Electronic Code Book

Saikat Karmakar | Sept 4 : 2021

Introduction

This course details the exploitation of a weakness in the authentication of a PHP website. The website uses ECB to encrypt information provided by users and use this information to ensure authentication. We will see how this behaviour can impact the authentication and how it can be exploited.

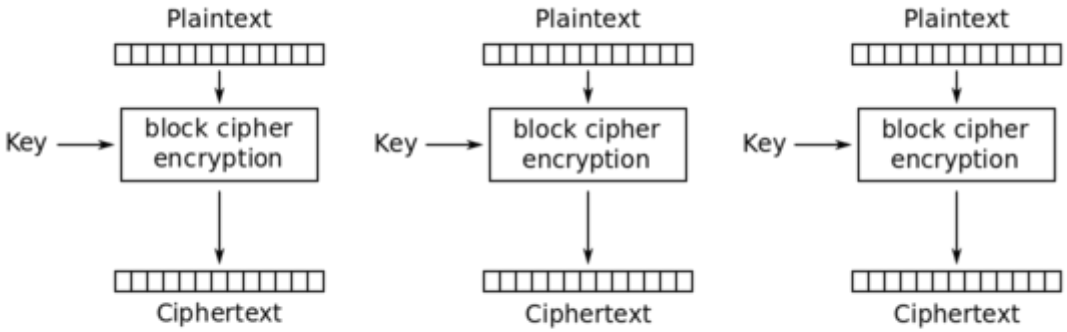
If you feel confident, you can try to do this exercise without following the course, then you can come back to the course to read some details and tips. If you want to do it by yourself, you can follow the following steps:

- create 2 users with similar names: `test1` and `test2` and the same password `password`, then look at the cookie sent back by the application.
- create a user with a really long name composed of the same character (like 20 times `a`) and look at the cookie sent back by the application. Try to look for a pattern that will give you more information on how the information can be manipulated.
- try to create a user with a username and password that will allow you to be logged in as `admin` by removing encrypted data.
- try to create a user with a username that will allow you to be logged in as `admin` by swapping encrypted blocks around.

ECB

ECB is an encryption mode in which the message is split into blocks of X bytes length and each block is encrypted separately using a key.

The following schema (source: Wikipedia) explains this method:



Electronic Codebook (ECB) mode encryption

You can check the recent XKCD on the Adobe's password leak to get an humoristic idea of the problems tied to ECB.

During the decryption, the reverse operation is used. Using ECB has multiple security implications:

- Blocks from encrypted message can be removed without disturbing the decryption process.
- Blocks from encrypted message can be moved around without disturbing the decryption process.

In this exercise, we will see how we can exploit these two weaknesses.

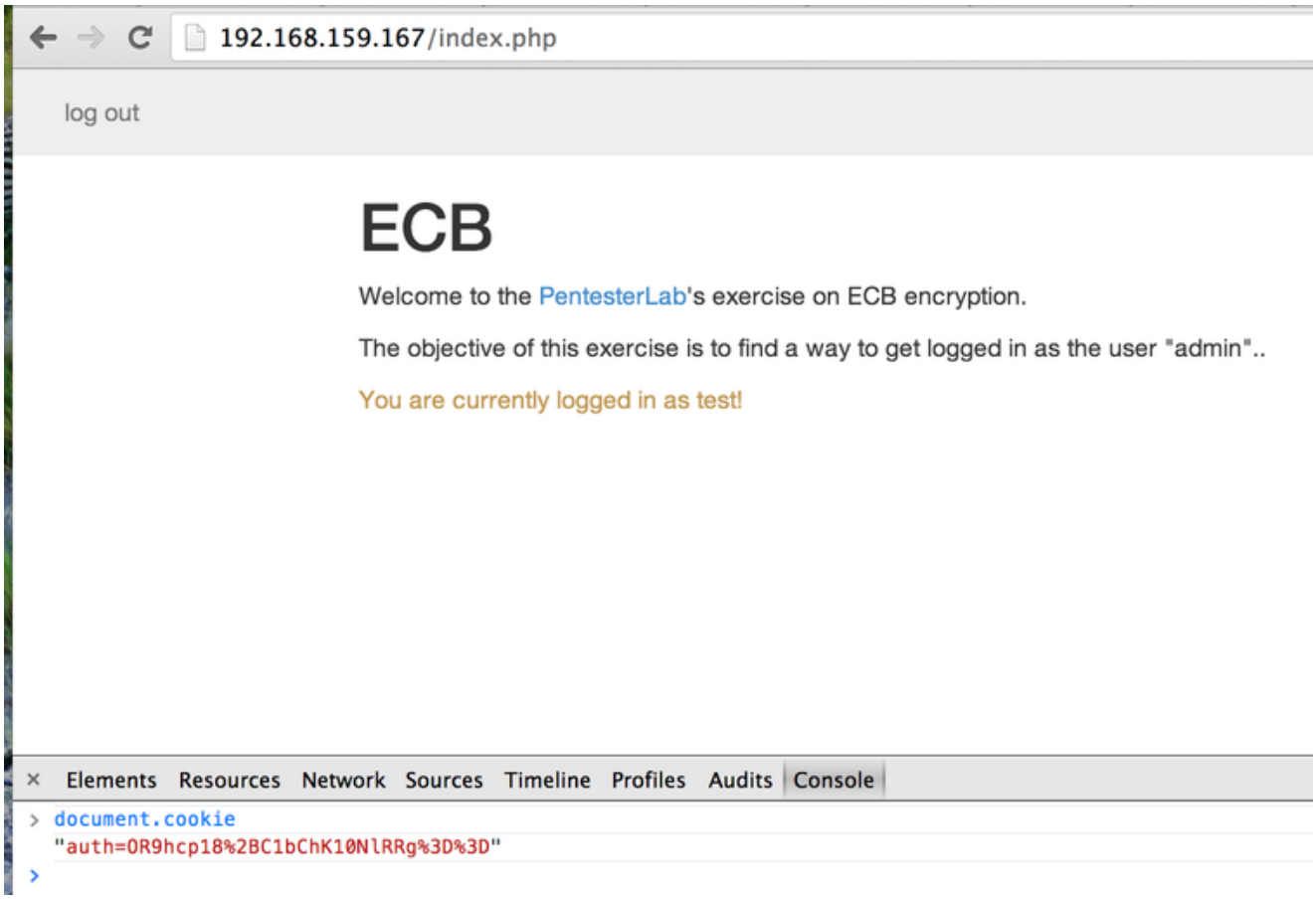
Detection of the vulnerability

In this exercise, you can register an account and log in with this account (to make things easier, you get automatically logged in when you register).

If you create an account and log in two times with this account, you can see that the cookie sent by the application didn't change.

If you log in many times and always get the same cookie, there is probably something wrong in the application. The cookie sent back should be unique each time you log in. If the cookie is always the same, it will probably always be valid and there won't be anyway to invalidate it.

If we look at the cookie, we can see that it seems uri-encoded and base64-encoded:



The 2 equals sign encoded as `%3d%3d` are a good indicator of base64-encoded string.

We can decode it using the following ruby code:

```
% irb
> require 'base64' ; require 'uri'
 => true
```

```ruby
> Base64.decode64(URI.decode("OR9hcp18%2BC1bChK10NlRRg%3d%3d"))
 => "9\x1Far\x9D|\xF8-[\n\x12\xB5\xD0\xD9QF"
```
language-ruby

Or by decoding the URI to a string manually and use the base64 command:

```bash
% echo "OR9hcp18+C1bChK10NlRRg==" | base64 -D | hexdump -C
0000000 39 1f 61 72 9d 7c f8 2d 5b 0a 12 b5 d0 d9 51 46  |9.ar.|.-[.....QF|
0000010
```
language-bash

On osX, the command `base64 -D` replaces `base64 -d`

In both cases, we can see that the information seems to be encrypted.

First, we can start by creating two accounts `test1` and `test2` with the same password: `password` and compare the cookies sent by the application. We get the following cookies (after URI-decoding):

Account:	test1	test2
Cookie:	vHMQ+Nq9C3MHT8ZkGeMr4w==	Mh+JMH1OMhcHT8ZkGeMr4w==

f we base64-decode both cookies, we get the following strings:

Account:	test1	test2
Decoded cookie:	\xBCs\x10\xF8\xDA\xBD\vs\aO\xC6d\x19\xE3+\xE3	2\x1F\x890}N2\x17\aO\xC6d\x19\xE3+\xE3

We can see that part of the decrypted values look really similar.

Now we can try to create a user with an arbitrary long username and password. For example, a username composed of 20 `a` and a password composed of 20 `a`. By creating this user, we get the following cookie:

```
document.cookie
"auth=GkzSM2vKHdcaTNIza8od1wS28inRHiC2GkzSM2vKHdcaTNIza8od1ys96EXmirn5"
```

If we decode this value, we get the following value:

```
\x1AL\xD23k\xCA\x1D\xD7\x1AL\xD23k\xCA\x1D\xD7\x04\xB6\xF2)\xD1\x1E
\xB6\x1AL\xD23k\xCA\x1D\xD7\x1AL\xD23k\xCA\x1D\xD7+=\xE8E\xE6\x8A\xB9\xF9
```

We can see that the following pattern (composed of 8 bytes): `\x1AL\xD23k\xCA\x1D\xD7` comes back multiple times:

\x1AL\xD23k\xCA\x1D\xD7\x1AL\xD23k\xCA\x1D\xD7\x04\xB6\xF2)\xD1\x1E
\xB6**\x1AL\xD23k\xCA\x1D\xD7**\x1AL\xD23k\xCA\x1D\xD7**+=\xE8E\xE6\x8A\xB9\xF9

Based on the size of the pattern, we can infer that the ECB encryption uses a block size of 8 bytes.

This example is using a weak encryption mechanism and it's likely that real life examples will use bigger block size.

The decoded information also shows us that the username and password are not directly concatenated and that a delimiter is added (since one of the block in the middle is different from the previous one).

We can think of the encrypted stream has one of the two following possibilities:

- The stream contains the username, a delimiter and the password:

- The stream contains the password, a delimiter and the username:

By creating another user with a long username and a short password, we can see that the following pattern is used: `username|delimiter|password`.

Now let's try to find the size of the delimiter, if we play with different size of username and password we get the following results:

Username length:	Password length:	Username+Password length:	Cookie's length (after decoding):
2	3	5	8
3	3	6	8
3	4	7	8
4	4	8	16
4	5	9	16

We can see that the size of the decoded cookie goes from 8 to 16 bytes when the length of the Username+Password is greater than 7. We can infer from this value that the delimiter is a single byte since the encryption is done per block of 8 bytes.

Another important thing is to see what part of the encrypted stream is used by the application when we send the cookie back. If we remove everything after the block corresponding to the delimiter, we can see that we are still authenticated. The password does not seem to be used when the cookie gets used by the application.

We now know that we just need to get the correct `username|delimiter` to get authenticated within the application as `username`.

If you can find what delimiter is used (or brute force it), you can try to create a user with a username that contains the delimiter (for example the username "`admin:`"). Using this method, you may be able to get logged in as `admin`. This web application prevents this type of attack.

Exploitation of the vulnerability

By removing information

The easiest way to get `admin` access is to remove some of the encrypted data. We know that the application uses the following format:

```
\[username\]:\[separator\]
```

and only uses the `username` when the cookie is sent back to the application. We also know that each block of 8 bytes is completely independent (ECB). To exploit this issue, we can create a username that contains 8 characters followed by the word `admin`:

aaaaaaaaadmin

And we will receive the cookie (retrieved using the Javascript Console):

```
document.cookie
"auth=GkzSM2vKHdfgVmQuKXLregdPxmQZ4yvj"
```

This value will get decoded as:

```
\x1AL\xD23k\xCA\x1D\xD7\xE0Vd.)r\xEBz\aO\xC6d\x19\xE3+\xE3
```

We can see the pattern `\x1AL\xD23k\xCA\x1D\xD7` detected previously with the username that contained 20 `a`.

We can then remove the first 8 bytes of information and reencode our payload to get a new cookie:

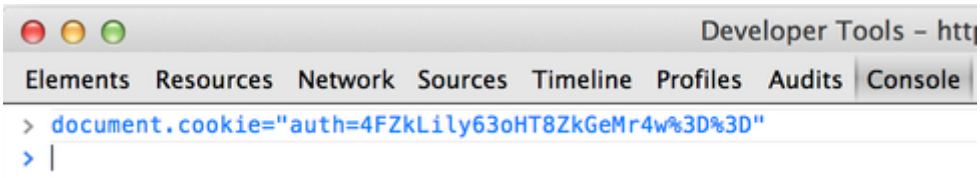
\xE0Vd.)r\xEBz\aO\xC6d\x19\xE3+\xE3

That will get encoded by the following ruby code:

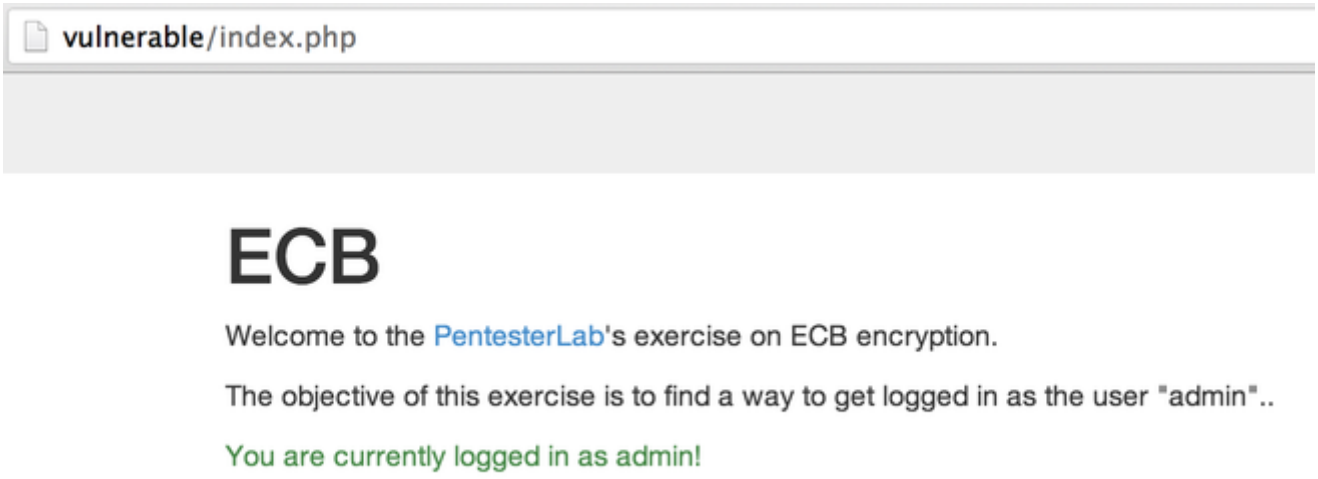
```ruby
% irb
> require 'cgi'; require 'base64'
 => true
> CGI.escape(Base64.strict_encode64("\xE0Vd.)r\xEBz\aO\xC6d\x19\xE3+\xE3"))
 => "4FZkLily63oHT8ZkGeMr4w%3D%3D"
```

language-ruby

Once you modify the cookie:



And send this value back to the application (by reloading the page), you get logged in as `admin`:



By swapping blocks around

A more complicated way to bypass this is to swap data around. We can make the assumption that the application will use an SQL query to retrieve information from the user based on his `username`. For some databases, when using the type of data `VARCHAR` (as opposed to `BINARY` for example), the following will give the same result:

```language-sql
SELECT * FROM users WHERE username='admin';

SELECT * FROM users WHERE username='admin        ';
```

The spaces after the value `admin` are ignored during the string comparison. We will use this to play with the encrypted blocks.

Our goal is to end up with the following encrypted data:

`ECB(admin [separator]password)`

We know that our separator is only composed of one byte. We can use this information to create the perfect `username` and `password` to be able to swap the blocks and get the correct forged value.

We need to find a username and a password for which:

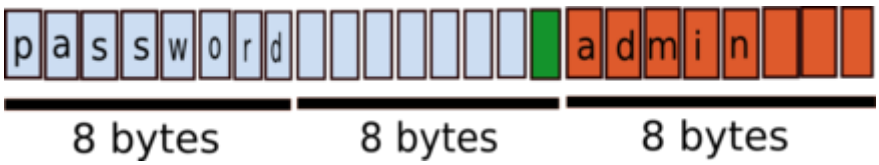
- the password starts with `admin` to be used as the new username.
- the encrypted password should be located at the start of a new block.
- the `username+delimiter` length should be divisible by the block size (from previous conditions)

By playing around, we can see that the following values work:

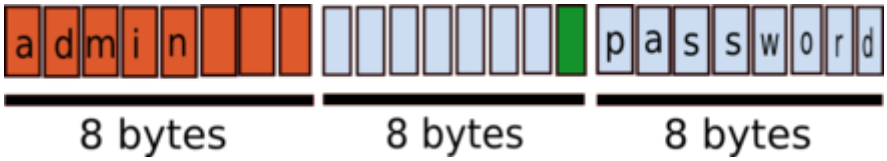
- a `username` composed of `password` (8 bytes) followed by 7 spaces (1 byte will be used by the delimiter).
- a `password` composed of `admin` followed by 3 spaces (`8 - length("admin")`).

When creating this user, use a proxy to intercept the request and make sure your browser didn't remove the space characters.

If you create correctly this user, the encrypted information will look like:



Using some Ruby (or even with Burp decoder), you can swap the first 8 bytes with the last 8 bytes to get the following encrypted stream:



Once you modify your cookie, and you reload the page, you should be logged in as `admin`: