    nmap = Nmap()
    if ports_range:
        # Scan range
        nmap_args = '-T4 -sV -p ' + ports_range
    else:
        # Scan top ports
        nmap_args = '-T4 -sV'
```

On démarre le scan.

```
    nmap_args = '-T4 -sV'
# nmap scan
colors.info(f"Scanning {target} ...")
output = nmap.scan_top_ports(target, args=nmap_args)
```

Dans le cas où la cible est un nom de domaine, la sortie de nmap renverra l'adresse IP résolu depuis le nom. Cela pose problème pour les données existantes, c'est pourquoi on modifie la valeur de **target** par la valeur de la première clé de la liste sortie du scan nmap.

```
# Loading the first key of output dictionnary to get data
target = list(output.keys())[0]
```

On récupère les informations intéressantes dans une nouvelle liste **services**, puis on retourne cette dernière.

```
for port in output[target]["ports"]:
    services.append({})

    for info in service_info:
        try:
            services[index][info] = port["service"][info]
            # Get port ID
            services[index]['port'] = port['portid']
        except KeyError:
            if verbose:
                colors.warning(f"{port['service']['name']} : No results about {info}")
            else:
                pass

    index += 1

return services
```

Cve.py

Utilise la librairie officielle de nvd.nist.gov : **nvdlib**

Son gros avantage est qu'elle exploite l'API NVD.NIST qui est publique, pas de besoin de clé API.

La fonction commence avec une vérification, si le service n'a pas de clé **'produit'** associée, alors on ne dispose d'assez d'informations pour réaliser une recherche.

```
def get_CVE(service):
    if 'product' in service:
        CVEs = list()
```

Ensuite, on vérifie si la clé **'version'** existe, si oui on combine les clés product et version afin de créer un mot clé pour la recherche. On utilise également une expression régulière afin de récupérer un

format pour ‘version’ qui soit standard afin que la recherche aboutisse correctement.

```
# Build keyword for NVDLIB request
if 'version' in service:
    # Extracting a 'correct' version pattern for keyword search
    r_version = re.match(r"(\d+\.)+\w{0,4}", service['version'])
    service['version'] = r_version.group()

    keyword = service["product"] + ' ' + service['version']
else:
    keyword = service["product"]
```

On lance la recherche. On collecte 3 infos dans l’objet qui est retourné : id, score, url. Enfin, on trie les CVEs par score.

```
    # Makes a NVD NIST API call and return an object
    cve_list = nvdlib.searchCVE(keyword=keyword)

    # Build a list of CVEs. Each item is a dictionnary
    index = 0
    for cve in cve_list:
        CVEs.append(dict())

        CVEs[index]["id"] = cve.id
        CVEs[index]["score"] = cve.score
        CVEs[index]["url"] = cve.url

        index += 1

    # Sort by score
    CVEs.sort(key=lambda item: item['score'][1], reverse=True)

    return CVEs
else:
```

Searchsploit.py

Scrapper pour exploit-db.com

La fonction searchsploit_by_cve(), en se basant sur la liste des CVEs récupérés avant, va chercher des exploits correspondant à chaque CVE.

L’outil linux **searchsploit** est un script bash. On ne peut l’utiliser en l’état, surtout sur windows. C’est pourquoi nous avons développé un scrapper web avec Selenium et Chromewebdriver.

On peut réaliser des requêtes dans l’URL comme avec le paramètre ‘**cve’**, en itérant dans a liste des CVEs on peut donc obtenir tous les exploits référencés pour chaque CVE.

```python
from colors import info, warning
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.chrome.options import Options

def searchsploit_by_cve(cve, verbose):
    chrome_options = Options()
    chrome_options.add_argument("--headless")
    chrome_options.add_argument("--disable-gpu")
    chrome_options.add_experimental_option('excludeSwitches', ['enable-logging'])
    driver = webdriver.Chrome(options=chrome_options)

    url = f"https://www.exploit-db.com/search?cve={cve}"
    search_result_xpath = "//table[@id='exploits-table']/tbody/tr"

    if verbose:
        info(f"Looking for {cve}'s exploits...")

    driver.get(url)
```

On attend que la recherche est terminé en attendant l'apparition d'une balise **<tr>** dans le tableau **'exploits-table'.**

```python
# Waiting for search results to appear
WebDriverWait(driver, 40).until(
    EC.presence_of_element_located((By.XPATH, search_result_xpath))
)

# Result of the research
results_row = driver.find_elements(By.XPATH, search_result_xpath)
```

Il est possible que la recherche termine mais ne retourne aucun résultat. On doit alors vérifier la présence d'une balise portant le nom de **'dataTables_empty'**. On ajoute **'No data available'**, sinon on continue.

Ensuite, on va récupérer tous les éléments **<a>** contenu dans le résultat de recherche. Extraire juste l'attribut **href** de chaque élément dans une compréhension de liste. De cette manière, on a récupéré le lien de la page de l'exploit et le lien de téléchargement direct. On les extraits dans une liste **'exploits'.**

```
    # Look for the presence of class "dataTables_empty", returns a list. If t
    if driver.find_elements(By.CLASS_NAME, "dataTables_empty"):
        exploits.append("No data available")
        if verbose:
            warning("No data available")
    else:
        if verbose:
            info("Got them !")
        for row in results_row:

            # Select all <a> tags
            anchors = row.find_elements(By.TAG_NAME, "a")

            # Extract each value of href attributes using list comprehension
            links = [elem.get_attribute('href') for elem in anchors]

            exploits_links = {
                "exploitdb_link" : "",
                "download_link" : ""
                }

            for link in links:
                if 'download' in link:
                    exploits_links["download_link"] = link
                elif 'exploits' in link:
                    exploits_links["exploitdb_link"] = link

            exploits.append(exploits_links)
    driver.quit()
    return exploits
```

Subdomains.py

Enumère les sous-domaines en se basant sur une liste prédéfinie contenu dans le fichier **'subdomains.txt'**.

On charge le fichier et on initialise des compteurs (pour l'affichage).

```
# Open subdomains list
with open(FILE_DIR + "subdomains.txt") as f:
    sub_domains=f.read().splitlines()

domain_count=len(sub_domains)
count=1
```

Pour chaque préfixe, on va réaliser une requête vers préfixe + domain. On gère les différentes erreurs, les connexion réussis sont des domaines valides : on les ajoute à la liste des URLs.

```
    urls = list()
    info("Enumerating subdomains...")
    for sub_domain in sub_domains:
        url=f"http://{sub_domain}.{domain}"
        try:
            requests.get(url,timeout=2)
        except requests.exceptions.ReadTimeout:
            error('Request timed out')
        except requests.ConnectionError:
            if verbose:
                warning(f"{count}/{domain_count} Sub domain not found : {url}")
            pass
        except KeyboardInterrupt:
            error('Keyboard interruption')
            break
        else:
            # Append URL to list of URLs
            info(f"{count}/{domain_count} Sub domain found : {url}")
            urls.append(url)
        count += 1

    info('Enumeration finished')

    return urls
```

Bruteforce.py

Module de bruteforce. On y retrouve 3 fonctions : ftp_connect, ssh_connect et bruteforce.

Ssh_connect()

Utilise la librairie Paramiko : https://www.paramiko.org/

Réalise une tentative de connexion SSH avec les paramètres données en arguments.

On traite deux erreurs : paramiko.AuthenticationException et paramiko.SSHException.

La première apparait lors d'une erreur d'authentification, puisque l'on test des centaines de combinaison, il est normal d'avoir des erreurs comme celle-ci.
La seconde peut apparaître suite à de multiples causes, mais dans ce contexte, on s'attend à ce que le serveur SSH finisse par stopper les réponses s'il reçoit trop de connexions dans un laps de temps réduit. On attend 30 secondes afin que le serveur réponde de nouveaux aux requêtes.

Enfin, on renvoie les combinaisons **username:password** qui ont réussis et on les renvoie.

```python
def ssh_connect(client, target, ssh_port, username, password, verbose):

    if verbose:
        info(f'Trying {username}:{password}...')

    try:
        client.connect(hostname=target, port=ssh_port, username=username, password=password, timeout=2)
    except paramiko.AuthenticationException:
        # Common error as long as we are bruteforcing, so log is optional
        if verbose:
            warning(f"Failed for {username}:{password}")
        return False
    except paramiko.SSHException:
        # SSHException is a generic error, in this context it must means that we tried to many login in a short time window
        warning(f"Maybe too much attempts. Waiting 30 seconds...")
        time.sleep(30)
    else:
        # success
        info(f'''Found one !
        USERNAME: {username}
        PASSWORD: {password}''')
        return True
```

Ftp_connect()

Utilise la librairie built-in ftplib.

Cette fonction réalise la même opération que son homologue ssh_connect mais sur le service FTP.

Bruteforce()

Utilise les deux précédentes fonctions et va réaliser des attaques par dictionnaire pour tous les services passés en argument (**services**), le username est donné durant l'execution de Blackhole tandis que la wordlist est récupéré, par défaut, depuis un fichier wordlist.txt ou alors en argument lors de l'appel de la fonction.

Les clients sont initilisés au début de la fonction, cela évite de créer et fermer sans arrêt des sesssions. Ici, un service = une session. Si des credentials valident sont trouvés, la liste est retournée.

```python
def bruteforce(services, target, username, verbose, passlist='wordlist.txt'):

    # Read each line of password list
    passlist = open(FILE_DIR + passlist).read().splitlines()

    credentials = list()

    for service in services:
        service_name = service[0]
        service_port = int(service[1])

        info(f'Bruteforcing {service_name}')
        # FTP client
        if service_name == 'ftp':
            ftp_client = ftplib.FTP()
            ftp_port = service_port
        else:
            ftp_client = False

        # SSH client
        if service_name == 'ssh':
            ssh_client = paramiko.SSHClient()
            ssh_port = service_port

            # Pass host key verification by adding it to known hosts
            ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        else:
            ssh_client = False

        for password in passlist:
            # Add login/password to credentials list for each combination that *_connect functions returns True value.
            if ssh_client:
                if ssh_connect(ssh_client, target, ssh_port, username, password, verbose):
                    credentials.append(f"ssh : {username}:{password}@{target}:{service_port}")

            if ftp_client:
                if ftp_connect(ftp_client, target, ftp_port, username, password, verbose):
                    credentials.append(f"ftp : {username}:{password}@{target}:{service_port}")
    return credentials
```

Colors.py

Permet de définir grâce à la librairie **Colorama** les messages avec les bons code couleur à savoir :

Info [+] Message

Warning [-] Message

Error [!] Message

Ces 3 informations sont définies afin d'être appelé dans chacun des scripts. On voit que seul le symbole au milieu des crochet est colorisé. On peut le voir car il est englobé par les fonctions **Fore.COULEUR** (ajout d'une couleur) et **Fore.RESET** (réinitialise les couleurs).

```
from colorama import init, Fore
# Init colorama
init()

# Log messages
def info(message):
    symbol = '[' + (Fore.GREEN + '+' + Fore.RESET) + '] '

    print(symbol + message)

def warning(message):
    symbol = '[' + (Fore.YELLOW + '-' + Fore.RESET) + '] '

    print(symbol + message)

def error(message):
    symbol = '[' + (Fore.RED + '!' + Fore.RESET) + '] '

    print(symbol + message)
```

Voici un exemple avec le script "Subdomains.py" :

```
[-]351/484700 Sub domain not found :http://donate.google.fr
[-]352/484700 Sub domain not found :http://ph.google.fr
[-]353/484700 Sub domain not found :http://downloads.google.fr
[-]354/484700 Sub domain not found :http://public.google.fr
[-]355/484700 Sub domain not found :http://cart.google.fr
[-]356/484700 Sub domain not found :http://legacy.google.fr
[+]357/484700 Sub domain found :http://hotels.google.fr
[-]358/484700 Sub domain not found :http://www.law.google.fr
[-]359/484700 Sub domain not found :http://feedback.google.fr
[!]Keyboard interruption
```

Report.py

C'est le module en charge de la génération d'un rapport HTML afin de regrouper les données de l'outil dans une interface.

La librairie Airium

Homepage : https://gitlab.com/kamichal/airium

Il s'agît d'un "convertisseur bidirectionnel HTML/Python".

Dans notre cas, il permet de générer dynamiquement du code HTML en permettant de :

- Respecter l'arborescence XML et avoir un code final propre
- Pas besoin de template
- Gérer les balises HTML "à la manière Python" sans avoir à écrire directement du code HTML.
- Injecter des données facilement
- Ne pas avoir à jouer avec des chaines de caractère extrêmement longue et difficile à débugger.

Fonction generate_report()

Cette fonction génère une page HTML et y intègre les données du scan.

Avec l'utilisation des context handler (with) on retrouve une indentation similaire au HTML et on peut ainsi construire sa page HTML sans se soucier des détails. On ouvre les balises, on ajoute des attributs si nécessaires et les balises sont ajoutés à l'objet Airium "page" que l'on peut ensuite écrire dans un fichier.

```
def generate_report(s
    page = Airium()
```

```
page('<!DOCTYPE html>')

with page.html(lang='en'):
    with page.head():
        page.link(rel="icon", type="image/svg", href="blackhole.svg")
        page.link(rel="stylesheet", href='./report.css')
        page.link(rel="stylesheet", href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css")
        page.meta(charset="utf-8")
        with page.title():
            page("Blackhole report")
    # HTML Body Start
    with page.body():
        with page.div(klass='header'):
            with page.h1(klass='main-title'):
                page('Blackhole Report')
```

La capture ci-dessus montre la balise **<head>** avec tous les attributs nécessaires. On reconnait des similitudes avec le HTML. Nous avons ajouté un fichier de style et une icône obtenue sur ***font-awesome.***

Les données des services scannés sont contenues dans une liste composée de différents types de données. On peut donc itérer à travers cette liste, créer des tableaux HTML, récupérer les données et ajouter autant de ligne et de colonnes que l'on a de données dans la liste. Exemple ci-après avec la section "**Service overview".**

On créer les conteneurs, les titres, on définit les classes.

```
## Service overview
with page.div(klass='section'):
  with page.div(klass='section-title'):
    page.h3(_t='Services Overview')
    page.i(onclick='toggleIcon(this)', klass='fa fa-eye-slash')
  with page.div(klass='section-content'):
```

Ensuite on créer un tableau HTML pour les services, on créer les entêtes de colonnes décrivant le contenu de chacune puis on itère au travers de la liste pour remplir le tableau, on créer une nouvelle ligne et 4 nouveaux éléments à chaque tour de boucle.

Les méthodes .get() permettent d'obtenir les valeurs des clés données en premier argument. Si la clé n'existe pas, renvoie une chaine vide.

```
with page.table(id="services-overview", klass='table'):
  with page.thead(klass='table-head'):
    page.th(_t='Name')
    page.th(_t='Product')
    page.th(_t='Version')
    page.th(_t='Port')
  with page.tbody():
    for service in services[1:-2]:
      try:
        # Check if keys are present, else write empty string
        name = service.get("name","")
        product = service.get("product","")
        version = service.get("version","")
        port = service.get("port","")
        with page.tr():
          page.td(_t=name)
          page.td(_t=product)
          page.td(_t=version)
          page.td(_t=port)
      except AttributeError:
        pass
```

On applique exactement le même principe avec des données différentes pour les autres sections du rapport puis on écrit le contenu de **"page"** dans un fichier **blackhole_report.html** (il faut écrire en mode 'bytes').

```
  page.script(src="report.js")
  # HTML Body End #
report_path = FILE_DIR + 'report/blackhole_report.html'
with open(report_path, 'wb') as f:
  f.write(bytes(page))

# Testing
```

Lorsqu'on ouvre le fichier avec un navigateur, on constate que les données sont affichées correctement.

Target : gruyere.corp

Services Overview

Name	Product	Version	Port
ftp	vsftpd	2.3.4	21
ssh	OpenSSH	4.7p1	22
telnet	Linux telnetd		23
smtp	Postfix smtpd		25
http	Apache httpd	2.2.8	80
pop3			110
netbios-ssn	Samba smbd	3.X	139
https			443
netbios-ssn	Samba smbd	3.X	445
ms-wbt-server			3389

CVEs

vsftpd 2.3.4

CVE-2011-2523

NVD NIST Link

CVSS

CVSS Version	Score	Severity
V3	9.8	CRITICAL

Exploits

Page URL	Download link
49757	Download
17491	Download

CVE-2011-0762

NVD NIST Link

CVSS

CVSS Version	Score	Severity
V2	4.0	MEDIUM

Exploits

Page URL	Download link
16270	Download

OpenSSH 4.7p1

CVE-2008-5161

NVD NIST Link

CVSS

CVSS Version	Score	Severity
V2	2.6	LOW

Exploits