VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS
INSTITUTE OF COMPUTER SCIENCE
INFORMATION TECHNOLOGIES STUDY PROGRAM

Network Security

PYTHON-JWT UP TO 3.3.3 AUTHENTICATION SPOOFING

Done by:
Austėja Bauraitė
Deividas Bendaravičius
Julius Valma
Emilis Šerys

Vilnius
2023

Contents

1 Python-jwt library and scenario overview

The Python-JWT is a Python module that facilitates the generation and verification of JSON Web Tokens. However, there's an issue with all versions preceding 3.3.4, which are vulnerable to Authentication Bypass by Spoofing. This vulnerability may have severe consequences such as identity spoofing, session hijacking, or even bypassing authentication altogether. Should an attacker get hold of a JWT, they can manipulate its contents at will, even without having knowledge of the secret key. This could potentially allow the attacker to identify as other users, take over their sessions, or evade authentication.

The root of this problem lies in the inconsistent ways that JWT parsers, used by Python-JWT and its dependent module jwcrypto, interpret data. If an attacker alternates between compact and JSON formats, they can trick jwcrypto into parsing different claims than those the signature was initially intended to validate.

Digging deeper, we can pinpoint the vulnerability to the __init__.py file in the python_jwt directory. Before rectifying the issue, this file failed to validate JWT format appropriately and would accept an intermixed array of compact and JSON payloads from an attacker. This loophole thus provided the attacker an avenue to mislead jwcrypto into interpreting different claims than those the signature was initially validating.

To demonstrate how this vulnerability works, we will stage a scenario with a "defending server" and an "attacking client":

- Defender scenario - the "defender" server will run a python-written service that uses JWT tokens to securely transfer data between parties. In particular, version 3.3.3 in which the vulnerability is present will be installed.
- Attacker scenario - the "attacker" will be sending a mix of compact and JSON representations as JWT tokens to bypass authentication/hijack the session.

2 Python-jwt upgrade on vulnerability prevention

Python code snippt is designed to add a basic level of validation to the JWT (JSON Web Token) input. The vulnerability in python-jwt 3.3.3 allowed an attacker to forge the JWT content arbitrarily without needing to know the secret key, potentially leading to unauthorized access.

```
102   _jwt_re = re.compile(r'^[A-Za-z0-9\-_]+\.[A-Za-z0-9\-_]+\.[A-Za-z0-9\-_]*$')
103   def _check_jwt_format(jwt):
104       if not _jwt_re.match(jwt):
105           raise _JWTError('invalid JWT format')
106
```

The code:

- It defines a regular expression (regex) to match the format of a properly structured JWT. A regular expression is a sequence of characters that forms a search pattern. It can be used to check if a string matches a specific pattern.
- JWTs consist of three parts separated by dots (.):
  - Header: Base64Url encoded
  - Payload: Base64Url encoded

  - Signature

  Each part consists of URL-safe characters, i.e., Alphanumeric characters, hyphen (-), and underscore (_). The regex is designed to match this format.

- The _check_jwt_format(jwt) function uses this regex to check if the input JWT matches the expected format. If it does not, an error is raised.

This code snippet acts as a first line of defense against malicious JWTs. By ensuring that the JWT at least has a basic structure, it prevents certain types of attacks where the attacker sends improperly formatted JWTs to exploit vulnerabilities in the parsing or processing code.

3 Vulnerability assessment

The problem in this Python JWT library, specifically in versions up to 3.3.3. "alg" field in the JWT token's header is that it's not being handled right. To give you some context, the "alg" parameter is what points out the cryptographic algorithm that's used to sign the token.

The vulnerability starts to appear when the library doesn't check the "alg" parameter as it should during the token verification process. This oversight can be exploited by crafty attackers, who can create a JWT token with a correct signature but a misleading "alg" value. They could set the "alg" value to "none" or "HS256" instead of the expected "RS256", confusing the server into accepting an illegitimate token as valid.

What this means is that an attacker can pose as a valid user and get their hands on restricted resources. If you're using this library, you should upgrade to version 3.3.4 or later. These newer versions have a patch that sorts out this issue by improving the way the token is checked as in the section mentioned above.

4 Algorithm check

This vulnerability issue arises from not strictly enforcing the allowed algorithms during the JWT verification process. In the verify_jwt function.

If the 'none' algorithm is in the allowed_algs list and no public key (pub_key) is provided, it will not raise an error related to the signature.

The use of the "none" algorithm in the JWT generation (generate_jwt) and verification (verify_jwt) process. The "none" algorithm allows creating a JWT without a signature. Setting the "alg" header to "none" if no private key is provided, means an unsigned token will be generated.

```
59      header = {
60          'typ': 'JWT',
61          'alg': algorithm if priv_key else 'none'
62      }
```

In the verify_jwt function, the code checks if "none" is in the allowed_algs. If "none" is allowed as an algorithm and no public key is provided for verification, the JWT signature is not verified, which could enable an attacker to forge JWTs without knowing the secret key.

However, if the 'none' algorithm is not in the allowed_algs list and no public key is provided, it will raise a _JWTError with the message "no key but none alg not allowed", effectively preventing the misuse of the 'none' algorithm.

This indicates that the library has implemented proper handling of the 'none' algorithm, and the vulnerability should be mitigated in python_jwt v3.3.3.

```
165
166     if pub_key:
167         token = JWS()
168         token.allowed_algs = allowed_algs
169         token.deserialize(jwt, pub_key)
170     elif 'none' not in allowed_algs:
171         raise _JWTError('no key but none alg not allowed')
172
```

5 Executing tests

The test designed to examine a potential security issue in the python_jwt library, version 3.3.3. It looks into whether an attacker can create a fake token by blending compact and JSON formats, and then sneak in false claims—for instance, changing the user or tweaking the expiration time. Our goal is to ensure that the library can spot and reject such faked tokens.

The process:

- A valid JWT is generated with a payload containing a subject (sub) claim for the user "alice". This is done using the jwt.generate_jwt function, and a valid key and algorithm. The JWT is set to expire in 60 minutes.
- The generated JWT is split into its three components: header, payload, and signature.
- The payload is base64url decoded and then JSON parsed into a dictionary object.
- The subject claim is then changed from "alice" to "Antanas", and the expiration time (exp) is set to a later date. This effectively forges a new set of claims.
- The modified claims are then JSON serialized, base64url encoded, and then inserted back into the original JWT, replacing the original payload.
- A fake token is constructed, however, it does not adhere to the required structure. This is accomplished by appending additional characters in front of the header section (equivalent to prepending `{"`). Subsequently, the modified payload, which is the core data segment, is integrated into the token. Finally, we leave the genuine signature (which is typically used to authenticate the token), instead leaving it blank (″). Counterfeit token has been fabricated that doesn't comply with the established format of a legitimate token.
- The jwt.verify_jwt function is then used to verify the forged token. Because the format of the JWT is incorrect, the newly added code in version 3.3.4 should raise an error with the message 'invalid JWT format'.
- The test then asserts that the exception raised has the expected error message, confirming that the vulnerability has been patched.

Considering the test is correct, the verification process should throw an error, and the test has done its job of catching and rejecting the fake token, meaning we can consider the vulnerability patched. Otherwise, the threat is still present, and more validation checks should be enforced to counter this kind of attack.

6 Difficulties and Benefits

Trying to replicate the vulnerability that resides in the Python-JWT 3.3.3, we have encountered some issues. First and foremost, we had a hard time understanding the root problem of where the issue originated from. However, after we thoroughly went through the Github repository that holds the source code of the library, we found the exact changes that solved the issue, how it was fixed and what was the vulnerability of the library. Secondly, before dealing with the issue, we knew about how JWT tokens were used in projects and their benefits, however, we lacked knowledge of how exactly they were generated and verified, in what format they were generated, and the parts that made up the token itself. This was a crucial breakthrough because it let us understand how, without proper validation, a user could parse a modified or spoofed version of the token, which resulted in the exact vulnerability that we recreated during this project. **Overall**, while we did not encounter many obstacles recreating this vulnerability, we did learn a lot about one of the still most used authentication methods - JWT tokens, and how exploiting poorly written and widely used code without written tests, could result in issues that could have significant and serious financial costs, data breaches and other damages. Additionally, we better understand the importance of tests in our code, and why code coverage is one of the most important steps in a project, that is open-source.