DD2525 LAB2

Group 16

Regular Expression Denial of Service (ReDoS)

We found two vulnerabilities.

Exploit

(1) The first vulnerable application logic subject to ReDoS attacks is validatePassword function in auth.js. It uses match () function however in JavaScript, the match () function is used to determine whether a string matches a certain pattern and it accepts a regular expression as a parameter and attempts to find parts of the string that match that regular expression. The actual attacker can enter “{a+}+b” as the username and “aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ab” as the password. When the regular expression engine processes this input, it encounters the pattern "{a+} +b" in the username, which contains nested repetitions. the engine must check each possible combination of the nested repetitions to find a match. Since the password "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ab" includes a long sequence of 'a's followed by a space and then 'ab', the engine will have to attempt numerous combinations of nested repetitions, gradually backing off to shorter matches. This process can lead to a significant number of steps being performed by the regular expression engine, potentially causing a ReDoS attack due to the exponential complexity.

(2) The second vulnerable application logic subject to ReDoS attacks is the search function of forum.js. The actual attacker can enter excessively long repetitions of characters in the search box such as a {1000000} b. Regular expressions work by matching patterns in text, and they often use backtracking algorithms to find all possible matches. When a regular expression contains a long repetition, such as {1000000}, the engine must attempt to match that exact number of repetitions, which can lead to significant backtracking. In the case of "a {1000000} b", the engine will try to match one billion "a"s followed by a single "b". This can result in exponential time complexity because the engine may need to backtrack extensively to explore all possible combinations of "a"s before determining that there is no match. Malicious actors can exploit this behavior by crafting input strings that trigger excessive backtracking, causing the regular expression engine to consume large amounts of CPU time and leading to a denial of service (DoS) condition.

Mitigation

(1) Changing the match() method to include() method works because the includes() method does not support regular expressions, which means that it cannot be used to execute a ReDoS attack by passing in a malicious regular expression pattern.

(2)The improvement is the addition of the regText function. The regText function first truncates the input text to a maximum of 100 characters using text.substring(0, 100). This step ensures that the input size passed to the regular expression engine is bounded, reducing the potential for excessive backtracking.

Next, the function applies the regular expression replacement operation on the truncated text using text.replace(/[-[\]{}()*+?.,\\^$|#\s]/g, "\\$&"). This regex pattern escapes special characters commonly used in regular expressions. By processing only the truncated input text, the regex engine has a limited scope to perform its matching operation, reducing the risk of ReDoS attacks.

Discussion

Existing web defenses such as rate limiting, input validation, and efficient algorithms can mitigate ReDoS (Regular Expression Denial of Service) attacks to some extent. Rate limiting can restrict the number of requests from a single IP address, preventing an attacker from overwhelming the server with malicious requests. Input validation can be employed to ensure that user input meets expected patterns, thereby reducing the likelihood of triggering catastrophic backtracking in regular expressions. Additionally, using efficient algorithms for tasks like parsing and string manipulation can minimize the impact of ReDoS attacks by reducing the time complexity of regular expression evaluations.

Web Side Channels

Exploit

The system promotion in forumerly can be used as a side channel. First, the attacker can obtain all registered license plate numbers by continuously registering on the registration page. The attacker can use a username that is not registered and enter two different passwords. Each time, only entering a different license plate number will reveal whether that license plate is registered or not. If the license plate is registered, the system will prompt "License plate already registered by other user". If not, the system will prompt "Passwords do not match". Indeed, the availability of usernames can also be determined using a similar method. After obtaining all registered license plate numbers, the attacker can log in to his profile and park the car to get the car parking information. For instance, the attacker obtains a license plate number "ABC123" and then parks "ABC123" as his car in the first position of the eighteen positions provided on the webpage. If the system doesn't prompt anything, the attacker can park "ABC123" in the next position until the system prompts, "This car is already parked in the selected location. Do not waste your money!", indicating that the car is parked there. If the attacker tries 18 times and the system still prompts nothing, it means the car is not parked. In this way, the attacker can obtain parking information for all registered license plate numbers.

We provide two versions of implementations to exploit each of the found side channels. The first one is "Simulate Real Attacks," which means it will attempt all combinations of six characters for A, B, C, D, and numbers, and it will store the obtained registered license plate

information in a list. Then, for each element in this list, it will iterate through parking locations until the system prompts "This car is already parked in the selected location. Do not waste your money!" or until all locations have been tried. The results along with the corresponding license plate numbers will be stored.
The other version is a simpler version which is for presentation. The total number of combinations for six characters from A, B, C, D, and numbers is indeed 15^6 and the process will take a significant amount of time to complete. Therefore, for demonstration purposes, let's assume that "ABC123" is registered and parked. Then, we can proceed with the aforementioned operation using this registered license plate.

Mitigation

Forumerly should enforce a strict policy where only the license plate associated with the registered account can be parked, and during registration, validate whether the license plate belongs to the registrant. This measure adds a layer of security, ensuring that only legitimate users can park their vehicles under their accounts, thus reducing the risk of unauthorized access or information leakage. To mitigate the side-channel attacks, Forumerly can also implement consistent error messages regardless of license plate validity, introduce randomized error messages, enforce rate limiting on registration attempts, and protect against user enumeration with measures like CAPTCHA challenges or registration attempt delays.

Discussion

Existing web defenses can aid in protecting against Web Side Channels attacks by implementing measures such as content security policies (CSP), cross-origin resource sharing (CORS) restrictions, and secure cookie settings. CSP can mitigate certain side-channel attacks by controlling which external resources can be loaded by a web page, limiting the potential leakage of sensitive information. CORS restrictions help prevent unauthorized access to resources across different origins, reducing the risk of side-channel attacks exploiting cross-origin interactions. Additionally, employing secure cookie settings such as HTTPOnly and Secure flags can mitigate the risk of side-channel attacks targeting session tokens or sensitive information stored in cookies.

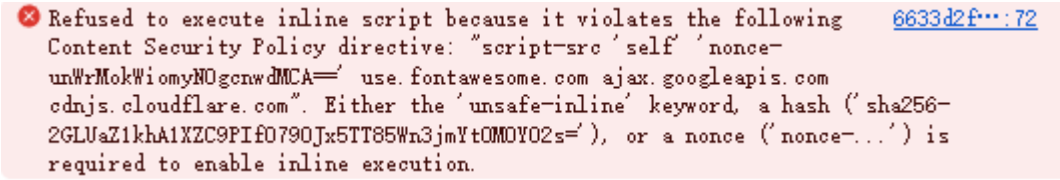
Cross Site Scripting (XSS)

Exploit

The exploit for the XSS vulnerability was carried out by uploading a malicious JavaScript file disguised as an image (.png.js) to the "images" of the “profile” page. This type of file, due to its misleading extension, bypassed the server's basic file validation checks. The actual exploit was triggered by inserting a malformed <script> tag within a post: <s<scriptcript src="/images/profileImages/test1"></script>. The use of <s before the <script> tag was a deliberate attempt to evade simple text replacement security measures in the forum's processing logic, which tried to strip out <script tags from user input. When a post containing the malicious script is viewed, it triggers an alert("XSS attack successful!"); popup. This alert serves as proof that the XSS attack

was successful, demonstrating that arbitrary JavaScript code has been executed in the viewer's browser. The real danger of this vulnerability lies in its potential for more harmful exploits beyond simple alert messages.

The Content Security Policy (CSP) implemented was designed to restrict the sources from which scripts could be loaded, presumably limiting them to scripts originating from the same domain ('self'). When I wrote '<s<scriptcript>alert('XSS');></script>' in the thread section, the script is rejected by CSP because it's not from 'self'.



However, because the malicious script was hosted on the same domain in the user-uploaded "profileImages" directory, the CSP did not effectively block the script execution. This vulnerability could potentially allow an attacker to execute scripts in the context of the user's session, leading to actions performed on behalf of the user, data theft, or session hijacking.

Mitigation

To mitigate this vulnerability, the escape function was introduced in the forum.js to process any user input before it is stored or displayed. The escape function converts special characters in the input into their HTML entity equivalents, thereby preventing any part of the input from being interpreted as executable code. This approach helps to safeguard against both stored and reflected XSS attacks by ensuring that all user-provided content is treated as plain text when rendered in a browser.

Remote Code Execution

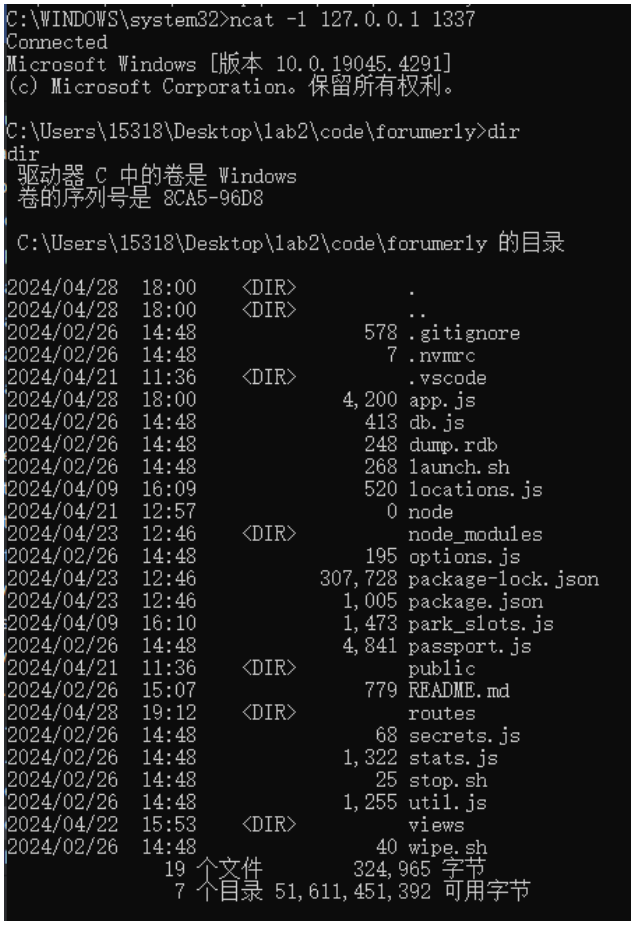
In the user.js of merge function, it exhibits a prototype vulnerability. While the 'isObject' function, used to determine if the target and source keys are the same, fails to differentiate between legitimate objects and prototype pollution, thereby merging them indiscriminately.

To exploit this vulnerability, I attacked in the upload/user section, which interestingly does not require authentication with no login. Using Python, I first uploaded a normal user file named 'ben.json' and another file containing potentially malicious data, 'ben_attack.json', using Python's 'requests' library to send data to the server. In the file 'ben_attack.json', I wrote a "proto" key to implement remote code execution via

prototype pollution.

This malicious file attempts to pollute the prototype of all objects by inserting a proto object in the JSON data. Within the ‘passport.js’ login source code, whenever ‘userAutoCreateTemplate’ is set to true, it triggers the execution of ‘const newUser = JSON.parse(eval(wrapperFunction))’. However, this use of eval() is risky as it allows the execution of a reverse shell script designed to establish a network connection. The reverse shell method, provided by the lab handbook, is intended to create a remote session.

Following this setup, if the app.js interface indicates successful registration, executing ncat -l 127.0.0.1 1337 on cmd can connects to the Forumly interface, indicating a successful remote connection.



Furthermore, when ben_attack.json contains "proto":{"toString": 1}, it causes processes that invoke this method to terminate due to incorrect return values, effectively causing a denial of service on the upload/user process. The server shows the error as blown.

```
TypeError: Object.prototype.toString.call is not a function
    at isMap (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\bson\lib\parser\utils.js:77:38)
    at serializeInto (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\bson\lib\parser\serializer.js:649:63)
    at Object.serialize (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\bson\lib\bson.js:122:61)
    at Msg.serializeBson (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\cmap\commands.js:531:21)
    at Msg.makeDocumentSegment (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\cmap\commands.js:525
:37)
    at Msg.toBin (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\cmap\commands.js:514:29)
    at MessageStream.writeCommand (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\cmap\message_stre
am.js:41:34)
    at write (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\cmap\connection.js:570:30)
    at Connection.command (C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\cmap\connection.js:299:13
)
    at C:\Users\15318\Desktop\lab2\code\forumerly\node_modules\mongodb\lib\sdam\server.js:198:18
```

Contributions

5.1&5.2 are completed by Tianyi Wang 5.3&5.4are completed by Keyu Li