Fortinet Assessment Report

Quentin Dubois

2023-11-04

Subject

Let's say you manage the pet store website and you want to ensure that the application programming interface (API) endpoints is secure.

You will focus on the following URLs :

- POST /v2/store/order
- GET /v2/store/order/{orderId}
- DELETE /v2/store/order/{orderId}

Questions

Question 1

Question : Provide an efficient and easy-to-use python code that queries these URLs and parameters

You can find the Python code of petstore client API in the following GitHub repository : QuentinDubois-Polytech/Fortinet-Assessment (github.com).

Question 2

Question : List common vulnerabilities that these API endpoints may contain.

To respond to this question, I retrieved my favorite site, which lists the most common vulnerabilities related to web applications, OWASP (Open Web Application Security Project) and in particular OWASP Top 10. I chose some of the vulnerabilities' types mentioned in this article that are applicable to our case study, and I developed a little bit on each one.

Broken Access Control

Broken Access Control means that users can perform actions that their permissions do not authorize. For example, accessing API endpoints for which the user does not have the necessary permissions, leading to the access or modification of protected and unauthorized data.

Identification and Authentication Failures

This part corresponds to the user's identity, authentication and session management. The most common vulnerabilities are exposed session identifiers in the URL, reusing session identifiers after successful login, and not invalidating session identifiers after the user's logout.

Code Injection

Code Injection means the user can inject data containing code that will be executed, changing the normal program's flow. It comes from using invalid and unchecked user input. As examples of code injection, there are : command injection, SQL Injection, NoSQL injection, Cross-site Scripting (XSS)...

Security Misconfiguration

The application's configurations are not properly defined. In the case of an API, the most common vulnerability is when error handling reveals stack traces or other overly informative error messages to users. This information can be collected during the recognition phase of an attacker and used to understand the behavior of the program or recover information on the versions of the software used.

Vulnerable and Outdated components

Some outdated components may contain known vulnerabilities (CVE), that can be used by an attacker to do unauthorized actions. The search for CVE on server components is easier if, as we saw above in the "Security Misconfigurations" section, the server leaks the software's versions in error messages. CVE is a general term that contains all sorts of vulnerabilities that can be exploited in software.

To give you an example of CVE type that can be used in our study case, it is "Parser DOS" which can also be called "Algorithmic Complexity Attack". The parsing of some specific data can take longer than others, even if they have the same size. This behavior comes from the implementation of the parser and, most of the time, a bug in it. This vulnerability can then be used by attackers to reduce the availability of a service by emitting requests containing these "specific data". By searching for an example on the Internet, I found an example of this CVE type in a library that parses regular expressions. You can check for yourself, if you are interested, at the following link : Regular Expression Denial of Service (ReDoS) in ua-parser-js | CVE-2022-25927 | Snyk.

I will stop here. I listed the most important vulnerabilities that could be applicable to our study case. Furthermore, I didn't speak of some other common vulnerabilities, like cryptographic failure, because there is no encryption on arguments or data between the client and the server through the API.

Question 3

Question : Propose tests to detect if these vulnerabilities exists on these endpoints.

Code injection

For all the code injections, our goal will be to add code as user input to the API and then see if the code is executed. If that is the case, the application is vulnerable to code injections.

SQL Injection

First, we need to find an API call that takes string parameters. Then we need to be sure these parameters do not need to be formatted in a certain way, for example, in the *ship_date* argument of the petstore API, the format expected is ISO 8601. When the right parameters are selected, we need to inject the following input : `"#"` or `"--"`, the goal is to inject a character interpreted as a comment to create a malformed SQL query, generated by the server. When, the SQL query is sent by the server to the SQL database, it should receive an error, which should be passed to the client (us). Most of the time, this type of error is not caught, and it will cause the server to send a 500, Internal Error, to the client. We can even check our assumption, if the stack trace is enabled in error messages. We will go deeper into this point in the section “Security Misconfiguration”.

To resume, by injecting our input, if the server returned an error to the client and the input should be accepted, we found an SQL injection vulnerability. Then, we can verify by trying to exploit the vulnerabilities, if we have a SELECT statement, by recovering unauthorized information.

There are some automated tools that can verify and exploit SQL injection on a web server. They are quite powerful and can even retrieve the different tables of the database and the data in them. They use a combination of different techniques : Error-based, Union-based... The tool I use the most is `sqlmap`.

Command injection

Here, we will try to add Linux code (bash) or Windows code (powershell, cmd). Normally, we already know which system we are targeting thanks to the recognition phase. A simple way to know the OS used by the machine hosting the API service is to use `nmap` with the `-sC` and `-sV` options. If an injected command is executed, we should retrieve the result of the command in the server response. The same principle goes for other injection techniques like Cross-Site Scripting (XSS), NoSQL injections...

Security Misconfiguration

Overly informative error messages

To detect if the application returns a stack trace in error messages to the user, we will have to provoke an error on the server side and analyze the error message in the response. An easy way to do that is to provoke a typing error by using a wrong type for a particular parameter, for example, a string instead of an int.

Vulnerable and Outdated components

This category is easier if we have a lot of information. If, during the recognition phase, we gather information about software and versions on the server side, our task will be really simple, we will just have to check if there is CVEs for these software programs in general vulnerability databases like exploit-db) or NVD (National Vulnerability Database) - NIST). We can also leverage the vulnerability

database of the software company, most of them are public, even if they are not necessarily up-to-date. Of course, the lookup in the vulnerabilities databases can be automated.

If we have less information, for example, only the version of the software, or even worse, no information at all on the software used. We will have to play “blind”, by trying to exploit vulnerabilities with automated tools and see if we get lucky.

Identification and Authentication Failures

The goal here is to see if we can impersonate another user just by having identification information on him (id, username, email…) or a session identifier, stolen somehow. We may also want to know if there are some mitigations to automated attacks by dictionaries or brute force on a user. This information could be useful if we want to break a user’s password.

Broken Access Control

Here, the goal is to try to access resources we are not supposed to, because we don’t have enough permissions. The first phase is to verify if the protected API routes require authentication. The second is to try the same routes with a user who does not have enough permissions. If, during the first or second phase, we find a match, then we have a vulnerability related to access control.