Bruteforcing Cookies

When the HTTP protocol was born, there was no need to track connection states. A user requested a resource, the server replied, and the connection was closed. It was 1991, and websites were quite different from what we are used to today. As you can imagine, almost no modern web application could work this way because they serve different content based on who requests it. A shopping cart, preferences, messages, etc., are good examples of personalized content. Fortunately, while developing the first e-commerce application, the precursor of WWW wrote a new standard, `cookies`.

Back then, a cookie, sent as a header at each request from the browser to the web application, was used to hold all user session details, such as their shopping cart, including chosen products, quantity, and pricing. Issues emerged very soon. The main concern nowadays is security. We know that one cannot trust the client when it comes to authorizing a modification or a view, but back then, the problem was also regarding the request's size. Cookies then started to be lighter and be set as a unique identifier that refers to a `session` stored on the server-side. When a visitor shows their cookies, the application checks any details by looking at the correct session on the server-side.

While we know that a web application could set many cookies, we also know that usually, `one or two` are relevant to the session. Session-related cookies are used to "discriminate" users from each other. Other cookies could be related to language, information about acceptance of Privacy, or a cookie disclaimer, among others. These cookies could be altered with no significant impact since they are not related to application security.

We know that other ways also exist to track users, for example, the already discussed `HTTP Authentication` or an in-page token like `ViewState`. `HTTP Authentication` is not common on Internet-facing applications, at least not as the primary layer for authentication. However, it could be the proper security barrier if we want to protect a web application `before` a user even reaches the login form.

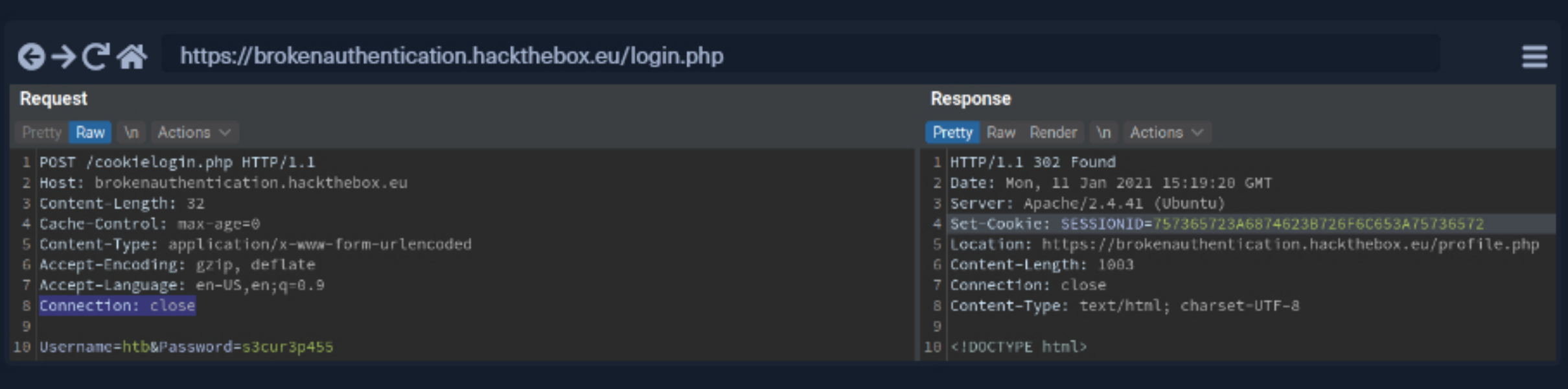
ViewState is an excellent example of an in-page security token, used by default by `.NET` web applications like any SharePoint site. `ViewState` is included as a hidden field in HTML forms. It is built as a serialized object containing useful information about the current user/session (where the user came from, where the user can go, what the user can see or modify, etc.) A `ViewState token` could be easily decoded if it is not encrypted. However, the main concern is that it could suffer from a vulnerability that leads to remote code execution even if it is encrypted. Session cookies can suffer from the same vulnerabilities that may affect password reset tokens. They could be predictable, broken, or forged.

Cookie token tampering

Like password reset tokens, session tokens could also be based on guessable information. Often, homebrewed web applications feature custom session handling and custom cookie-building mechanisms to have user-related details handy. The most common piece of data we can find in cookies is user grants. Whether a user is an admin, operator, or basic user, this is information that can be part of the data used to create the cookie.

Unfortunately, as already discussed, it is not rare to see tokens generated from important values, such as userid, grants, time, etc. Imagine a scenario where part of a web application's functionality is to get back the plaintext from an encoded/encrypted value. This means that one-way encryption such as hashing is out of the question. Of course, there is no real reason to store data in session cookies because everything that is part of the session should be handled and validated server-side, and their content should be completely random.

Let us consider the following example.



Line 4 of the server's response shows a `Set-Cookie` header that sets a `SESSIONID` to `757365723A6874623B726F6C653A75736572`. If we decode this hex value as ASCII, we see that it contains our `userid`, `htb`, and role (standard `user`).

```
Govardhan Gujji22@htb[/htb]$ echo -n 757365723A6874623B726F6C653A75736572 | xxd -r -p; echo

user:htb;role:user
```

We could try escalating our privileges within the application by modifying the `SESSIONID` cookie to contain `role:admin`. This action may even allow us to bypass authentication altogether.

Remember me token

We could consider a `rememberme` token as a session cookie that lasts for a longer time than usual. `rememberme` tokens usually last for at least seven days or even for an entire month. Given their long lifespan, `rememberme` tokens could be easier to brute force. If the algorithm used to generate a `rememberme` token or its length is not secure enough, an attacker could leverage the extended timeframe to guess it. Almost any attack against password reset tokens and generic cookies can also be mounted against `rememberme` tokens, and the security measures a developer should put in place are almost the same.

Encrypted or encoded token

Cookies could also contain the result of the encryption of a sequence of data. Of course, a weak crypto algorithm could lead to privilege escalation or authentication bypass, just like plain encoding could. The use of weak crypto will slow an attacker down. For example, we know that ECB ciphers keep some original plaintext patterns, and CBC could be attacked using a padding oracle attack.

Given that cryptography attacks require some math basics, we have developed a dedicated module rather than overflowing you with too many extras here.

Often web applications encode tokens. For example, some encoding algorithms, such as hex encoding and base64, can be recognized by a trained eye just by having a quick look at the token itself. Others are more tricky. A token could be somehow transformed using, for example, XOR or compression functions before being encoded. We suggest always checking for magic bytes when you have a sequence of bytes that looks like junk to you since they can help you identify the format. Wikipedia has a list of common file signatures to help us with the above.

Take this recipe at CyberChef. The token is a valid base64 string, that results in a set of apparently useless hex bytes. Magic bytes are `1F 8B`, a quick search on Wikipedia's file signatures page indicates that it could be a gzipped text. By pausing `To hex` and activating `Gunzip` inside the CyberChef recipe we just linked, we can see that it is indeed gzipped content.

As said, a trained eye will probably spot encoders just by looking at the string. Decodify is a tool written to automate decode guessing. It doesn't support many algorithms but can easily help to spot some basic ones. CyberChef offers a massive list of decoders, but they should be used manually and checked one at a time.

Start with a basic example on this recipe at CyberChef. The recipe start with component paused so we can do one step at time.

Encoded text contains chars that don't look like printable ASCII hex code (ASCII printable) that could be printed on a terminal and is in the range from 0x20 to 0x7f). We can see some 0x00 and 0x09 that are outside the printable range. We, therefore, should exclude any Base* algorithm, like Base64 or Base32, because they are built with a subset of printable ASCII. The string still looks like a hex-encoded one, so we can try to force a *From hex* operation by un-pausing the first block by clicking the pause button to grey it out. As expected, all we have is junk. Inspecting the string, we can see that it starts with *42 5a*. Checking Wikipedia List of file signatures page looking for those two bytes, also called *Magic bytes*, we see that they refer to the bzip algorithm. Having found a possible candidate, we un-pause. Second step: Bzip2 decompress. The resulting string is *ZW5jb2Rpbmdfcm94* and doesn't tell us much: there could be another encoding step. We know that when there is the need to move data to and from a web application, a very common encoder is Base64, so we try to give it a shot.

Search for *Base64* in the upper-left search form and drag&drop *From Base64* to our recipe: we have our string decoded, as you can see.

To become more comfortable with CyberChef, we suggest practicing by encoding with different algorithms and reversing the flow.

Sometimes cookies are set with random or pseudo-random values, and an easy decode doesn't lead to a successful attack. That's why we often need to automate the creation and test of such cookies. Assume we have a web app that creates persistent cookies starting from the string *user_name:persistentcookie:random_5digit_value*, then encodes as base64 apply ROT13 and converts to hexadecimal to stores it in a database so it could be checked later. And assume we know, or suspect, that `htbadmin` used a persistent cookie. ROT13 is a special case of Caesar Cypher where every char is rotated by 13 positions. It was quite commonly used in the past, but even though it's almost dead nowadays, it is an interesting alternative to bz2 compression for this example.

Even if the space of random values is very small, a manual test is out of the question. Therefore we created a Python proof of concept to show the possible flow to automate this type of attack. Download automate_cookie_tampering.py script, read and understand the code.

Often when developers think about encryption, they see it as a strong security measure. Still, in this case they miss the context: given that there is really no need to store data in session cookies and they should be pure random, there is no need to encrypt or encode them.

Weak session token

Even when cookies are generated using strong randomization, resulting in a difficult-to-guess string, it could be possible that the token is not long enough. This could be a problem if the tested web application has many concurrent users, both from a functional and security perspective. Suppose space is not enough because of the Birthday paradox. In that case, two users might receive the same token. The web application should always check if a newly generated one already exists and regenerate it if this is the case. This behavior makes it easier for an attacker to brute force the token and obtain a valid one.

Following the example we saw when talking about `Short token` on the `Predictable reset token` section, we could try to brute force a session cookie. The time needed would depend on the length and the charset used to create the token itself. Given this is a guessing game, we think a truly incremental approach that starts with `aaaaaa` to `zzzzzz` would not pay dividends. That is why we prefer to use `John the Ripper`, which generates non-linear values, for our brute-forcing session.

We should examine some cookie values, and after having observed that the length is six and the charset consists of lowercase chars and digits, we can start our attack.

Wfuzz can be fed by another program using a classic pipe, and we will use `John` as a feeder. We set `John` in incremental mode, using the built-in "LowerNum" charset that matches our observation (`--incremental=LowerNum`), we specify a password length of 6 chars (`--min-length=6 --max-length=6`), and we also instruct `John` to print the output as stdout (`--stdout`). Then, `wfuzz` uses the payload from stdin (`-z stdin`) and fuzzes the "HTBSESS" cookie (`-b HTBSESS=FUZZ`), looking for the string `"Welcome"` (`--ss "Welcome"`) in server responses for the given URL.

```
Govardhan Gujji22@htb[/htb]$ john --incremental=LowerNum --min-length=6 --max-length=6 --stdout| wfuzz -z

********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: https://brokenauth.hackthebox.eu/
Total requests: <<unknown>>

===================================================================
ID           Response   Lines    Word     Chars       Payload
===================================================================

Created directory: ~/john-the-ripper/297/.john
Press 'q' or Ctrl-C to abort, almost any other key for status
000009897:   200        9 L      31 W     274 Ch      "abaney"
```

This attack could take a long time, and it is infeasible if the token is lengthy. Chances are higher if cookies last longer than a single session, but do not expect a quick win here. We encourage you to practice using a PHP file you can download here. Please read the file carefully and try to make it print the congratulations message.

Start Instance

∞ / 1 spawns left

Waiting to start...

Questions

Cheat Sheet

Answer the question(s) below to complete this Section and earn cubes!

Target: Click here to spawn the target system!

+ 2 Tamper the session cookie for the application at subdirectory /question1/ to give yourself access as a super user. What is the flag?

Submit your answer here... Submit No Hint

Authenticate to with user "htbuser" and password "htbuser"

+ 2 Log in to the target application and tamper the rememberme token to give yourself super user privileges. After escalating privileges, submit the flag as your answer.

Submit your answer here... Submit Hint

Previous Next

Cheat Sheet

Resources

Go to Questions

Table of Contents

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left