HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
School of Information and Communication Technology

CTF CHALLENGE WRITE-UP

Web Exploitation

BKSEC: Maze Maze Mazeee

DucTapCodeDao 202416788

Mục lục

1 Phân tích Challenge

Challenge cho chúng ta một đường link:

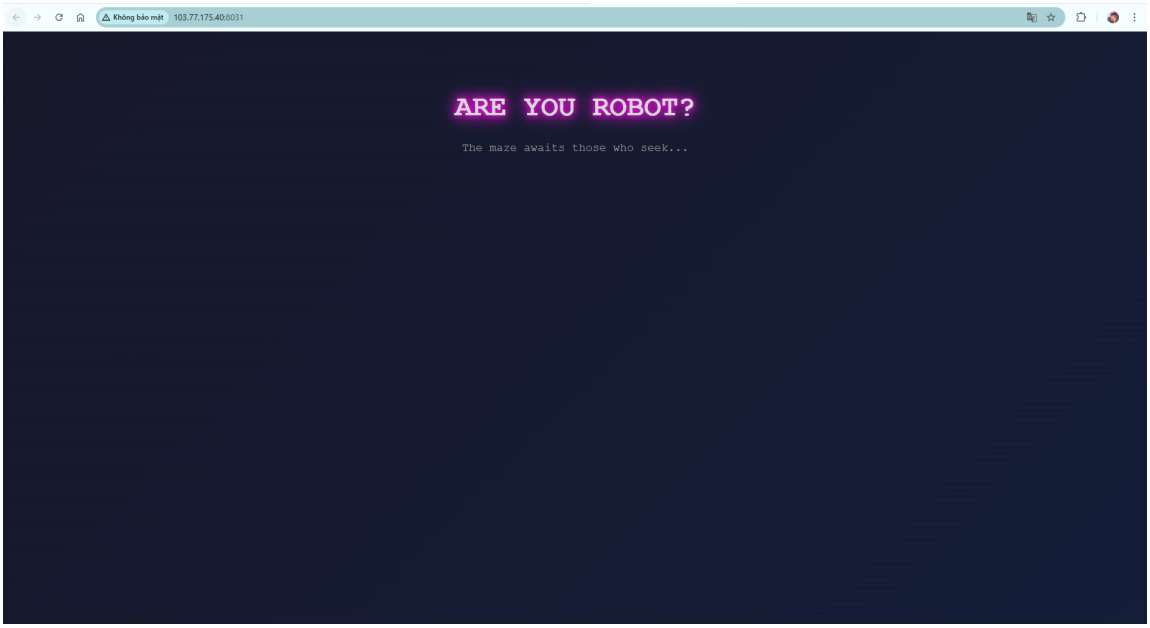
```
http://103.77.175.40:8031
```

cùng với một đoạn gợi ý:

> *"Warming up by escaping the maze is very common in CTF competitions."*

Vậy nhiệm vụ của chúng ta là **thoát khỏi mê cung** (*maze*) để tìm ra **flag**.

Trước tiên, **Duc** mở trang web bằng trình duyệt *Google Chrome*, và đây là giao diện của trang web:



Hình 1: Giao diện khi mở đường link ở trình duyệt thông thường (Google Chrome).

Nhìn vào giao diện, **Duc** chỉ thấy 2 dòng chữ là *"ARE YOU ROBOT?"* và *"The maze awaits those who seek..."*.

Vì thế, **Duc** đoán rằng có thể challenge này yêu cầu người giải thay đổi phần *HTTP Header* của đường link sao cho server nhận dạng người sử dụng là *Robot*.

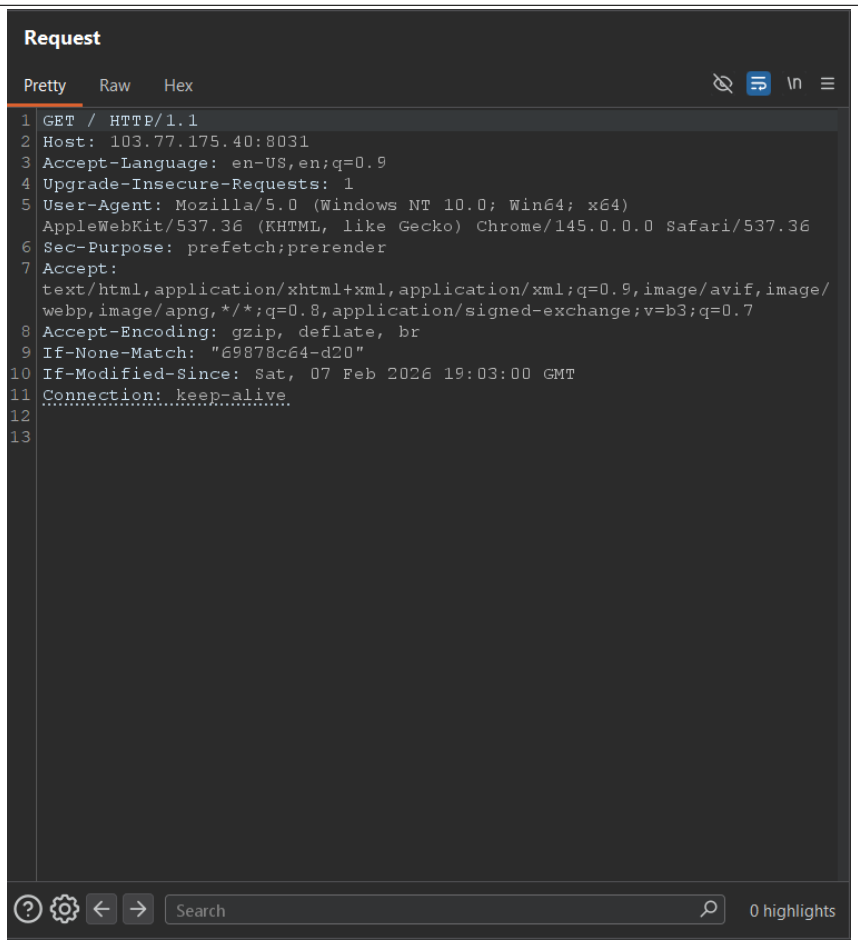
Để làm điều này, **Duc** ném link này vào *Burp Suite* để bắt đầu vọc vạch.

2 Sửa HTTP Header bằng Burp Suite và phát hiện ra hướng đi sai

Duc bật *Intercept* để có thể bắt request dễ hơn, sau đó ném lại đường link vào trình duyệt.

Tiếp theo, **Duc** chọn request `GET` trong mục *HTTP history* và dùng chức năng *Send to Repeater* để chỉnh sửa *HTTP Header* thuận tiện hơn.

Và đây là phần *HTTP Header* của trang web:



Hình 2: HTTP request header của trang web khi truy cập bằng trình duyệt thông thường.

Duc nhận thấy rằng nếu muốn "giả làm robot", phần cần ưu tiên chỉnh sửa nhất là trường *User-Agent*.

Vì vậy, **Duc** lần lượt thử thay đổi *User-Agent* thành các giá trị như `Robot`, rồi đến `Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)` (*Googlebot*), ...

Tuy nhiên, tất cả các thử nghiệm này đều trả về kết quả **404 Error**.

Điều đó cho thấy hướng tiếp cận dựa trên việc chỉnh *User-Agent* có vẻ không đúng với ý đồ của challenge. Vì vậy, **Duc** quyết định chuyển sang một hướng tiếp cận khác để tiếp tục khai thác challenge này.

3 Đổi target resource của HTTP request và thành công tìm được hướng đi đúng

Duc suy luận rằng nếu mình là một *robot* (crawler), thì thay vì truy cập trực tiếp trang chủ, nhiều khả năng sẽ đọc file `robots.txt` trước để lấy các rule điều hướng.

Vì vậy, **Duc** chỉnh dòng đầu của request trong *Repeater* thành:

```
GET /robots.txt HTTP/1.1
```

sau đó nhấn *Send*. Kết quả trả về gồm 2 dòng:

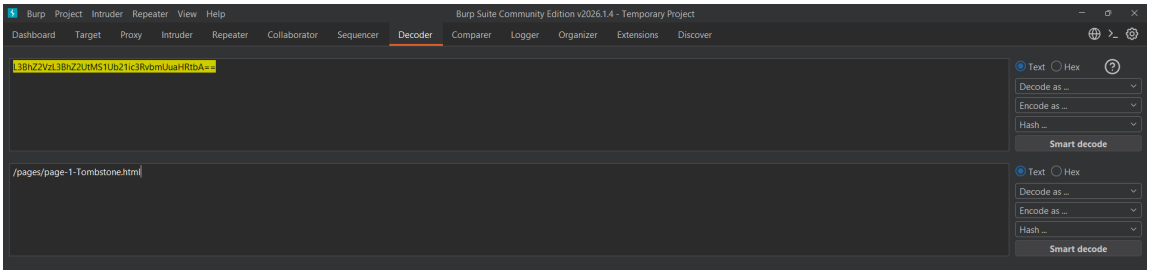
```
User-agent: *
Disallow: L3BhZ2VzL3BhZ2UtMS1Ub21ic3RvbmUuaHRtbA==
```

Đây là dấu hiệu cho thấy **Duc** đã đi đúng hướng và tìm được file chứa *PATH* cần thiết.

Quan sát kỹ hơn, **Duc** nhận ra giá trị của trường `Disallow` có dạng chuỗi được mã hoá *Base64*. Vì vậy, **Duc** copy chuỗi này, chuyển sang tab *Decoder* trong Burp Suite, paste nội dung vào và chọn *Decode as Base64*.

Sau khi giải mã, **Duc** thu được một *PATH* mới:

```
/pages/page-1-Tombstone.html
```



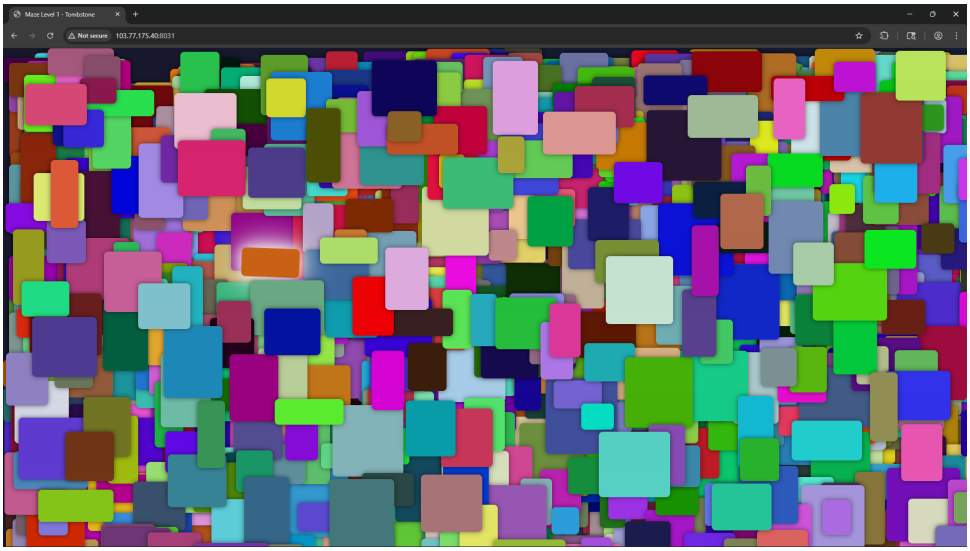
Hình 3: Giải mã giá trị trong trường `Disallow` bằng *Decoder* của Burp Suite để thu được *PATH* kế tiếp.

4 Chạy thử đường PATH được Decode và tìm ra cách giải Round 1 của Maze

Sau khi tìm được *PATH* sau khi giải mã, **Duc** đưa lại vào request với dòng đầu:

```
GET /pages/page-1-Tombstone.html HTTP/1.1
```

sau đó nhấn *Send* và thu được giao diện sau:



Hình 4: Giao diện tại `/pages/page-1-Tombstone.html` sau khi truy cập bằng *Burp* hoặc trình duyệt.

Quan sát giao diện này, **Duc** nhận thấy hướng khai thác nhiều khả năng nằm trong phần *Source* của *Response* thay vì giao diện hiển thị trực tiếp.

Vì vậy, **Duc** thử tìm từ khoá `href` trong *Response* và phát hiện một *PATH* mới, cụ thể là `page-2-Deadwood.html`.

Duc tiếp tục thử request:

```
GET /pages/page-2-Deadwood.html HTTP/1.1
```

Kết quả trả về là một trang mới nhưng có giao diện gần như giống hệt `page-1`. Sau khi lặp lại thao tác này thêm vài lần, kết quả vẫn tương tự: mỗi trang đều dẫn đến một *PATH* tiếp theo được giấu trong *Source*.

Từ đó, **Duc** nghĩ đến việc viết một script *Python* để tự động hoá quá trình này: truy cập một trang, trích xuất URL của trang tiếp theo, rồi tiếp tục truy cập trang kế tiếp và lặp lại như vậy cho đến khi không còn lấy được *PATH* mới thì dừng lại.

5 Tự động hoá việc lần theo các trang bằng Python

Sau khi nhận thấy mỗi trang đều chứa liên kết đến một trang kế tiếp trong phần *Source*, Duc viết một script Python để tự động hoá quá trình crawl: truy cập trang hiện tại, trích xuất URL tiếp theo, rồi lặp lại cho đến khi gặp *flag* hoặc không còn liên kết mới.

```python
import re                                   # dùng regex để tìm flag hoặc tìm pattern của URL kế tiếp.
import time                                 # dùng sleep() để delay giữa các request (tránh spam server).
import argparse                             # đọc tham số dòng lệnh (--start, --proxy, …) khi chạy script.
from urllib.parse import urljoin, urlparse  # urljoin: ghép URL tuyệt đối từ URL hiện tại + href tương đối (giống cách browser xử lý link).
                                            # urlparse: tách URL để lấy netloc (domain:port) nhằm kiểm tra "out of scope".
import requests                             # gửi HTTP request như một "mini browser".
from bs4 import BeautifulSoup               # parse HTML để trích href từ các thẻ <a>.

FLAG_PATTERNS = [
    re.compile(r"BKSEC\{[^}]+\}", re.I)
    # [^}]+ = "mọi ký tự bất kỳ, miễn không phải }", lặp 1+ lần.
    # re.I = case-insensitive (không phân biệt hoa thường), dù với BKSEC{} thường không cần lắm.
]

NEXT_PAGE_REGEX = re.compile(r"page-\d+[-_A-Za-z0-9]*\.html?$", re.I)
# page- literal
# \d+ = số trang
# [-_A-Za-z0-9]* = phần tên sau số (tùy bài)
# \.html? = .htm hoặc .html (vì l là tùy chọn)
# $ = kết thúc chuỗi (đảm bảo link kết thúc bằng .htm/.html)

def find_flag(text: str) -> str | None:
    for rx in FLAG_PATTERNS:
        m = rx.search(text)
        if m:
            return m.group(0)
    return None
# Nhận vào text (thường là HTML body).
# Lặp qua từng regex trong FLAG_PATTERNS.
# rx.search(text) tìm match đầu tiên ở bất kỳ vị trí nào.
# Nếu match, trả về m.group(0) = toàn bộ chuỗi khớp (ví dụ BKSEC{...}).
# Nếu không tìm thấy gì, trả None.

def extract_next_url(html: str, current_url: str) -> str | None:
    soup = BeautifulSoup(html, "html.parser")
    hrefs = []
    for a in soup.select("a[href]"):
        href = (a.get("href") or "").strip()
        if not href or href == "#":
            continue
        hrefs.append(href)

    if not hrefs:
        return None

    # ưu tiên chain kiểu page-x
    for h in hrefs:
        if NEXT_PAGE_REGEX.search(h):
            return urljoin(current_url, h)

    # fallback: lấy cái đầu
    return urljoin(current_url, hrefs[0])


def main():
    ap = argparse.ArgumentParser(description="CTF Maze crawler (via Burp proxy).")
    ap.add_argument("--base", default="http://103.77.175.40:8031", help="Base URL")
    ap.add_argument("--start", default="/pages/page-1-Tombstone.html", help="Start path or full URL")
    ap.add_argument("--max", type=int, default=500, help="Max pages")
    ap.add_argument("--delay", type=float, default=0.05, help="Delay between requests (seconds)")
    ap.add_argument("--proxy", default="http://127.0.0.1:8080", help="Burp proxy URL")
    ap.add_argument("--insecure", action="store_true", help="Disable TLS verification (useful if HTTPS through Burp)")
    args = ap.parse_args()

    # Burp proxy
    proxies = {"http": args.proxy, "https": args.proxy}

    session = requests.Session()
    session.headers.update({
        "User-Agent": "Mozilla/5.0 (CTF Maze Crawler via Burp)",
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
        "Connection": "close",
    })

    # Start URL: nếu user đưa full URL thì dùng luôn, không thì join với base
    if args.start.startswith("http://") or args.start.startswith("https://"):
        url = args.start
    else:
        url = urljoin(args.base, args.start)

    visited = set()

    for step in range(1, args.max + 1):
        if url in visited:
            print(f"[!] Loop detected: {url}")
            return
        visited.add(url)

        r = session.get(
            url,
            timeout=20,
            allow_redirects=True,
            proxies=proxies,
            verify=(not args.insecure),
        )

        print(f"[{step:03}] {r.status_code} {url}")

        # scan flag in body
        flag = find_flag(r.text)
        if flag:
            print(f"\n[+] FLAG FOUND: {flag}")
            print(f"[+] URL: {url}")
            return

        # keyword hint (không phụ thuộc find_flag)
        if re.search(r"\bflag\b", r.text, re.I):
            print("\n[!] Found keyword 'flag' in body. Inspect this page.")
            print(f"[!] URL: {url}")
            return

        next_url = extract_next_url(r.text, url)
        if not next_url:
            print("[!] No next <a href> found. Stop.")
            print(f"[!] Last URL: {url}")
            return

        # stay in-scope
        if urlparse(next_url).netloc != urlparse(args.base).netloc:
            print(f"[!] Out of scope next URL: {next_url}")
            return

        url = next_url
        time.sleep(args.delay)

    print("[-] Reached max steps without finding flag.")


if __name__ == "__main__":
    main()
```


Hình 5: Mã nguồn Python crawler được chụp bằng CodeSnap.

1) `find_flag(text)` Hàm này dùng để tìm *flag* trong nội dung phản hồi (thường là HTML response body). Cụ thể, hàm sẽ duyệt qua danh sách các biểu thức chính quy trong biến `FLAG_PATTERNS`, sau đó dùng `search()` để tìm chuỗi khớp đầu tiên. Nếu tìm thấy, hàm trả về toàn bộ chuỗi *flag* (ví dụ `BKSEC{...}`); nếu không tìm thấy thì trả về `None`.

2) `extract_next_url(html, current_url)` Hàm này dùng để trích xuất URL của trang kế tiếp từ mã HTML hiện tại. Đầu tiên, hàm parse HTML bằng `BeautifulSoup`, sau đó lấy toàn bộ các thẻ `<a>` có thuộc tính `href`. Tiếp theo:

- Bỏ qua các `href` rỗng hoặc chỉ là `#`.
- Ưu tiên chọn liên kết khớp mẫu `page-{số}-*.html` bằng regex `NEXT_PAGE_REGEX`.
- Nếu không có link nào khớp mẫu ưu tiên, hàm dùng *fallback* là lấy liên kết đầu tiên.

Cuối cùng, hàm dùng `urljoin()` để chuyển liên kết tương đối thành URL đầy đủ dựa trên `current_url`.

3) `main()` Đây là hàm điều phối toàn bộ quá trình crawl. Các nhiệm vụ chính:

- Khởi tạo các tham số dòng lệnh bằng `argparse` (base URL, start path, số bước tối đa, delay, proxy Burp, v.v.).
- Tạo `requests.Session()` và thiết lập các *HTTP Header* cơ bản (đặc biệt là `User-Agent`).
- Xác định URL bắt đầu: nếu `-start` là URL đầy đủ thì dùng luôn, nếu không thì ghép với `-base`.
- Dùng tập `visited` để tránh lặp vô hạn nếu gặp vòng lặp URL.
- Trong mỗi vòng lặp:
  - Gửi request đến trang hiện tại (qua Burp proxy nếu có).
  - In ra trạng thái `HTTP status code` và URL.
  - Gọi `find_flag()` để tìm flag trong response body.
  - Nếu chưa có flag, gọi `extract_next_url()` để lấy URL kế tiếp.
  - Kiểm tra *scope* bằng `urlparse()` để đảm bảo crawler không đi ra ngoài domain challenge.
  - Chờ một khoảng `delay` rồi tiếp tục vòng lặp.
- Nếu chạy đến giới hạn `-max` mà chưa thấy flag, script sẽ dừng và in thông báo tương ứng.

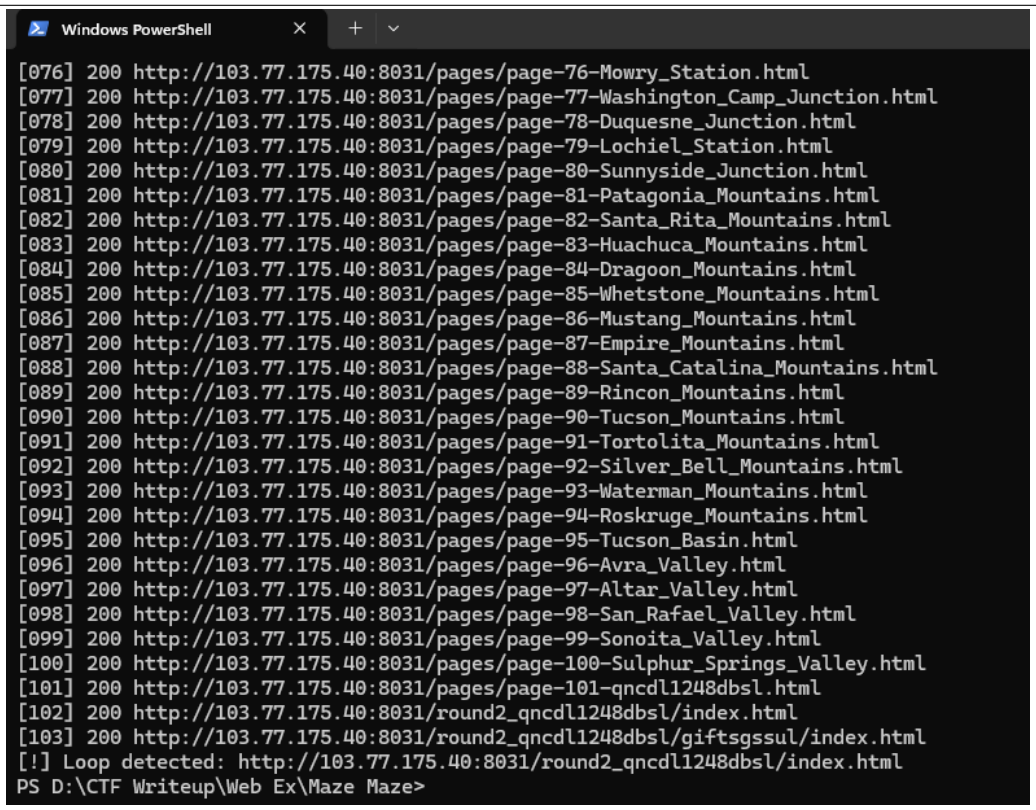
4) Khối `if __name__ == "__main__":` Đây là điểm khởi chạy của chương trình Python. Khi file được chạy trực tiếp, Python sẽ gọi `main()`. Cách viết này giúp tách phần "định nghĩa hàm" và phần "thực thi", đồng thời thuận tiện nếu sau này muốn import file này như một module.

5.1 Chạy file code Python và thành công bước vào round 2

Duc mở *Terminal* tại thư mục chứa file `maze_crawler_burp.py` và chạy lệnh sau:

```
python .\maze_crawler_burp.py -base http://103.77.175.40:8031 -start
/pages/page-1-Tombstone.html -max 200 -delay 0.02 -proxy "http://127.0.0.1:8080"
```

Sau khi chạy, Duc thu được kết quả như sau:



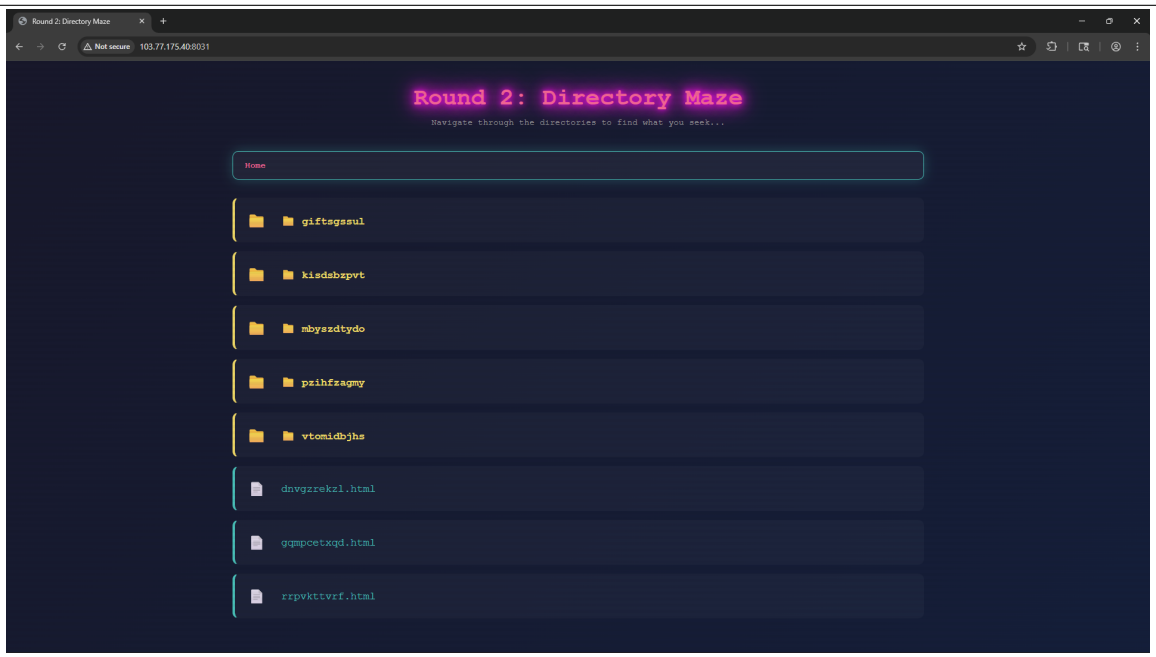
Hình 6: Kết quả chạy script `maze_crawler_burp.py` trong Terminal.

Dòng cuối của kết quả cho biết chương trình đã bị *loop detected*, đồng thời trả ra một đường link mới:

```
http://103.77.175.40:8031/round2_qncdl1248dbsl/index.html
```

Điều này cho thấy script đã lần theo maze thành công và chạm tới một nhánh mới của challenge.

Duc tiếp tục ném đường link này vào *Burp Suite* (hoặc mở trực tiếp trên trình duyệt) và thu được giao diện của **Round 2** như sau:



Hình 7: Giao diện Round 2 – Directory Maze sau khi lần theo maze thành công.

6 Giải mã Round 2

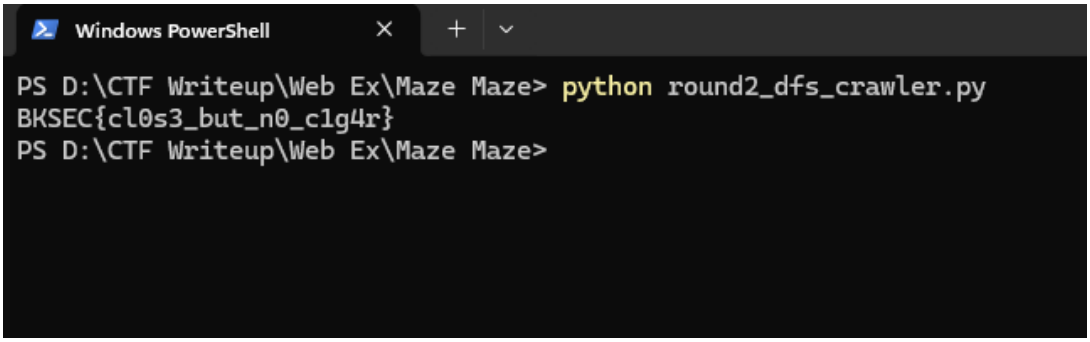
Theo như trang web gợi ý thì round này Duc cần phải đi vào từng URL có trong Round 2, sau đó kiểm tra phần *Response* để tìm *flag*. Nếu chưa thấy flag thì tiếp tục đi vào URL tiếp theo, lặp lại cho đến khi đã duyệt hết toàn bộ các URL có thể đi được.

Duc quyết định dùng **ChatGPT** để viết một file Python tự động thực hiện công việc này. Ở lần đầu, Duc chọn hướng duyệt **DFS** (*Depth-First Search*) để có thể lần theo cây thư mục và đi sâu vào từng nhánh.

Và đây là đoạn code *Python* mà ChatGPT đã viết:

```python
import re
import sys
import time
from urllib.parse import urljoin, urlparse

import requests
from bs4 import BeautifulSoup

START_URL = "http://103.77.175.40:8031/round2_qncdl1248dbs1/"
TIMEOUT_SEC = 20
DELAY_SEC = 0.0  # increase (e.g., 0.02) if you want to be gentler
MAX_PAGES = 200000  # safety cap

FLAG_RE = re.compile(r"BKSEC\{[^}]+\}")

SKIP_SCHEMES = ("javascript:", "mailto:", "tel:")
SKIP_HREFS = {"", "#", "./", "../", ".."}


def extract_links(html: str, base_url: str) -> list[str]:
    """
    Extract absolute URLs from <a href="..."> in HTML.
    Handles directory listings too.
    """
    soup = BeautifulSoup(html, "html.parser")
    links: list[str] = []

    for a in soup.select("a[href]"):
        href = (a.get("href") or "").strip()
        if not href or href in SKIP_HREFS:
            continue
        if href.lower().startswith(SKIP_SCHEMES):
            continue

        abs_url = urljoin(base_url, href)
        # Drop fragment
        abs_url = abs_url.split("#", 1)[0]
        links.append(abs_url)

    # De-duplicate while preserving order
    seen = set()
    uniq = []
    for u in links:
        if u not in seen:
            seen.add(u)
            uniq.append(u)
    return uniq


def in_scope(url: str, start_netloc: str) -> bool:
    p = urlparse(url)
    return p.scheme in ("http", "https") and p.netloc == start_netloc


def main() -> int:
    start_netloc = urlparse(START_URL).netloc
    session = requests.Session()
    session.headers.update(
        {
            "User-Agent": "Mozilla/5.0 (CTF DFS Crawler)",
            "Accept": "*/*",
            "Connection": "close",
        }
    )

    visited: set[str] = set()
    stack: list[str] = [START_URL]
    pages_fetched = 0

    while stack:
        url = stack.pop()  # DFS
        if url in visited:
            continue
        if not in_scope(url, start_netloc):
            continue
        visited.add(url)

        try:
            r = session.get(url, timeout=TIMEOUT_SEC, allow_redirects=True)
        except requests.RequestException:
            continue

        pages_fetched += 1
        if pages_fetched > MAX_PAGES:
            print("Not found flag")
            return 1
        text = r.text if isinstance(r.text, str) else ""
        m = FLAG_RE.search(text)
        if m:
            print(m.group(0))
            return 0
```

Duc chạy file `round2_dfs_crawler.py` và thu được kết quả như sau:



Hình 9: Kết quả chạy lần đầu: script tìm được một flag và in ra màn hình.

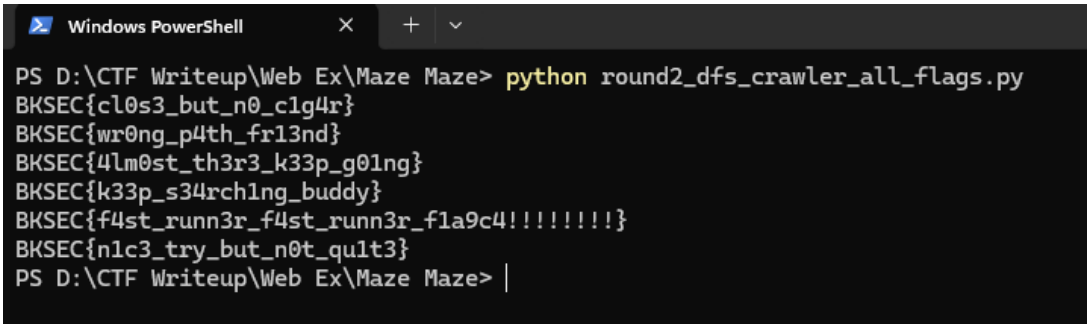
Tuy nhiên, khi Duc nhập flag mà chương trình trả về thì hệ thống báo *Incorrect*. Điều này cho thấy challenge này có **nhiều flag gây nhiễu**, chứ không phải chỉ có một flag duy nhất.

Vì vậy, Duc tiếp tục nhờ **ChatGPT** viết lại file Python. Điểm khác biệt là ở phiên bản mới, chương trình sẽ **không dừng ngay khi gặp flag đầu tiên**, mà sẽ tiếp tục duyệt hết các URL và **thu thập tất cả flag** xuất hiện, sau đó in ra *output* theo từng dòng.

Và đây là file Python mới:

```python
import re
import sys
import time
from urllib.parse import urljoin, urlparse

import requests
from bs4 import BeautifulSoup

START_URL = "http://103.77.175.40:8031/round2_qncdl1248dbsl/"
TIMEOUT_SEC = 20
DELAY_SEC = 0.0      # set e.g. 0.02 to be gentler
MAX_PAGES = 300000   # safety cap

FLAG_RE = re.compile(r"BKSEC\{[^}]+\}")

SKIP_SCHEMES = ("javascript:", "mailto:", "tel:")
SKIP_HREFS = {"", "#", "./", "../", ".."}


def extract_links(html: str, base_url: str) -> list[str]:
    """Extract absolute URLs from <a href="..."> in HTML (directory listings included)."""
    soup = BeautifulSoup(html, "html.parser")
    links: list[str] = []

    for a in soup.select("a[href]"):
        href = (a.get("href") or "").strip()
        if not href or href in SKIP_HREFS:
            continue
        if href.lower().startswith(SKIP_SCHEMES):
            continue

        abs_url = urljoin(base_url, href).split("#", 1)[0]
        links.append(abs_url)

    # De-duplicate while preserving order
    seen = set()
    uniq = []
    for u in links:
        if u not in seen:
            seen.add(u)
            uniq.append(u)
    return uniq


def in_scope(url: str, start_netloc: str) -> bool:
    p = urlparse(url)
    return p.scheme in ("http", "https") and p.netloc == start_netloc


def main() -> int:
    start_netloc = urlparse(START_URL).netloc

    session = requests.Session()
    session.headers.update(
        {
            "User-Agent": "Mozilla/5.0 (CTF DFS Crawler - collect flags)",
            "Accept": "*/*",
            "Connection": "close",
        }
    )

    visited: set[str] = set()
    stack: list[str] = [START_URL]  # DFS stack
    pages_fetched = 0

    found_flags: set[str] = set()
    found_order: list[str] = []  # keep discovery order for nicer output

    while stack:
        url = stack.pop()
        if url in visited:
            continue
        if not in_scope(url, start_netloc):
            continue
        visited.add(url)

        try:
            r = session.get(url, timeout=TIMEOUT_SEC, allow_redirects=True)
        except requests.RequestException:
            continue

        pages_fetched += 1
        if pages_fetched > MAX_PAGES:
            break

        text = r.text if isinstance(r.text, str) else ""

        # Collect ALL flags in this page (not just first)
        for m in FLAG_RE.finditer(text):
            flag = m.group(0)
            if flag not in found_flags:
                found_flags.add(flag)
                found_order.append(flag)

        # Only parse links if it looks like HTML
        ct = (r.headers.get("Content-Type") or "").lower()
        if ("text/html" in ct) or ("<html" in text.lower()):
            links = extract_links(text, r.url)

            # Reverse to keep a more natural DFS order
            for nxt in reversed(links):
                if in_scope(nxt, start_netloc) and nxt not in visited:
```

Sau khi chạy file Python mới, Duc thu được kết quả như sau:



Hình 11: Kết quả chạy phiên bản DFS thu thập toàn bộ flag trong Round 2.

Duc nhập thử từng flag được in ra, và cuối cùng xác định được **flag đúng** của bài này là:

`BKSEC{f4st_runn3r_f4st_runn3r_f1a9c4!!!!!!!!}`

Problem solved!