# A Script-Based Approach To Assessing Predictable Cookie Values In Web Applications

*Using Python and Arithmetic Encoding to develop a cookie value visualisation tool.*

**Ance Strazdina**

CMP320: Advanced Ethical Hacking

BSc (Hons) Ethical Hacking, Year 3

2022/23

*Note that Information contained in this document is for educational purposes.*

# Abstract

Ensuring that website cookie values are not predictable is crucial to the security of web applications depending on what role on the site the cookie exhibits. For example, easily guessable session ID values can pose significant risks such as the possibility of session hijacking which can lead to a threat actor impersonating a legitimate user and stealing sensitive information, among others, therefore the unpredictability of cookie values is imperative. At the same time, efficient identification of vulnerabilities during security assessments is vital to resolving these issues rapidly, which is why scripting, which aids with the automation of tasks, plays a crucial role in cybersecurity.

This project aimed to develop a script that visualises website cookie values over time to assess their predictability during web application penetration testing. To meet this aim, the OWASP WebScarab proxy, which features a similar non-script functionality, was investigated, and a Python script was developed in a virtual environment on a Linux-based system. It utilized libraries such as *requests* for making HTTP requests and retrieving cookies, as well as *mathplotlib* for plotting the cookie values. Additionally, it demonstrated the use of arithmetic encoding to compress the cookie values to a plottable format.

The project resulted in a successful implementation of a command-line tool capable of visualising the correlation between website cookie values and time. It consisted of several phases executed on runtime which were: making HTTP requests and retrieving cookies from a target website, using an implementation of arithmetic encoding to encode the cookie values, and plotting these to a scatter graph along with timestamps to showcase their relation. The generated output files were saved to a target-specific directory created on runtime and followed a naming convention of using the name of the specific cookie it related to. In addition, the handling of command-line arguments and performance considerations such as multiprocessing was implemented. Lastly, while the developed tool facilitates the identification of predictable cookie values, allowing for further threat recognition and mitigation, therefore contributing to web application security and outlining the efficiency of scripting, several improvements such as an enhanced implementation of arithmetic encoding and request rate-limiting, and further cookie analysis by the script have been outlined and discussed within this report.

# Contents


1 Introduction ..... 1

1.1 Background ..... 1

1.2 Aim ..... 4

2 Procedure ..... 5

2.1 Overview of Procedure ..... 5

2.1.1 Development Environment ..... 5

2.1.2 Used Libraries ..... 5

2.2 Retrieving Session IDs ..... 6

2.2.1 Authentication ..... 7

2.2.2 Output Directory ..... 7

2.3 Encoding ..... 8

2.3.1 Probability Distribution ..... 9

2.3.2 Arithmetic Encoding ..... 10

2.4 Visualisation ..... 11

2.5 Command-Line Interface ..... 11

2.5.1 User Input Error Handling ..... 12

2.6 Performance Considerations ..... 12

3 Discussion ..... 14

3.1 General Discussion ..... 14

3.2 Countermeasures ..... 16

3.3 Future Work ..... 17

References ..... 18

Appendices ..... 20

Appendix A – Source Code ..... 20

cookies.py ..... 20

# 1 INTRODUCTION

## 1.1 BACKGROUND

An important phase of web application penetration testing as defined by OWASP Web Security Testing Guide v4.2 is Session Management Testing (OWASP, 2020). Part of this involves testing the use of session tokens and cookies throughout the website. As both are used by websites to maintain a user's session state, their security is critical and an important aspect of this is ensuring that they are not predictable (Gregg & Santos, 2022). If session tokens or cookies are easily guessable, it can enable attacks such as session hijacking where attackers access a user's session by using their session ID (Figure 1-1) to take place. After hijacking a session an attacker can impersonate users, carry out fraudulent financial transactions, steal sensitive information, and perform other unauthorized actions on a website (Stuttard & Pinto, 2011), therefore it is important to ensure that the session IDs used by a web application are not predictable.

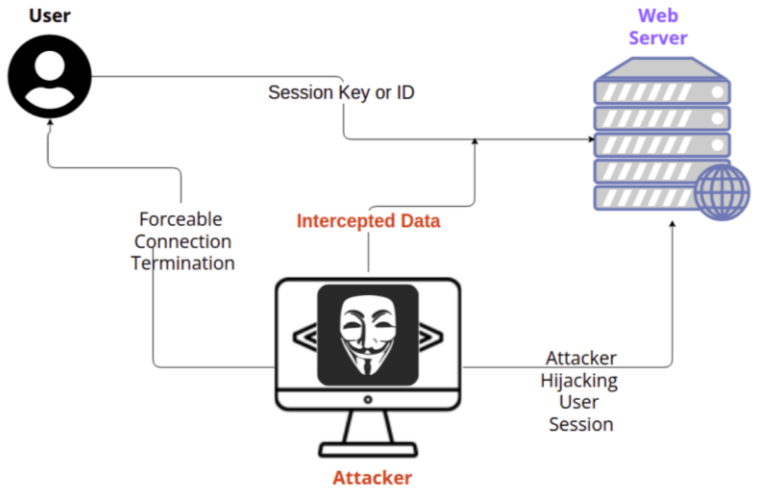


*Figure 1-1 A diagram of a session hijacking attack (Sunny Valley Networks, no date)*

A notable tool for examining cookies is WebScarab which is an open-source web proxy developed by OWASP. While other proxies such as Portswigger's BurpSuite and OWASP'S other proxy – ZAP, are frequently used within the industry (Whitehorn, 2020), WebScarab offers a distinct feature of analysing session identifiers and visualising them which allows testers to easily observe patterns in cookie values to determine whether they are easily predictable. The visualisation feature displays session identifier values over time in a scatter graph format (Figure 1-2 and Figure 1-3).

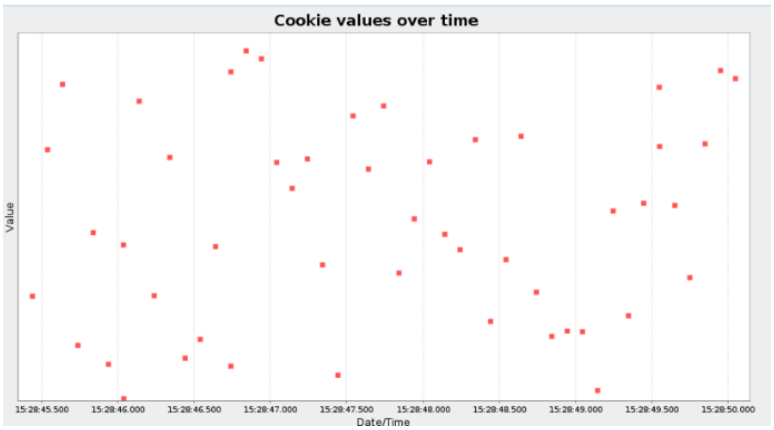


*Figure 1-2 Random cookie values visualised by WebScarab*

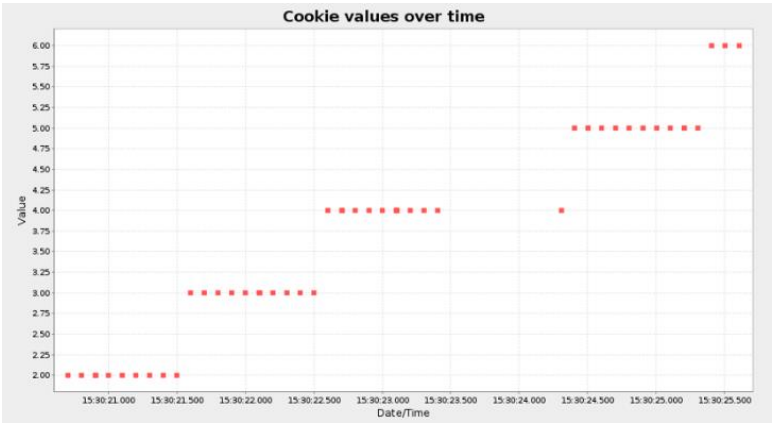


*Figure 1-3 Predictable cookie value visualised by WebScarab*

A graph that follows a logical sequence as opposed to a seemingly random pattern, indicates predictable cookie values. The logical pattern reveals that the same values and encoding are present in the cookie over time. Additionally, a horizontal line in a graph would mean that the cookie value does not depend on the time at all.

WebScarab's session ID visualisation functionality follows an outline of the user accessing the target website while this proxy is active and performing any actions, for example, authentication, that are needed for the site to set a cookie. The site responses which include the cookie being set are then examined in WebScarab. More values of the same cookie can then be obtained by repeating the request that prompted the cookie to be set. Lastly, the cookie values are plotted on a graph showing their dependence on the specific time they were assigned. An additional step between the obtaining of cookie values and plotting them is an encoding operation to obtain a plottable numeric value as many of them follow an encoding scheme, for example, Base64, or a combination of multiple schemes.

As WebScarab is open source, its code, written in Java, can be examined in the public GitHub repository: *OWASP/OWASP-WebScarab* (OWASP, 2020). Figure 1-4 contains a calculation function from the *OWASP-WebScarab/src/org/owasp/webscarab/plugin/sessionid/DefaultCalculator.java* file, which is used to

obtain a decimal representation of a cookie's value. It implements arithmetic encoding to obtain this value by performing operations on each character in the value string starting from the last character and moving towards the start. During every iteration it updates the encoded value, stored in the *total* variable, by encoding the current character. The *max* variable, which represents the range of possible values, is also updated by multiplying it by the character sets length.

```
public BigInteger calculate(SessionID id) {
    if (_cache.containsKey(id)) return _cache.get(id);
    String value = id.getValue();
    Matcher matcher = _pattern.matcher(value);
    if(matcher.matches() && matcher.groupCount()>=1) {
        value = matcher.group(1);
    } else {
        return BigInteger.ZERO;
    }
    BigInteger total = BigInteger.ZERO;
    BigInteger max = BigInteger.ONE;
    int length = value.length();
    _logger.fine("Calculating '" + value + "'");
    for (int i=0; i<length; i++) {
        String charset = _chars.get(i);
        char ch = value.charAt(length - 1 - i);
        int pos = charset.indexOf(ch);
        _logger.fine("Working on position " + i + ", character '" + ch + "'");
        _logger.fine("pos is " + pos);
        BigInteger val = new BigInteger(Integer.toString(pos));
        total = total.add(val.multiply(max));
        max = max.multiply(new BigInteger(Integer.toString(charset.length())));
    }
    if (_max == null || _max.compareTo(total)<0) _max = total;
    if (_min == null || _min.compareTo(total)>0) _min = total;
    _cache.put(id, total);
    return total;
}
```

*Figure 1-4 WebScarabs arithmetic encoding function (OWASP, 2012)*

While WebScarab's Visualisation feature is helpful, using it can be time consuming. It requires launching the proxy and consumes additional time if it is not installed which is likely to be the case as other proxies are more widely used by penetration testers as mentioned before. Subsequently, it requires the user to browse the target website until session IDs are assigned and then requires interaction with the proxy's user interface to obtain more cookies and visualise them. A different implementation of this functionality could be reproduced with scripting, possibly improving the efficiency of this task.

Scripting is the use of coding to automate repetitive tasks and processes to save time, reduce errors, and improve efficiency. It involves using scripting languages to instruct a runtime environment. Unlike compiled languages, which are translated into machine code before execution, scripting languages are interpreted and executed directly (Joy, no date). Examples of scripting languages are Python, Bash, PowerShell, and Ruby. Scripting is essential to cybersecurity as it enables the automation of many system administration, vulnerability scanning, and penetration testing tasks (Alam, 2022; Gillespie-Lord, 2022). Similarly, threat actors can utilise scripting for attack automation and script-based malware (Propper,

2022). In penetration testing, it is an indispensable tool for performing security evaluations on web applications or larger entities such as company networks and utilised to automate reconnaissance phases, vulnerability scanning, and delivering exploits, among others (Andress & Linn, 2016). These tools are useful in the many web application penetration testing stages defined by different methodologies, of which the Session Management Testing stage defined by OWASP Web Security Testing Guide v4.2 is addressed with the development of a cookie value visualisation script.

## 1.2 AIM

This project aims to produce a Python script that performs website cookie value visualisation demonstrating how these values correspond with time. This would aid in determining if a website's cookies are predictable during web application penetration testing. This aim consists of the following sub-aims:

- Developing a functionality that retrieves session IDs from a given website.
- Implementation of an algorithm that encodes the session ID values to a format that can be plotted to a graph.
- Implementation of functionality that plots the encoded and time values.
- Consideration of how the output produced by the script is saved.
- Handling of command-line arguments.

# 2 PROCEDURE

## 2.1 OVERVIEW OF PROCEDURE

The procedure was split into smaller steps corresponding to the sub-aims defined in the project aims section (Section 1.2). These phases included:

1. Retrieving session IDs:
   1.1. Sending HTTP requests to the target site.
   1.2. Obtaining the returned cookie value dictionary from the session object.
   1.3. Writing the timestamp of the request and the response cookie value to a CSV (in a separate file for each unique cookie) and saving it.
2. Encoding:
   2.1. Performing arithmetic encoding on the cookie values in each CSV file and appending the new values to the CSV.
3. Visualisation:
   3.1. Reading the encoded cookie values and timestamps, plotting them to a scatter graph, and saving it.
4. Implementing a command-line interface.
5. Additional performance considerations.

Section 2.2 onwards follows the defined phases in explaining the delivery of the final script, the full source code can be found in Appendix A – Source Code.

### 2.1.1 Development Environment

This project was developed in a *venv* virtual environment on a Linux-based system and written in Python 3.

During the testing process, the web server on Tiny Core Linux virtual machine was reused from *CMP319 – Web Application Penetration Testing*. It hosted a suspectable web application for *Astley Skateshop* which was known to use both randomised and predictable cookie values which made it a suitable target to test the script's functionality on in addition to not being a public-facing entity. Additionally, three other links – *https://github.com*, *https://www.google.com*, and *http://google.com,* were tested during the later stages of development.

### 2.1.2 Used Libraries

The following libraries were used for the script:

- **External libraries:**
  - **Tldextract** – to obtain the target domain for naming the output directory.
  - **Requests** – to make HTTP requests to the target and obtain cookies.
  - **Pandas** – to append the encoded value column and labels to an existing CSV.
  - **Matplotlib:**

        - **Pyplot** – to create a scatter graph of cookie values over time.
        - **Dates** – to format the time axis.
- **Python standard library:**
    - **Argparse** – to implement a command-line interface.
    - **Sys** – to terminate the script's execution if needed.
    - **Multiprocessing** – to concurrently encode and draw different cookies.
    - **CSV** – to read/write CSV files.
    - **Time** – for implementing a basic request delay.
    - **Pathlib (Path)** – to create/remove output directories.
    - **Datetime** – to obtain current time and format timestamps.
    - **Collections (Counter)** – to count unique characters in a cookie value string.

The used external libraries were listed in a *requirements.txt* file (Figure 2-1) to allow users to easily obtain them with *pip install*.

```
matplotlib==3.7.1
pandas==1.5.3
requests==2.27.1
tldextract==3.4.0
```

*Figure 2-1 requirements.txt*

## 2.2 Retrieving Session IDs

The session IDs were retrieved by making HTTP requests with the *requests* library. A set number of requests specified by the user were made to the target site, and the session cookies were obtained from the session object. The session cookie directory was then iterated and a new CSV file for each new cookie encountered was made. The request time (in a datetime format) and the returned cookie value were written to the file with a new row for each request. Additionally, each file was added to a list of created filenames which would later be used to encode the cookie values of each file and visualise them. Figure 2-2 displays this process.

```
for x in range (reqs):
    # make the request and post data (if any)
    sess = requests.Session()
    auth = sess.post(url, data=payload, timeout = timeout)
    resp = sess.get(url, timeout = timeout)
    cookies = sess.cookies.get_dict() # get cookie names and values in a dict

    # iterate thorugh the dictionary if there are multiple cookies
    for key, value in cookies.items():
        filename = path + key + '.csv'

        # keep track of how many different cookie files are created so they can all be parsed later
        if filename not in c_files:
            c_files.append(filename)

        # make (or open) the csv file and append the cookie and timestamp
        with open(filename, 'a', newline='') as file:
            writer = csv.writer(file)
            row = [datetime.now(), value]
            writer.writerow(row)
```

*Figure 2-2 Making requests to the target URL and writing the cookie values to CSV files*

In the case any exceptions were encountered, the script output these and terminated (Figure 2-3). Additionally, to avoid the script hanging for a prolonged time in some cases while waiting for a connection (e.g. in the case of an invalid URL being accessed), a connection timeout of 15 seconds was specified (seen in Figure 2-2).

```
except Exception as e:
    Path(path).rmdir() # delete the empty directory
    sys.exit(f"Script terminated due to encountered exceptions: {str(e)}")
```

*Figure 2-3 Deleting the empty directory and terminating after outputting the encountered exceptions*

Both a POST and a GET request were made to the target site during each iteration of the for loop. The POST request was used for authentication and posted any supplied payload with user credentials to a login form indicated in the site link, obtaining any cookies assigned upon successful authentication, while the GET request obtained any cookies that were assigned by just visiting the site. The GET request was considered vital to the functionality of the script as in the case of the user not desiring to authenticate or supplying the wrong credentials nothing would be returned after posting an empty/incorrect payload and no cookie values would be obtained.

Lastly, as cookies were sometimes observed not to update at all while the requests were being made, a simplistic request delay mechanism was added to the script. It used the *sleep* function from the *time* library to delay the code responsible for making requests for 0.5 seconds between each request (Figure 2-4) and could be used as a command line flag. This delay allowed to slow the requests down, so they would not complete before the cookie value gets updated, outlining any changes in cookie values while simultaneously making fewer requests.

```
# basic delay
if throttle:
    time.sleep(0.5)
```

*Figure 2-4 Request delay*

### 2.2.1 Authentication

The authentication followed a simple outline of posting a user-supplied payload to the target site. It required for the specific authentication form to be linked to, and the payload to be supplied in a comma-separated format containing the field names and values, including the name and value of the submit button. Figure 2-5 contains an example payload with the credentials that were used for authenticating with the test web application.

```
user_email,hacklab@hacklab.com
user_password,hacklab
user_login,
```

*Figure 2-5 Example of a formatted payload used to authenticate with the target*

### 2.2.2 Output Directory

The generated cookie value CSV files were saved to a directory created within the calling directory. This directory was created on runtime using the domain name, obtained with the *tldextract* library, combined

with the execution timestamp as the directory name (Figure 2-6). The directory was created with *pathlib* and its naming convention ensured that it would not conflict with other directories. As some sites assign numerous cookies, all the graphs and generated CSV files (making it two output files for each cookie) were contained within the new directory to organise them. Additionally, output files generated on runtime did not pose the risk of overwriting/clashing with existing files as the output directory was empty upon creation.

```
# make an output directory based on the domain name and current time
domain = tldextract.extract(url).domain
timestamp = datetime.now().strftime('%d%m%y_%H%M%S')
path = domain + '_' + timestamp + '/'
Path(path).mkdir(exist_ok=True)
print(f"Results will be written to '{path}'")
```

*Figure 2-6 Creating an output directory*

## 2.3 ENCODING

As discussed in Section 1.1, WebScarab utilises arithmetic encoding to obtain decimal representations of the cookie value string, so an implementation of this compression algorithm was used in this scenario as well. The encoding stage ensured that the cookie values could be visualised on a graph during later stages, as decimal values can be plotted. The implementation of this algorithm was based on Arithmetic Coding For Data Compression by Wien, et al. (1987) which defines the steps of determining the character distributions in each cookie and encoding the characters based on this distribution in an interval of [0;1]. These steps are further discussed in Section 2.3.1 and Section 2.3.2.

To encode the cookie values the script first iterated through each row of the CSV file created for a specific cookie in the previous phase (Section 2.2) and passed the value from each row to an arithmetic encoding function. The returned encoded cookie values were stored in a list and appended to the CSV file, with the addition of column labels, by using the *pandas* library (Figure 2-7).

```
# parse the cookie csv file and update it with the encoded values
def parse(filename):
    probabilities = {}

    # open cookie file
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        newdata = []

        # encode each cookie and add the value to the list
        for row in reader:
            newdata.append(encode(row[1], probabilities))

        # rewrite the file adding encoded values and labels
        df = pd.read_csv(filename, names=['Time', 'Value'])
        df['Decimal Value'] = newdata
        df.to_csv(filename, index = False)
```

*Figure 2-7 Cookie values passed to the encoding function and encoded values added to the CSV*

The main encode function, which wrapped around the probability and arithmetic encoding functionalities discussed before, consisted of a check for saved cookie values to avoid repeated calculations if a cookie value to encode had not changed from the previous one. If the values matched, the previous encoded value was returned. If a new encoding operation was needed, it updated a probabilities dictionary and called the arithmetic encoding function. It also inserted both the original and encoded cookie values in a cache list for future comparison before returning (Figure 2-8).

```
# main cookie encoding function
def encode(cookie, probabilities):
    # to avoid repeated calculations check if there is a cached cookie and encoded value
    # if list is not empty compare current and previous cookie, if the cookie value is the same use last encoded value and return
    if cache:
        if cookie == cache[0]:
            return cache[1]

    # update character probabilities based on current cookie and encode it using arithmetic encoding
    probabilities = get_probability(cookie, probabilities)
    encoded_value = arithmetic_encode(cookie, probabilities)

    # add cookie and its encoded value to the cache
    cache.insert(0, cookie)
    cache.insert(1, encoded_value)

    return encoded_value
```

*Figure 2-8 Main encoding function*

### 2.3.1 Probability Distribution

As seen in Figure 2-7, an empty probabilities dictionary was initialised for each CSV file. It stored the cookie value character probability distribution and updated it for each subsequent cookie it encoded. The probabilities were calculated by first using a counter to count the occurrences of every unique character present in the cookie, and then determining the length of its value string. Then the probability of a specific character by dividing its occurrence by the total character count was calculated. Subsequently, the probability distribution's end value in the interval [0,1) was calculated by adding the probability to the start variable, which was 0 at the beginning (Figure 2-9).

```
# get probabilities
def get_probability(cookie, probabilities):
        char_count = Counter(cookie)
        total_chars = len(cookie)
        start = 0.0

        for char, count in char_count.items():
            probability = count / total_chars # occurence of an unique character against the total amount of characters
            end = start + probability # end of distribution
```

*Figure 2-9 Character probability calculation*

This resulted in a pair of floats representing the cumulative distribution of a specific character in the cookie. This pair of values was added to the probability dictionary with the character it was representing as the key (Figure 2-10). The start variable (0 in the beginning) was also reassigned to be the same as the end variable for the next iteration, so the succeeding character distribution would start from where the previous one ended.

```
    else:
        probabilities[char] = (start, end) # add previosuly unencountered characters

    start = end # reassign start of next distribution

return probabilities
```

*Figure 2-10 Adding the cumulative character distribution to the probability dictionary*

Later during the encoding process, the probabilities were updated for previously encountered characters. This involved reassigning the start and end values of a character distribution interval in the probabilities dictionary with updated values. The values were updated by multiplying the old start and end by the width of the newly calculated range. This scaled the old range to fit within the new one. Adding the start value to the scaled range offsets defined the new probability distribution (Figure 2-11).

```
if char in probabilities:
    # update character probability
    old_start, old_end = probabilities[char]
    new_start = start + old_start * (end - start)
    new_end = start + old_end * (end - start)
    probabilities[char] = (new_start, new_end)
```

*Figure 2-11 Updating the probability distribution*

### 2.3.2 Arithmetic Encoding

The main arithmetic encoding operations occurred in the *arithmetic_encode()* function. It initialised an interval between 0 and 1 representing the possible range for the decimal cookie values to fall into. Then the characters in the cookie were iterated through to encode the entire string. This was done by updating the start of the possible encoded value interval each time by incrementing it with its size and the specific character's probability range's starting value. Then the size variable was updated by multiplying it by the size of the probability range which scaled down the interval for each subsequent character to encode. This procedure was repeated for each character in the cookie and the midpoint between the final *interval_start* and *interval_size* values was used as the encoded value (Figure 2-12). It was additionally subtracted from 1 to cast smaller values to larger ones and vice versa for clarity because it could be observed that before flipping the values, they were decreasing for cookies that depended on time passing instead of increasing. For plot formatting purposes, this result was also multiplied by 10.

```
# perform arithmetic encoding to encode the cookie value into decimal
def arithmetic_encode(cookie, probabilities):
    # initialize the interval to 0 - 1
    interval_start = 0.0
    interval_size = 1.0

    # encode each character in the cookie
    for char in cookie:
        char_range = probabilities[char] # get the probabilities range for the current character
        # update the interval based on the character range
        interval_start += interval_size * char_range[0]
        interval_size *= char_range[1] - char_range[0]

    return (1 - (interval_start + 0.5 * interval_size)) * 10 # return the midpoint of interval as the encoded value.
```

*Figure 2-12 Arithmetic encoding function*

## 2.4 VISUALISATION

The plotting was performed with the *mathplotlib* libraries' *pyplot* interface. The CSV file for each cookie was opened and the timestamps and encoded decimal values were read into a list for each while skipping over the header row with column labels. Next, a scatter graph was created and formatted, which included adding axis names and labels, formatting the timestamps with *mathplotlib's dates* interface, and setting maximum tick frequency and label rotation. Lastly, the graph was saved as a PNG file with the same name as the cookie (Figure 2-13).

```
# make a graph
def draw(filename):
    timestamps = []
    values = []

    imagefile = filename[:-4] + '.png' # image file name

    # read the timestamps and encoded cookie values
    with open(filename, 'r') as readf:
        reader = csv.reader(readf, delimiter=',')
        next(reader) # skip the header row

        for row in reader:
            timestamps.append(datetime.strptime(row[0], '%Y-%m-%d %H:%M:%S.%f'))
            values.append(float(row[2]))

    plt.scatter(timestamps, values, c = 'r')  # plot the values to a graph

    # format the graph
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
    plt.gca().xaxis.set_major_locator(plt.MaxNLocator(7))
    plt.xticks(rotation = 25)
    plt.xlabel('time')
    plt.ylabel('value')
    plt.title(f"{filename[:-4].split('/')[1]} Values Over Time", fontsize = 20)

    plt.savefig(imagefile, bbox_inches='tight') # save the image
    plt.close()
```

*Figure 2-13 Graphing function*

## 2.5 COMMAND-LINE INTERFACE

The command-line interface for this script was implemented with *argparse* (Figure 2-14). A positional argument was defined to use for the target URL and optional arguments were made for the authentication payload, request number, and request delay. The provided arguments were used during runtime, and, additionally, this allowed a help page to be created to improve user experience.

```
parser = argparse.ArgumentParser(prog = 'cookies.py',  description = 'Cookie value over time visualisation tool',

parser.add_argument('url', type = str, help = 'url to test. if authenticating, provide the url of the form to pos
parser.add_argument('-p', '--payload', type = str, help = textwrap.dedent('credentials to use for authenticating
parser.add_argument('-r', '--requests', type = int, help = 'number of requests in the range [10, 200] to make. de
parser.add_argument('-t', '--throttle', action='store_true', help = 'use a simple request delay of 0.5 seconds. u

args = parser.parse_args()
main(args)
```

*Figure 2-14 Fragment of the code fragment using argparse*

### 2.5.1 User Input Error Handling

Error handling was partly done with *argparse* as argument types could be specified. Additionally, user input for the authentication payload was read into a dictionary, and errors pretraining to non-existent files or invalid file extensions were handled (Figure 2-15). No further checks were done to see if the payload matched the formatting required, however, the payload was displayed on runtime so any errors in it could be identified by the user.

```
# 2) get payload for authentication
payload = {} # default payload is empty

# if payload is provided, read it into a dictionary
if args.payload:
    try:
        with open(args.payload) as file:
                for line in file:
                    (key, sep, val) = line.strip().partition(',')
                    payload[key] = val
    except OSError:
        sys.exit("No such file or directory!")
    except UnicodeDecodeError:
        sys.exit("Invalid filetype!")
```

*Figure 2-15 Handling payload file-related errors*

Additionally, to avoid the user making an unnecessary amount of requests, the supplied number of requests was checked on runtime and defaulted to 50 if it was out of a specified range of 10 to 200 (Figure 2-16).

```
# if request number is provided, use that
if args.requests:
    reqs = args.requests
    if reqs < 10 or reqs > 200:
        # number is not within the allowed range
        print("Number of requests must be in the range [10, 200], defaulting to 50.")
        reqs = 50
```

*Figure 2-16 Request limiting*

## 2.6 PERFORMANCE CONSIDERATIONS

To improve the efficiency of this script, multiprocessing was used to concurrently parse and draw the outputs for each unique cookie (Figure 2-17 and Figure 2-18). This offered slightly increased speeds when executing these tasks.

```
get_cookies(url, payload, reqs, throttle) # get cookies
multiprocess() # parse the created cookie files
```

*Figure 2-17 Multiprocessing function call*

```python
# multiprocessing function
def multiprocess():
    print("Encoding and drawing cookie values...")

    num_processes = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(num_processes)

    pool.map(process, c_files) # process each created cookie CSV file

    pool.close()
    pool.join()

def process(filename):
    parse(filename) # parse the collected data
    draw(filename) # plots the values on graphs and saves them
```

*Figure 2-18 Multiprocessing function*

# 3 DISCUSSION

## 3.1 GENERAL DISCUSSION

The core aim of developing a predictable cookie value visualisation tool has been met successfully. The resulting script meets the sub-aim of making requests and is capable of making GET and POST HTTP requests to a supplied target website to obtain cookie values from it. Furthermore, it meets the encoding aim by demonstrating the use of arithmetic encoding to compress these values into decimal formats to plot them to a graph to meet the visualisation requirement. Figure 3-1 and Figure 3-2 demonstrate two PNG files generated by the script on runtime which indicate randomised cookie values for *PHPSESSID* and predictable ones for *SecretCookie*.

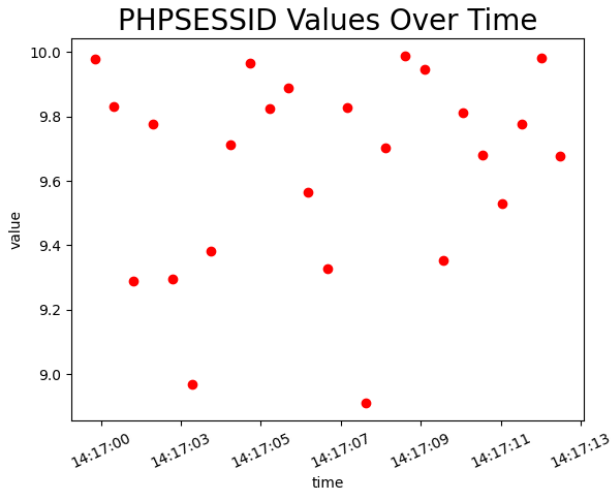


*Figure 3-1 Random cookie values visualised*

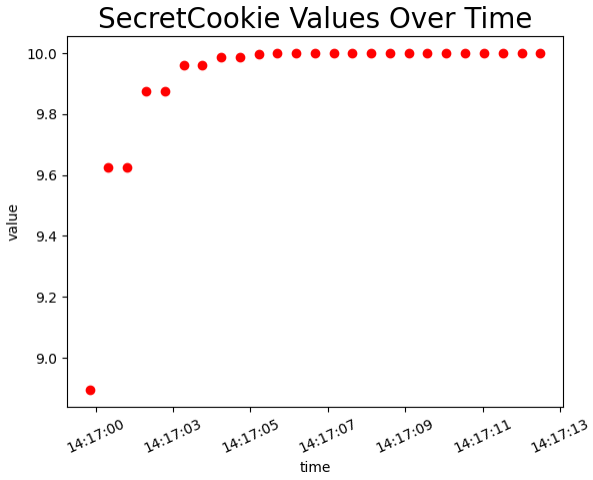


*Figure 3-2 Predictable cookie values visualised*

Additionally, the output is saved to a directory named after the target site and execution timestamp, which is created on runtime (Figure 3-3). If errors pertaining to establishing a connection to the target are encountered, the empty directory is deleted. The generated output files include a CSV file, containing timestamps, cookie values, and encoded values with column labels for the user (Figure 3-4), and a PNG file, containing a scatter graph that showcases the cookie value changes over time, for each unique cookie. These files are also named to match the cookie name (Figure 3-5).

```
(venv) ance@ANCES-HP:~/files/py$ ls -d */
192.168.1.10_190523_124908/  github_190523_122826/  google_190523_124400/
```

*Figure 3-3 Directories created by the script*

```
(venv) ance@ANCES-HP:~/files/py/192.168.1.10_190523_124908$ cat SecretCookie.csv | head
Time,Value,Decimal Value
2023-05-19 12:49:08.967536,6861636o6p6162406861636o6p61622r636s6q3n6861636o6p61623n31363834343936393438,8.796894043733936
2023-05-19 12:49:09.492327,6861636o6p6162406861636o6p61622r636s6q3n6861636o6p61623n31363834343936393439,9.578335362274649
2023-05-19 12:49:10.016078,6861636o6p6162406861636o6p61622r636s6q3n6861636o6p61623n31363834343936393439,9.578335362274649
```

*Figure 3-4 CSV file generated by the script*

```
(venv) ance@ANCES-HP:~/files/py$ ls google_190523_124400/
AEC.csv  AEC.png  CONSENT.csv  CONSENT.png  SOCS.csv  SOCS.png  __Secure-ENID.csv  __Secure-ENID.png
```

*Figure 3-5 Output CSV and PNG files for each cookie*

Additionally, this script supports command-line arguments. Figure 3-6 showcases the use of command line arguments with this script and the text it produces on runtime.

```
(venv) ance@ANCES-HP:~/files/py$ python cookies.py http://192.168.1.10/userlogin.php -p p.txt -r 25 -t
Getting cookies...
Using payload: {'user_email': 'hacklab@hacklab.com', 'user_password': 'hacklab', 'user_login': ''}
Throttling: True
Results will be written to '192.168.1.10_190523_124908/'
Encoding and drawing cookie values...
Done!
```

*Figure 3-6 Using command-line arguments and running the script*

A help menu, explaining the use of this script and its usage, can be output with the *-h* (or *--help*) argument (Figure 3-7).

```
(venv) ance@ANCES-HP:~/files/py$ python cookies.py -h
usage: cookies.py [-h] [-p PAYLOAD] [-r REQUESTS] [-t] url

Cookie value over time visualisation tool

positional arguments:
  url                   url to test. if authenticating, provide the url of the form to post data to

optional arguments:
  -h, --help            show this help message and exit
  -p PAYLOAD, --payload PAYLOAD
                        credentials to use for authenticating in a file form. has to follow a comma seperated value
                        format of [fieldname],[value] (including the submit button value) with a new row for each field
                        Example:

                        username,user
                        password,pass
                        submit,

  -r REQUESTS, --requests REQUESTS
                        number of requests in the range [10, 200] to make. defaults to 50.
  -t, --throttle        use a simple request delay of 0.5 seconds. useful to slow requests down to observe changes if
                        outputted graph is a horizontal line.
```

*Figure 3-7 Help menu*

Lastly, additional performance considerations have been acknowledged with the use of multiprocessing (Section 2.6) to parse and draw separate cookie value files faster, however, considering how light this script is, the difference is not major.

Overall, this script performs its designated task sufficiently and helps visualise possibly predictable session ID values that could pose a risk to the security of a given website. By doing this, these vulnerabilities can be identified and mitigated to avoid security incidents such as session hijacking to take place.

## 3.2 COUNTERMEASURES

This script lacks an enhanced accuracy of the arithmetic encoding algorithm it implements. One of the ways this manifests is the compressed values that follow a logically increasing sequence quickly reaching the upper limit of the interval. While the algorithm performs sufficiently, it can be improved by solving this and other issues. One of the key considerations would be an approach closer to WebScarab's implementation, which is also described in Arithmetic Encoding for Data Compression by Wien, et al., where in addition to the encoding of each character, maximum and minimum limits are defined to ensure that the calculation does not escape the allowed interval. While an interval of [0;1] is present in the code, there is no check to ensure that the encoded value remains within it. Additionally, the probability distributions defined by these sources differ from the approach utilised by the script, for example, Wien, et al. define the use of frequency tables rather than calculating the probabilities from scratch as done in the script. A backwards convention also appears throughout both WebScarab and Arithmetic Encoding for Data Compression, the probabilities are stored backwards for convenience, but more notably, Webscarab iterates through the cookie value string backwards, which is suspected to be why the algorithm used here returned values that were flipped and had to be reordered by subtracting them from a set number. The ease of implementation for this would be the hardest of the suggested improvements and require a thorough understanding of the arithmetic encoding algorithm, however, it would contribute the most to the current performance of the script.

Next, the request delay used by the script, while fit for this scenario as the number of requests is relatively low and does not execute in parallel to other tasks in addition to the delay being short, could be improved with a simple use of rate limiting libraries like *ratelimit*, which can offer bigger flexibility to the request delay functionality of the code. In addition to this, custom delay lengths could be specified by the user.

Error handling in this script has some drawbacks in the aspect that it could be more user-friendly. Many exceptions are handled in a single *Except* block, which does not allow customising error messages to inform the user adequately other than outputting the specific errors raised on runtime, which is not always sufficient to explain what has occurred more in-depth. Additionally, upon connection-related errors, the empty output directory is deleted, however, this is done under the assumption that the error is encountered during the first request made and does not account for the connection being terminated on runtime. A basic solution to this is to implement and requires the identification of the specific errors that can be raised at a given part of the code and handling them within a separate *Except* block. Additionally, a check for whether the output directory is empty can be made before attempting a deletion. If some output has already been generated it can be either deleted as well or a note inserted within a directory that errors were encountered.

Another drawback of this script is the set timeout. Ideally, it could use a default value and allow the user to specify it. This would avoid the script from terminating because of the user taking longer than the specified timeout to establish a connection. Implementing the solution to this would require minimal changes to the code, with an additional optional *argparse* argument added where the user can specify a timeout. This argument would be passed to the cookie gathering function or the default 15 seconds used instead.

## 3.3 FUTURE WORK

Aside from work relating to improving the components of the script identified in the previous sections, future work relating to the continued development of the script could include adding additional logic to the code which could determine whether to use a delay or not based on the returned cookie value changes.

This script could also perform analysis of the output files, and determine whether the cookies are guessable by examining the encoded values and the user could be informed of this before examining the output themselves. It could output the results of this stage in a report format.

Additionally, the extent of session ID analysis this script undertakes can be expanded even further by analysing other aspects relating to their security. This would involve examining cookie attributes, expiration times, and flag usage to identify potential weaknesses or misconfigurations. Again, this would be presented in a report format.

# REFERENCES


Alam, N., 2022. *3 Scripting Languages Cybersecurity Professionals Must Know.* [Online]
Available at: https://security.packt.com/3-scripting-languages-cybersecurity-professionals-must-know/
[Accessed 1 April 2023].

Andress, J. & Linn, R., 2016. *Coding for Penetration Testers.* 2nd ed. Cambridge: Syngress.

Gillespie-Lord, E., 2022. *What Is Scripting and What Is It Used For?.* [Online]
Available at: https://www.bestcolleges.com/bootcamps/guides/what-is-scripting/
[Accessed 1 April 2023].

Gregg, M. & Santos, O., 2022. *CEH Certified Ethical Hacker Cert Guide.* 4th ed. London: Pearson IT Certification.

Joy, A., no date. *Why Scripting is Essential for Cybersecurity Professionals.* [Online]
Available at: https://pythonistaplanet.com/scripting-in-cybersecurity/
[Accessed 1 April 2023].

OWASP, 2012. *DefaultCalculator.java.* [Online]
Available at: https://github.com/OWASP/OWASP-WebScarab/blob/d22bd604b981ae2de73cd5d39305ac25c9367d1b/src/org/owasp/webscarab/plugin/sessionid/DefaultCalculator.java#L126
[Accessed 4 April 2023].

OWASP, 2020. *OWASP-WebScarab.* [Online]
Available at: https://github.com/OWASP/OWASP-WebScarab/tree/d22bd604b981ae2de73cd5d39305ac25c9367d1b
[Accessed 4 April 2023].

OWASP, 2020. *WSTG - v4.2.* [Online]
Available at: https://owasp.org/www-project-web-security-testing-guide/v42/
[Accessed 4 April 2023].

Propper, G., 2022. *What are script-based attacks and what can be done to prevent them?.* [Online]
Available at: https://www.helpnetsecurity.com/2020/07/31/what-are-script-based-attacks/
[Accessed 1 April 2023].

Stuttard, D. & Pinto, M., 2011. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws.* 2nd ed. Indianapolis: John Wiley & Sons.

Sunny Valley Networks, no date. *What is a Session Hijacking?.* [Online]
Available at: https://www.sunnyvalley.io/docs/network-security-tutorials/what-is-session-hijacking
[Accessed 18 May 2023].

Whitehorn, M., 2020. *What tools are used when penetration testing a Web Application?.* [Online]
Available at: https://www.secureideas.com/knowledge/what-tools-are-used-when-penetration-testing-

a-web-application
[Accessed 10 April 2023].

Wien, I. H., Neal, R. M. & Cleary, J. G., 1987. Arithmetic Encoding for Data Compression. *Communications of the ACM ,* 30(6), p. 520–540.

# APPENDICES

## APPENDIX A – SOURCE CODE

cookies.py

```
"""
Name: cookies.py
Desc: cookie value over time visualisation tool
Auth: Ance Strazdina
Date: 22/05/2023
"""

# imports
import argparse, sys, multiprocessing, tldextract, requests, csv, time
from pathlib import Path
from datetime import datetime
from collections import Counter
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

# global variables
c_files, cache = [], []

# get cookie names and values and write to csv
def get_cookies(url, payload, reqs, throttle):
    print("Getting cookies...")
    print(f"Using payload: {payload}")
    print(f"Throttling: {throttle}")

    # make an output directory based on the domain name and current time
    domain = tldextract.extract(url).domain
    timestamp = datetime.now().strftime('%d%m%y_%H%M%S')
    path = domain + '_' + timestamp + '/'
    Path(path).mkdir(exist_ok=True)
    print(f"Results will be written to '{path}'")

    timeout = 15 # connection-timeout for requests, to avoid the script hanging

    try:
        for x in range (reqs):
            # make the request and post data (if any)
```

```
            sess = requests.Session()
            auth = sess.post(url, data=payload, timeout = timeout)
            resp = sess.get(url, timeout = timeout)
            cookies = sess.cookies.get_dict() # get cookie names and values in a dict

            # iterate thorugh the dictionary if there are multiple cookies
            for key, value in cookies.items():
                filename = path + key + '.csv'

                # keep track of how many different cookie files are created so they can all be
parsed later
                if filename not in c_files:
                    c_files.append(filename)

                # make (or open) the csv file and append the cookie and timestamp
                with open(filename, 'a', newline='') as file:
                    writer = csv.writer(file)
                    row = [datetime.now(), value]
                    writer.writerow(row)

            # basic delay
            if throttle:
                time.sleep(0.5)

    except Exception as e:
        Path(path).rmdir() # delete the empty directory
        sys.exit(f"Script terminated due to encountered exceptions: {str(e)}")

# parse the cookie csv file and update it with the encoded values
def parse(filename):
    probabilities = {}

    # open cookie file
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        newdata = []

        # encode each cookie and add the value to the list
        for row in reader:
            newdata.append(encode(row[1], probabilities))

        # rewrite the file adding encoded values and labels
        df = pd.read_csv(filename, names=['Time', 'Value'])
        df['Decimal Value'] = newdata
```

```
        df.to_csv(filename, index = False)

# main cookie encoding function
def encode(cookie, probabilities):
    # to avoid repeated calculations check if there is a cached cookie and encoded value
    # if list is not empty compare current and previous cookie, if the cookie value is the
same use last encoded value and return
    if cache:
        if cookie == cache[0]:
            return cache[1]

    # update character probabilities based on current cookie and encode it using arithmetic
encoding
    probabilities = get_probability(cookie, probabilities)
    encoded_value = arithmetic_encode(cookie, probabilities)

    # add cookie and its encoded value to the cache
    cache.insert(0, cookie)
    cache.insert(1, encoded_value)

    return encoded_value

# get probabilities
def get_probability(cookie, probabilities):
        char_count = Counter(cookie)
        total_chars = len(cookie)
        start = 0.0

        for char, count in char_count.items():
            probability = count / total_chars # occurence of an unique character against the
total amount of characters
            end = start + probability # end of distribution

            if char in probabilities:
                # update character probability
                old_start, old_end = probabilities[char]
                new_start = start + old_start * (end - start)
                new_end = start + old_end * (end - start)
                probabilities[char] = (new_start, new_end)
            else:
                probabilities[char] = (start, end) # add previosuly unencountered characters

            start = end # reassign start of next distribution
```

```
        return probabilities

# perform arithmetic encoding to encode the cookie value into decimal
def arithmetic_encode(cookie, probabilities):
    # initialize the interval to 0 - 1
    interval_start = 0.0
    interval_size = 1.0

    # encode each character in the cookie
    for char in cookie:
        char_range = probabilities[char] # get the probabilities range for the current
character
        # update the interval based on the character range
        interval_start += interval_size * char_range[0]
        interval_size *= char_range[1] - char_range[0]

    return (1 - (interval_start + 0.5 * interval_size)) * 10 # return the midpoint of interval
as the encoded value. subtracted from 1 for graphing purpouses (encoding makes technically
"increasing" cookie values decrease). multiplied by 10 for graphing purpouses

# make a graph
def draw(filename):
    timestamps = []
    values = []

    imagefile = filename[:-4] + '.png' # image file name

    # read the timestamps and encoded cookie values
    with open(filename, 'r') as readf:
        reader = csv.reader(readf, delimiter=',')
        next(reader) # skip the header row

        for row in reader:
            timestamps.append(datetime.strptime(row[0], '%Y-%m-%d %H:%M:%S.%f'))
            values.append(float(row[2]))

    plt.scatter(timestamps, values, c = 'r')  # plot the values to a graph

    # format the graph
    plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))
    plt.gca().xaxis.set_major_locator(plt.MaxNLocator(7))
    plt.xticks(rotation = 25)
    plt.xlabel('time')
    plt.ylabel('value')
```

```
    plt.title(f"{filename[:-4].split('/')[1]} Values Over Time", fontsize = 20)

    plt.savefig(imagefile, bbox_inches='tight') # save the image
    plt.close()

# multiprocessing function
def multiprocess():
    print("Encoding and drawing cookie values...")

    num_processes = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(num_processes)

    pool.map(process, c_files) # process each created cookie CSV file

    pool.close()
    pool.join()

def process(filename):
    parse(filename) # parse the collected data
    draw(filename) # plots the values on graphs and saves them

# main function
def main(args):
    # 1) get url
    url = args.url

    # 2) get payload for authentication
    payload = {} # default payload is empty

    # if payload is provided, read it into a dictionary
    if args.payload:
        try:
            with open(args.payload) as file:
                    for line in file:
                        (key, sep, val) = line.strip().partition(',')
                        payload[key] = val
        except OSError:
            sys.exit("No such file or directory!")
        except UnicodeDecodeError:
            sys.exit("Invalid filetype!")

    # 3) get number of requests to make
    reqs = 50 # default number of requests the script will make
```

```
    # if request number is provided, use that
    if args.requests:
        reqs = args.requests
        if reqs < 10 or reqs > 200:
            # number is not within the allowed range
            print("Number of requests must be in the range [10, 200], defaulting to 50.")
            reqs = 50

    # 4) get throttle
    throttle = args.throttle

    get_cookies(url, payload, reqs, throttle) # get cookies
    multiprocess() # parse the created cookie files

    print("Done!")

# parse the arguments & call the main function
if __name__ == '__main__':
    parser = argparse.ArgumentParser(prog = 'cookies.py',  description = 'Cookie value over
time visualisation tool', epilog = 'Ance Strazdina, CMP320 Scripting Project, 2023', add_help
= True, formatter_class = argparse.RawTextHelpFormatter)

    parser.add_argument('url', type = str, help = 'url to test. if authenticating, provide the
url of the form to post data to.')
    parser.add_argument('-p', '--payload', type = str, help = 'credentials to use for
authenticating in a file form. has to follow a comma seperated value\nformat of
[fieldname],[value] (including the submit button value) with a new row for each
field.\nExample:\n \nusername,user\npassword,pass\nsubmit, \n \n')
    parser.add_argument('-r', '--requests', type = int, help = 'number of requests in the
range [10, 200] to make. defaults to 50.')
    parser.add_argument('-t', '--throttle', action='store_true', help = 'use a simple request
delay of 0.5 seconds. useful to slow requests down to observe changes if\noutputted graph is a
horizontal line.')

    args = parser.parse_args()
    main(args)
```