TARUMT Hackathon 2024

Lucky Team

Team Members:

Tan Yin Wun

Wong Mei Jing

Ashley Law Jia Ye

Problem Statement:

In the digital era, web application security is critical to safeguarding sensitive data. However, identifying vulnerabilities can be challenging sometimes. This project aims to develop a web application security scanner that automates vulnerability detection and provides actionable recommendations for developers. By empowering developers with tools to strengthen security, we aim to reduce the potential risk of data breaches before any serious effects are caused.

Introduction

Data Leaking Risk Scanner is a web app to determine the web page's data leaking risk by scanning the URL that users provide and detect if it has any vulnerability or not.

We do a quick review on some web vulnerability scanner on the internet such as Mozilla Observatory and Google Security Scanner as a reference to gather information on what are those vulnerabilities that we need to detect and how can we develop a system to do it.

We also did some research on OWASP to learn more about cybersecurity before developing our web app, and finally, we decided to check the vulnerability of the webpage from several common aspects. Due to the limitations of time, we're not able to test all the aspects of the security assessment listed in OWASP manual so we only choose a few common tests to check the vulnerability of the webpage.

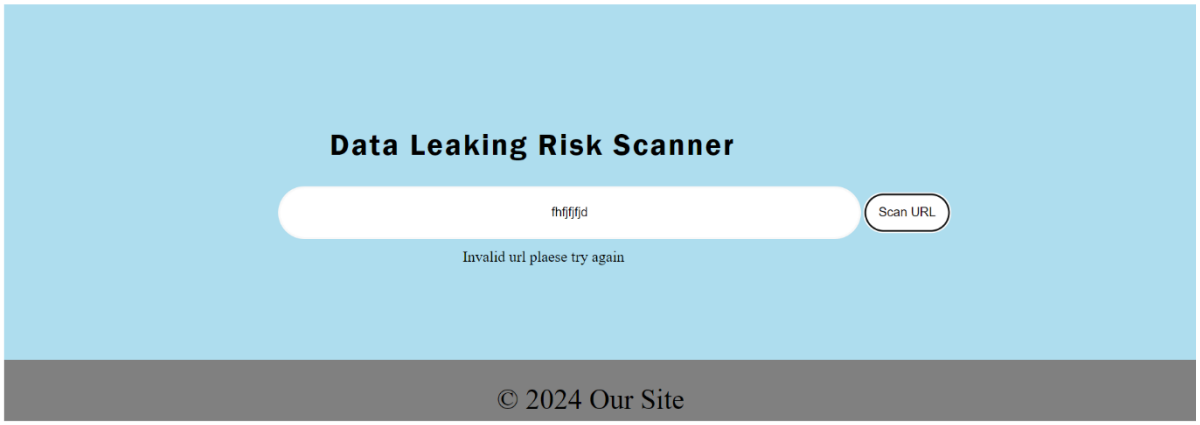
First it will scan the URL to check if it is a valid URL or not, if it is, it will test the clickjack vulnerability, SQL injection vulnerability, implementation of authentication and authorization mechanism, existence of cookies and third-party integration, and also the validation of hostname and scheme. We use Python backend and Flask model to connect with the JavaScript frontend to complete this project.

Objectives

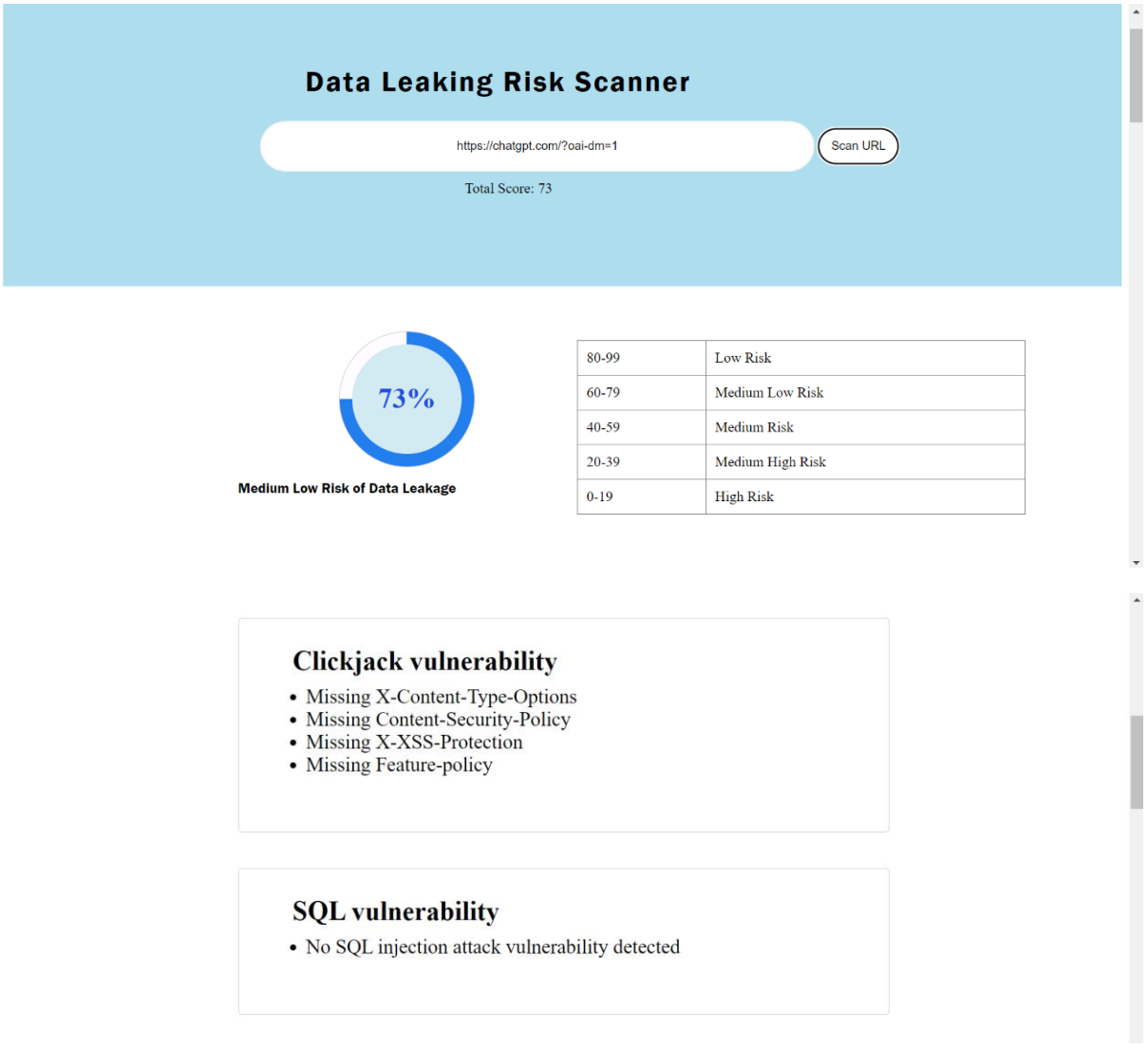
- Develop a comprehensive web application security scanner capable of identifying common vulnerabilities.
- Offer actionable recommendations and best practices for remediation to empower developers to address identified vulnerabilities effectively.
- Enhance the user experience by providing intuitive interfaces for configuring scans, viewing results, and accessing remediation guidance.
- Measure the effectiveness of the security scanner in reducing the incidence of security vulnerabilities and mitigating the risk of data breaches in web applications.

User interface

If invalid URL:



If valid URL:



Hostname vulnerability

- chatgpt.com is valid hostname

Scheme vulnerability

- Secure scheme

Authentication vulnerability

- No authentication required

Authorization vulnerability

- No Authorization Required

Cookies vulnerability

- No normal cookies detected

3rd party integration vulnerability

- There are 1 third party cookies detected
- No request to external domain detected

Tips for Developer

- Add on missing clickjack headers
- Add on authentication process when login or sign up
- Provide Authorization Process whenever need to access other resources
- Manage Third Party Cookie Settings to reduce the risk of data leakage
- Conduct a Security Assessment
- Monitor for Security Incidents in real-time

Tips for User

- Be Cautious with Clicks, input fields or even when accessing the website
- Use Strong, Unique Passwords
- Be Wary of Phishing Attempts
- You may choose to block third-party cookies or delete cookies periodically
- Consider using security tools such as web security suites
- Consider installing reputable browser extensions that offer privacy-enhancing features
- Avoid Providing Sensitive Information

Coding part

Clickjack: we check for some common headers of security measurements to prevent clickjacking vulnerabilities using a Boolean array to return value to the front end and tell the user which headers are missing there.

```
def clickjack(url):
    bool_array = [False, False,False,False,False,False,False]
    clickjackScore=0
    try:
        response = requests.get(url)
        if "Strict-Transport-Security" in response.headers:
            bool_array[0]= True
            clickjackScore+=2
        if "X-Content-Type-Options" in response.headers:
            bool_array[1]= True
            clickjackScore+=2
        if "Content-Security-Policy" in response.headers:
            bool_array[2]= True
            clickjackScore+=2
        if "X-Frame-Options" in response.headers:
            bool_array[3]= True
            clickjackScore+=2
        if "X-XSS-Protection" in response.headers:
            bool_array[4]= True
            clickjackScore+=2
        if "Referrer-Policy" in response.headers:
            bool_array[5]= True
            clickjackScore+=2
        if "Feature-policy" in response.headers:
            bool_array[6]= True
            clickjackScore+=2

    except requests.RequestException as e:
        print("Error making HTTP request:", e)

    return bool_array,clickjackScore
```

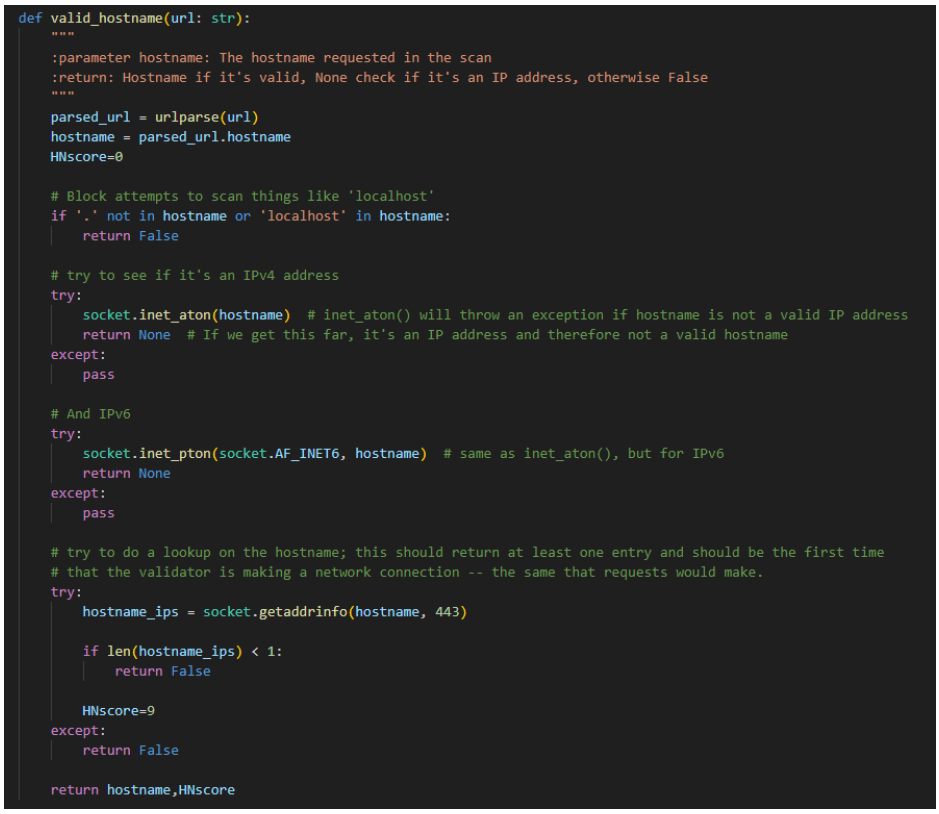
Scheme: we parsed the URL and checked if the scheme was empty. An empty scheme usually indicates a relative URL. If the scheme is empty, it considers it a relative URL else it considers it a relative protocol. We also check if it is secure with https scheme.

```
def valid_scheme(url):
    parsed_url = urlparse(url)
    scheme = parsed_url.scheme
    schemeScore=0
    if scheme == '':
        if url.netloc == '':
            # Relative URL (src="/path")
            relativeorigin = True
        else:
            # Relative protocol (src="//host/path")
            relativeorigin = False
    else:
        relativeorigin = False

    # check if it's a secure scheme
    if scheme == 'https' or (relativeorigin and scheme == 'https'):
        securescheme = True
        schemeScore=8
    else:
        securescheme = False

    return securescheme,schemeScore
```

Hostname: we check if it is an IP address or a valid hostname, if it is IP address or localhost it will return false which means it is an invalid hostname.



```python
def valid_hostname(url: str):
    """
    :parameter hostname: The hostname requested in the scan
    :return: Hostname if it's valid, None check if it's an IP address, otherwise False
    """
    parsed_url = urlparse(url)
    hostname = parsed_url.hostname
    HNscore=0

    # Block attempts to scan things like 'localhost'
    if '.' not in hostname or 'localhost' in hostname:
        return False

    # try to see if it's an IPv4 address
    try:
        socket.inet_aton(hostname)  # inet_aton() will throw an exception if hostname is not a valid IP address
        return None  # If we get this far, it's an IP address and therefore not a valid hostname
    except:
        pass

    # And IPv6
    try:
        socket.inet_pton(socket.AF_INET6, hostname)  # same as inet_aton(), but for IPv6
        return None
    except:
        pass

    # try to do a lookup on the hostname; this should return at least one entry and should be the first time
    # that the validator is making a network connection -- the same that requests would make.
    try:
        hostname_ips = socket.getaddrinfo(hostname, 443)

        if len(hostname_ips) < 1:
            return False

        HNscore=9
    except:
        return False

    return hostname,HNscore
```

SQL injection: we check the details of the form in HTML and try to get a response from HTTP request to determine if each form has SQL injection vulnerability.

```python
def sql_injection_scan(urlToBeChecked):
    forms = get_forms(urlToBeChecked)
    print(f"[+] Detected {len(forms)} forms on {urlToBeChecked}.")

    sqlScore = 17  # set sqlScore=0

    for form in forms:
        details = form_details(form)

        for i in "\"'":
            data = {}
            for input_tag in details["inputs"]:
                if input_tag["type"] == "hidden" or input_tag["value"]:
                    data[input_tag['name']] = input_tag["value"] + i
                elif input_tag["type"] != "submit":
                    data[input_tag['name']] = f"test{i}"

            form_details(form)

            # Make the request with modified input data
            # to avoid res has no value
            res = None
            try:
                if details["method"] == "post":
                    res = s.post(urlToBeChecked, data=data)
                elif details["method"] == "get":
                    res = s.get(urlToBeChecked, params=data)
            except Exception as e:
                print(f"Error occurred during request: {e}")

            if res is not None:
                if vulnerable(res) and sqlScore>0:
                    print("SQL injection attack vulnerability in link: ", urlToBeChecked)
                    sqlScore -= 12
                else:
                    print("No SQL injection attack vulnerability detected")
                    break

    return sqlScore
```

Authentication and authorization: we test the authentication mechanism by testing the status code, authentication-related headers and meta tags from the response of the HTTP request. For checking the authorization mechanism, we use the same method to check its authorization-related meta tags.

```python
def test_authentication(url):
    authen=False
    try:
        # Send a GET request to a protected endpoint
        response = requests.get(url)

        # Check the response status code
        if response.status_code == 401:
            authen=True
            print("Authentication is required. The website has an authentication process.")
        elif response.status_code == 200:
            print("No authentication required. The website does not have an authentication process.")
        else:
            print("Unexpected response status code:", response.status_code)
    except requests.exceptions.RequestException as e:
        print("An error occurred:", e)

    return authen
```

```python
def analyze_authentication_layers(url):
    authen_header=False
    try:
        # Send a GET request to the specified URL
        response = requests.get(url)

        # Check for authentication-related headers
        if 'WWW-Authenticate' in response.headers:
            authen_header=True
            print("The website requires authentication.")
            # You can further analyze the contents of the 'WWW-Authenticate' header
            # to gather information about the authentication mechanism.
        else:
            print("No authentication headers found. The website may not require authentication.")
    except requests.exceptions.RequestException as e:
        print("An error occurred:", e)

    return authen_header
```

```python
def check_authentication_meta_tags(url):
    authen_meta=False
    try:

        response = requests.get(url)
        response.raise_for_status()  # Raise an exception for HTTP errors

        # Parse the HTML content of the response
        soup = BeautifulSoup(response.content, 'html.parser')

        meta_tags = soup.find_all('meta')

        # Check for meta tags containing authentication-related information
        authentication_meta_tags = [tag for tag in meta_tags if 'authentication' in tag.get('name', '').lower()]

        # Print the authentication-related meta tags
        if authentication_meta_tags:
            print("Authentication-related meta tags found:")
            for tag in authentication_meta_tags:
                print(tag)
            authen_meta=True
        else:
            print("No authentication-related meta tags found.")

        return authen_meta

    except requests.exceptions.RequestException as e:
        print("An error occurred while fetching the URL:", e)
    except Exception as e:
        print("An error occurred:", e)
```

Cookies: we try to get cookies from HTTP requests and count how many of them because the more cookies it has, means that it may have more potential risk of data leaking due to cookies capturing users' information.

```python
def check_for_cookies(url):
    try:
        # Send a GET request to the specified URL
        response = requests.get(url)

        # Print information about the cookies
        cookies = response.cookies

        # Initialize score
        total_score = 17

        # Count the number of normal (first-party) cookies
        normal_cookie_count = Counter(cookie.domain == response.url.split('//')[1].split('/')[0] for cookie in cookies)

        if normal_cookie_count[True] > 0:
            print(f"Number of normal (first-party) cookies found: {normal_cookie_count[True]}")
            total_score -= 1  # Deduct 1 if any first-party cookie found
        else:
            print("No normal (first-party) cookies found.")
            total_score = 17

        return total_score,normal_cookie_count[True]
    except requests.exceptions.RequestException as e:
        print("An error occurred while fetching the URL:", e)
        return 0
    except Exception as e:
        print("An error occurred:", e)
        return 0
```

Third-party integration: Same as cookies, the more third-party integration of a website, the higher the potential risk of data leaking. So we check for third-party cookies and request to external domain to determine the number of third-party integrations.

```python
def analyze_website(url):
    request_count=0
    thirdCookies=0
    try:
        response = requests.get(url)
        external_requests = find_external_requests(response)
        third_party_cookies = check_cookies(response)

        total_score = 17

        if external_requests:
            for request_url in external_requests:
                print(request_url)
                request_count+=1
                total_score -= 1 # Deduct 1 for external requests
        else:
            print("No external domain found.")
            total_score = 17

        if third_party_cookies:
            print("Third-party cookies found:")
            for cookie in third_party_cookies:
                print(cookie)
                thirdCookies+=1
                total_score -= 1  # Deduct 1 for third-party cookies
        else:
            print("No cookies found.")
            total_score =17

        # Print the final deducted score
        return total_score,request_count,thirdCookies

    except requests.exceptions.RequestException as e:
        print("An error occurred while fetching the URL:", e)
    except Exception as e:
        print("An unexpected error occurred:", e)
```

Flask: we used Flask to connect the Python backend to the JavaScript frontend by passing all the variables returned from those functions. We just need to run the app.py and go to “localhost:5000” then we can see the user interface and start testing.

```python
@app.route('/test', methods=['POST'])
def test():
    output = request.get_json()
    print(output) # This is the output that was stored in the JSON within the browser
    url = output.get('url', '')
    error=scan_url(url)
    #avoid getting invalid url
    if(error):
        return jsonify(error=error)
    else:
        boolArr=clickjack(url)[0]
        CJScore=clickjack(url)[1]
        hostName=valid_hostname(url)[0]
        hnScore=valid_hostname(url)[1]
        scheme=valid_scheme(url)[0]
        schemeScore=valid_scheme(url)[1]
        auth_header=analyze_authentication_layers(url)
        auth_meta=check_authentication_meta_tags(url)
        aaScore=calculate_authentication_score(url)
        authorize=check_authorization_meta_tags(url)[1]
        sqlScore=sql_injection_scan(url)
        cookiesScore=check_for_cookies(url)[0]
        cookiesNum=check_for_cookies(url)[1]
        thirdPartyScore=analyze_website(url)[0]
        thirdPartyRequest=analyze_website(url)[1]
        thirdPartyCookies=analyze_website(url)[2]
        total=CJScore+hnScore+schemeScore+aaScore+sqlScore+cookiesScore+thirdPartyScore
        print(total)
        return jsonify(boolArr=boolArr,CJScore=CJScore,sqlScore=sqlScore,hnScore=hnScore,hostName
                       cookiesNum=cookiesNum,thirdPartyCookies=thirdPartyCookies,thirdPartyRequest=t
```

Conclusion

In conclusion, we created a simple basic data leaking risk scanner in this short period of time that functions properly and can get accurate results. We’ve learned a lot throughout the whole process of developing this web app and have a clear understanding of cybersecurity technologies to detect and prevent data leaking. Data leaking is always a hot topic to be solved on the internet and we hope our web app can contribute to society by solving a part of the issue. By creating a tool that empowers users to detect and mitigate data leakage risks, we aspire to make a meaningful contribution to cybersecurity efforts.