Lacuna Software OTP challenge

João Tito do Nascimento Silva
jt.mat@hotmail.com

Abstract—this little work has the purpose of exposing and developing the ideas for solving the OTP challenge given by *Lacuna Software*. The ideas developed here are simple and give the theoretical basis and methodology for developing the code, that is the final objective of this work. Some readers may be interested in just understand the solution provided in this work, and, for those, the section Introduction may be skipped.

Index Terms—One time pad, cryptography, security, software, Lacuna Software.

1 Introduction

In January 2021, a challenge given by Lacuna Software was published as a way of finding new developers and grow their team. As the challenge specification states, it is based on the *One Time Pad* cryptography algorithm. This algorithm is unbreakable, but there are some conditions for this to be true. One of the most important conditions is that each key **must** be used only once, and should not be used again. These requirements may bring some trouble when the implementation is not good (for example, when keys are chosen by means of some easily predictable key generation function). In the case of this challenge, the key is actually repeated, and the process of generating ciphertext for tokens is simply a xor operation of some content with this key.

1.1 Understanding the challenge

In this subsection, the objective is to describe briefly the algorithm used and pave the way to introduce the ideas for breaking it later.

1.1.1 One time pad

The OTP algorithm used in this challenge consists of applying a xor operation to each character of text and the key. The server always uses the same key for encrypting content.

1.1.2 How is OTP used?

The OTP is used to generate *authentication tokens* for a User in a simple API. The token is generated by applying the OTP algorithm described above to some content. One of the information in the content of the token is the username, that occupies a max. of 32 characters in the content. When the username is smaller than this, the remaining space is filled with a fixed character.

1.1.3 The API

There is a simple API that enables us to create and login with user credentials. The API also has an endpoint that only the user ”master” can use.

POST	/api/users/create
POST	/api/users/login
GET	/api/secret

The login endpoint emits tokens using the OTP described. Each token expires in 20s after the emission.

1.2 The objective

Our objective in the challenge is to forge a valid token for the user ”master” and use it to make a GET request to the restricted endpoint.

2 Breaking the OTP

Now we proceed to analyse the One Time Pad weaknesses and try to find a way to forge a new token.

2.1 What information is needed to forge a token?

To create a token, the API uses, at least, this information: the username of the user authenticating, the padding character and the key. The user name we already have, as it is chosen by us when we create the user or, in the case of ”master”, the user name is given. The key and the padding are more challenging to find, and we are going to try to do it.

2.2 The XOR operation in the token emission

To find the key(or, at least, a part of it) and the padding char, we must exploit the XOR weaknesses associated with the repeated key. In this case, we may use some properties of this operation, which are the *commutativity* and the *associativity*, that state that $A \oplus B = B \oplus A$ and $A \oplus (B \oplus C) = (A \oplus B) \oplus C$, respectively. Also, it is important to note that $A \oplus A = 0$.

To show how those properties may be used to find some new information, let’s define a function T(p) that takes some plain text (in this case, the token plain content) and generates a new token by applying the xor operation between the key and the plain text:

$$T(p) = p \oplus k$$

Where p is the plain text and k is the key. There is a great chance that p is actually not just a function of the user, but also a function of time $p(u,t)$, as a token that is valid now will not be valid 20 seconds later. So the content of the token is time dependent, but to make the notation in the development easier, we will denote it just as p, the content already generated for some time and some user.

Now, given those definitions, what happens if we take a token and XOR it with the original plain text?

$$p \oplus T(p) = p \oplus (p \oplus k) = (p \oplus p) \oplus k = 0 \oplus k = k$$

A smaller equation:

$$p \oplus T(p) = k$$

Which means that, if we know the token content and the token, then we can xor them and find the key! In our case, we don't know the complete content, but we know that at least some part of it is composed by the username and, if the username is smaller than 32, a padding.

2.3 Finding where the username goes

We proceed to find a part of the key based on the process describe before and in the fact that we know some user name for some user that we can create. But, to find it out, we must know where the username goes in the generated token. We don't actually know *where* the username in placed in the token, but is reasonable to **assume that the username is always ciphered in the same position in the generated token**. This assumption is important and gives us the base to use the XOR operation properties.

To see how it helps us, let's imagine that we have two different tokens for two different users for which we know the usernames u_1 and u_2. Let's also call p_1 and p_2 the token plain content for the first and the second users, respectively. Using the T function mentioned before, we can apply the following development:

$$T(p_1) \oplus T(p_2) = (p_1 \oplus k) \oplus (p_2 \oplus k) = p_1 \oplus p_2$$

Now, let's call $X_u = u_1 \oplus u_2$. As u_1 and u_2 are parts of p_1 and p_2 and XOR is a bit-to-bit operation, it is reasonable to think that the position where X_u is found in $p_1 \oplus p_2$ is exactly the position where the usernames are placed when the token is generated (here is where our assumption that the usernames are always ciphered in the same position takes place). So this gives us a way to find the username position inside the generated token.

Summing up all the ideas, we must XOR two tokens for two different users and XOR their usernames. Then, we take these results and try to match the latter inside the former. This will give us a position that is exactly the position where the username must be put (including the padding character). As we don't know the padding, we must use usernames with maximum length (32 characters, according to the challenge reference).

Once we have the username position, we may proceed to discover the part of the key that is associated with this position.

2.4 Finding a part of the key

Now that we know how to find the username position, we can use this information to find a part of the key. But before proceeding, we must make an important assumption: **we will assume that besides the username, the other content of the token is user agnostic; in other words, it doesn't depend on the user that requested it and, thus, can be reused, even with a different user**. There is a great chance that the other parts of the token include the token expiration information, so we must reuse it before the expiration delta (20s).

To do it, we may first build a *mask*, that will be "xored" with a valid token to give us some information about the key. Considering that we will xor a token for some user, this mask must be a sequence of bytes that is built like this:

$$\ldots\ 0\ 0\ 0\ 0\ [username + padding\ bytes]\ 0\ 0\ 0\ 0\ \ldots$$

The development that follows will call this the *username mask* Why should the mask be like this? As our objective is to find the part of the key that is used to cipher the username, this part should be xored with the username bytes, according to the equation

$$p \oplus T(p) = k$$

shown before. But the rest of the token, because of our assumption, maybe reused to forge a new token. So if we set all the other bytes to 0, the xor result will have the key bytes in the position of the username and the rest will be *copied* from the token, to be reused later. The result of this operation will be referred as *token mask* in the development that follows, to simplify the notation. Using T_{mask} as the token mask, the equation is:

$$T_{mask} = T(p) \oplus (\ldots\ 0\ 0\ 0\ [username{+}padding\ bytes]\ 0\ 0\ 0\ \ldots)$$

It is important to notice that we still don't have the padding character. So, we must use maximum length usernames again (32 characters).

Once we have figured out the token mask, we can proceed to find the padding character.

2.5 Finding the padding char

With the token mask in hands, we may find the padding char. For this, we may create and login a user with a username smaller than the maximum. Than, we take the new token and xor it with the token mask. This works because of the XOR properties, again:

$$T(p) \oplus k = (p \oplus k) \oplus k = p$$

The equation shows us that, as the token mask contains the key, this operation will reveal the a part of the original content that corresponds to where the username is ciphered, including the padding char.

Once it is done, we can finally forge a token.

2.6 Forging a new token

At this point, we have methods to find all the information we need. Now, to forge a token, we will use a simple process with this information. This process consists of generating a new username mask and XOR it with a token mask. As described in the process of finding a part of the key, this will cipher the username and reuse the other parts of the token that was used to generate the token mask before:

$$T(p) = U_{mask} \oplus T_{mask}$$

where U_{mask} is the username mask for the user for who we wish to forge a token, and T_{mask} is a *valid* token mask. We must remember that the token mask has an expiration time equals to the expiration time of the token used to build it.

3 THOUGHTS ABOUT THE CHALLENGE

This section is directed to suggest ways to improve the algorithm and avoid token tampering.

3.1 Improving the algorithm

To avoid the problems of the OTP, the first step would be to meet all of its requirements. So, as we have seen, the critical requirement that was broken in the vulnerable application was that the key must never be reused. This requirement would make the OTP unbreakable, but also brings some trouble to the implementer, because, in this case, it would be necessary some kind of key management that could remember, for each token, what was the key used. This key management could be some kind of algorithm that could calculate the key or could be just a service that recovered this information from some database or storage. But, for both these options, there are problems.

In the first case, in which the key is calculated, the function that calculates the key should be very unpredictable, and this already brings some risk for the implementation. But also, the implementer would need to find a way to know how to calculate the key given only the token. One possible way could be to set an index somewhere in the token to be used in the key generation function, but this is usually very expensive(for a really secure generation function), and for many requests it could become a problem to the performance of the system. The biggest risk in this case is actually the possibility of a related-key attack, by statistical and mathematical analysis of many tokens.

In the second case, in which the key is retrieved, the latency to a database or the time to read the key from a storage could become expensive and affect, again, the performance of the system. Also, in this case, it could be even more secure and easier to just generate a random string as the token and store in the database the information associated with this string, without using OTP or any other cryptography. But this cpuld be really expensive in most of the cases, and becomes worse when the system receives a big volume of requests.

So we may even think of ways to use OTP meeting the requirement of not repeating a key, but this is risky and also expensive many times. A better way to generate a token maybe to use another algorithms.

3.2 Changing the algorithm

To get a really nice algorithm to generate tokens, we can think that:

1) It is really useful to have information stored inside the token rather than reading it from somewhere else, as this read operation may cause performance problems.
2) We must verify that the token was actually forged by us rather than forged by someone else. (authenticity)
3) We must verify that the token was not modified. (integrity)
4) The tokens must not be too big, as we want to use it in HTTP requests.

The second and the third items that we wish to have can be achieved by using a digital signature. In order to get the other items, it is very common to use a special type of token, called *Json Web Tokens* (JWT). This is a form of token that is massively used today and is already implemented in many programming languages. Also, the popularity and the number of researches it has give to this algorithm a great reliability. Thus, this is a great candidate for replacing the OTP based token emission.

3.2.1 What is a JWT?

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA. [1]

JWT can also be encrypted to get secrecy between the perties communicating.

A token emited using JWT has a structure based in *claims*. The claims are the content of a JSON, and provide information about the parties. For example, JWT can be used in *RBAC* by providing a user's role in a claim.

3.2.2 JWT format

A token in JWT has a common format:

$$header.payload.signature$$

Where:

1) Header: JSON that contains information about the algorithm, such as the algorithm used and the type of the token. Example:

```
{
    "alg": "HS256",
    "typ": "JWT"
}
```

2) Payload: JSON that contains the claims. The claims maybe *registered claims*, *public claims* or *private claims*. Example:

```
{
    "sub": "123456789",
    "email": "jt.mat@hotmail.com",
    "role": "admin"
}
```

3) Signature: a digital signature generated by using some strong algorithm over a string composed by

$$base64(header) + "." + base64(payload)$$

The algorithm maybe HMAC, for example. It is also common to use asymmetric key algorithms, such as RSA.

All these fields are Base64 encoded before being used to form the token.

3.3 Notes about JWT

Even thought JWT is a good algorithm, we must be careful. First of all, the claims should not be exposed if they contain sensible content. In this case, the token must be encrypted. Another item is that we must always check if the implementation of JWT that we are using checks the algorithm passed in the header part of the token or just uses what is received, because an attacker could abuse it and force the application to accept a plain token, for example, by choosing the algorithm as "none" or even force the application to drop a public key [3].

Another important note about JWT is that it is also important to implement some authorization scheme in most of the cases, as JWT just checks if the sender is authenticated and doesn't guarantee that the sender is authorized to do the action requested.

4 Some notes about the implementation

The implementation is a C# code that runs an API to demonstrate the steps to break the challenge. To understand the implementation, the reader may look at the implementation of the token analysis service (Lacuna.Api/Services/TokenAnalyserService.cs). It is also useful to take a look in the Binary.cs file in Lacuna.Shared/Entities.

In order to keep a good architecture and a clean code, the code follows some guidelines of Domain Driven Design [2] and Clean Architecture, by setting the Domain code apart from the Controllers' code and any code that may depend on many other packages.

5 Conclusion

This challenge shows us that, although OTP, in theory, is unbreakable, its implementation brings some challenges and risks. In our real life implementations it is always important to keep a routine of study to make sure that we understand what we do, and also to understand how it can be improved.

References

[1] https://jwt.io/introduction
[2] https://martinfowler.com/bliki/DomainDrivenDesign.html
[3] https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/