



SC4013 Application Security

Web Application Firewall (WAF)

Rule Development

Name: Ang Kai Jun

Matriculation No.: U2122649H

Contents Page

Introduction.....	3
Environment Setup.....	3
EC2 instance	3
Application Load Balancer (ALB).....	4
Web Application Firewall	4
Vulnerability Analysis and WAF rules	5
Introduction to server code and API endpoints:	5
Vulnerabilities and WAF rules.....	5
1. Geo-Blocking:	5
2. IP Whitelist:	6
3. Local File Inclusion (LFI):	7
4. Remote File Inclusion	8
5. SQL Injection	10
Rule Priority	11
Testing.....	11
Correctness Test.....	11
Performance Impact Analysis	12
Rule Bypass and Mitigation:	13
Invalid JSON attack.....	13
Mitigation	14
WAF Strengths and Limitations:	14
Conclusion and Recommendations	15
References	16
Annex	17

Introduction

The project explores the Web Application Firewall (WAF) Rule Development. In this project, I created a vulnerable backend server system, that is susceptible to the various types of vulnerabilities that we learnt in class. Eg SQL Injection, Local File Inclusion, Remote File Inclusion, XSS, etc. This is followed by the development of WAF rules to prevent these vulnerabilities from being exploited, and then testing the correctness and performance impact of the WAF rules.

Environment Setup

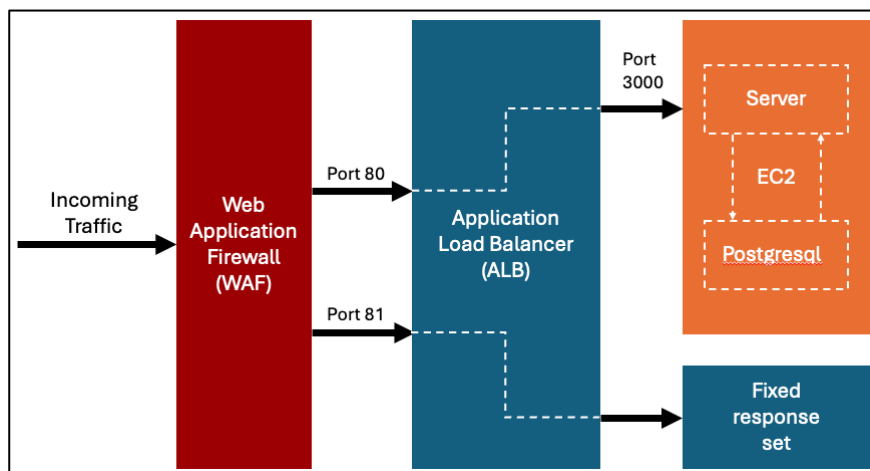


Fig 1: Diagram showing the setup architecture

The environment for the project is set up on Amazon Web Services (AWS). As seen from Fig 1, there are 3 parts to the setup: WAF, ALB, and the EC2 instance.

EC2 instance

Firstly, the EC2 instance created is used to host the backend server and PostgreSQL. The server is written using the Express.js framework and can be found on Github (<https://github.com/kaijun123/SC4013-WAF>). The security policy of the EC2 instance is crafted to expose port 3000 to custom TCP connections. Port 3000 is the port that the server will be listening to for incoming API calls, hence it is important to expose the port.

Application Load Balancer (ALB)

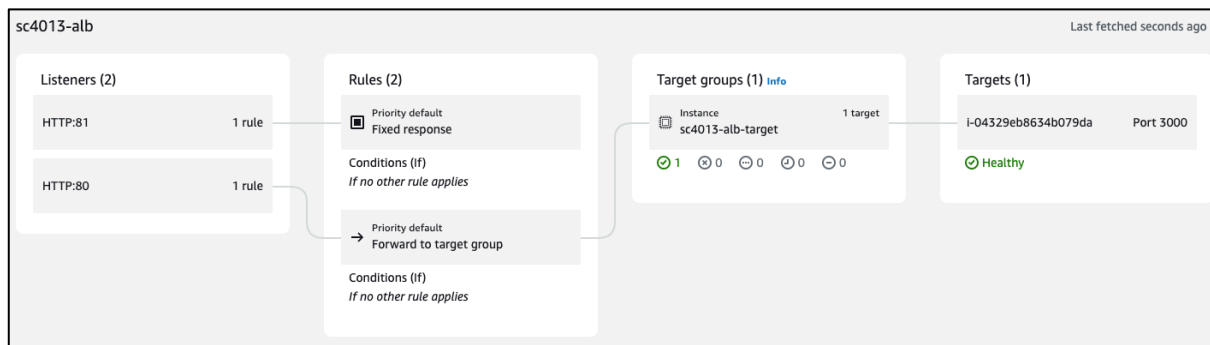


Fig 2: Diagram showing the routing rules of the ALB

Secondly, the ALB is created as AWS does not allow for the WAF to be associated directly to the EC2 instance. The alternative to the ALB would be an API Gateway. The API Gateway is given the following routing rules:

1. For requests received on Port 80: Route them to Port 3000 of the EC2 instance
2. For requests received on Port 81: Return a fixed response to the client (for debugging purposes, to determine if the load balancer is running)

Web Application Firewall

Lastly, WAF stands before the ALB, and is the service that will filter, monitor and block incoming traffic based on the rules created.

Vulnerability Analysis and WAF rules

Introduction to server code and API endpoints:

- **Endpoints**
 - **/admin:** admin functionalities
 - **/delete:** delete a specific book based on a given id
 - **/deleteAll:** delete all the books in the database
 - **/insert:** insert a new book
 - **/user:** user functionalities
 - **/list:** list all the books in the database
 - **/download:** download the electronic version of a specific book
 - **/upload:** upload the electronic version of a specific book
 - **/lfi:** test for Local File Inclusion
 - **/rfi:** test for Remote File Inclusion

Fig 3: List of endpoints available on the server

The API endpoints were created to mimic the functionalities of an online book repository, whereby users can upload and download electronic versions of books freely. No Authentication system was created for ease of testing.

The endpoints written are shown in Fig 3.

Vulnerabilities and WAF rules

1. Geo-Blocking:

NOT Statement 1	Action
Request option	The action to take when a web request matches the rule statement.
Originates from countries	Action
Country	Block
Singapore - SG	Custom response code
IP address in header	401
X-Forwarded-For	Add labels
Fallback for missing IP address	-
No match	

Fig 4: Geo-Blocking rule

To restrict application requests only from places where the service needs to be provided and reduce the attack surface, a Geo-Blocking WAF rule was implemented as seen in Fig 4. The rule restricts access only to IP addresses from Singapore.

Statement

Inspect

Originates from a country in

Country codes

Choose country codes

Singapore - SG

X

IP address to use to determine the country of origin

When a request comes through a CDN or other proxy network, the source IP address identifies the proxy and the original IP address is sent in a header. Use caution with the option, IP address in header, because headers can be handled inconsistently by proxies and they can be modified to bypass inspection.

Source IP address

IP address in header

Fig 5: Geo-Blocking rule

To be able to test the correctness of the rule, the rule targets the IP addresses in the *X-Forwarded-For* header, rather than the source IP address. However, in a production environment, the rule should target the source IP address instead. Fig 5 shows the configurations available by AWS WAF when choosing how to determine the IP address.

2. IP Whitelist:

```
const router = Router()

router.delete("/delete", deleteHandler)
router.delete("/deleteAll", deleteAllHandler)
router.post("/insert", insertHandler)

export default router
```

Fig 6: List of Admin endpoints

Fig 6 shows the endpoints that is only for admin access. Since the app does not contain any authentication and authorization mechanisms, a rule was created to only allow access to the admin endpoints by a selected list of whitelisted IP addresses.

NOT Statement 1	Statement 2	Action
<div>IP set name</div> <div>sc4013-ip-whitelist</div> <div>IP set description</div> <div>List of ips that should be able to reach the loadbalancer and ec2 instance</div> <div>IP set ARN</div> <div></div> <div>IP address version</div> <div>IPV4</div> <div>IP address in header</div> <div>X-Forwarded-For (Any IP address)</div> <div>Fallback for missing IP address</div> <div>No match</div> <div>IP addresses</div>	<div>Field to match</div> <div>URI path</div> <div>Positional constraint</div> <div>Contains string</div> <div>Search string</div> <div>/admin</div> <div>Text transformations</div> <div> <div>None (Priority 0)</div> </div>	<div>The action to take when a web request matches the rule statement.</div> <div>Action</div> <div>Block</div> <div>Custom response code</div> <div>402</div> <div>Add labels</div> <div>-</div>

Fig 7: IP Whitelist rule

Fig 7 shows the rule for IP whitelisting of the admin endpoints. It checks for the */admin* string in the URI as well as whether the IP address in the *X-Forwarded-For* header exists within the

IP set. An IP set is a group of IP addresses that can be customized. In this case, the IP set contains IP addresses that are allowed to access the admin endpoints (ie IP addresses of admins).

Similar to the Geo-Blocking rule, the rule checks the X-Forwarded-For header for testing purposes. In a production setting, the source IP address should be used instead.

3. Local File Inclusion (LFI):

```
import { NextFunction, Request, Response } from "express"

export const localFileInclusionHandler = async (req: Request, res: Response, next: NextFunction) => {

  try {
    const { path } = req.query
    if (!path || path === "") {
      return res.status(400).json({ "status": "Fail: path not provided" })
    }

    // Transfers the file at the given path. Sets the Content-Type response HTTP header field based on the filename's extension.
    // VULNERABLE CODE: Local File Inclusion; can be made to transfer sensitive information such as keys
    return res.status(200).sendFile(path as string)
  } catch (error) {
    console.error(error)
    return res.status(200).json({ "status": "Internal Server Error" })
  }
}
```

Fig 8: Local File Inclusion Vulnerability

The code snippet shown above is for the `/user/lfi` endpoint. It is a hypothetical endpoint that takes in the path of a file as a query parameter and then sends the file back to the client. It is vulnerable to the Local File Inclusion vulnerability as it allows the user to gain access to local files on the server, such as those that include credentials such as `/etc/passwd` or ssh keys, especially if the server is running as the root user.

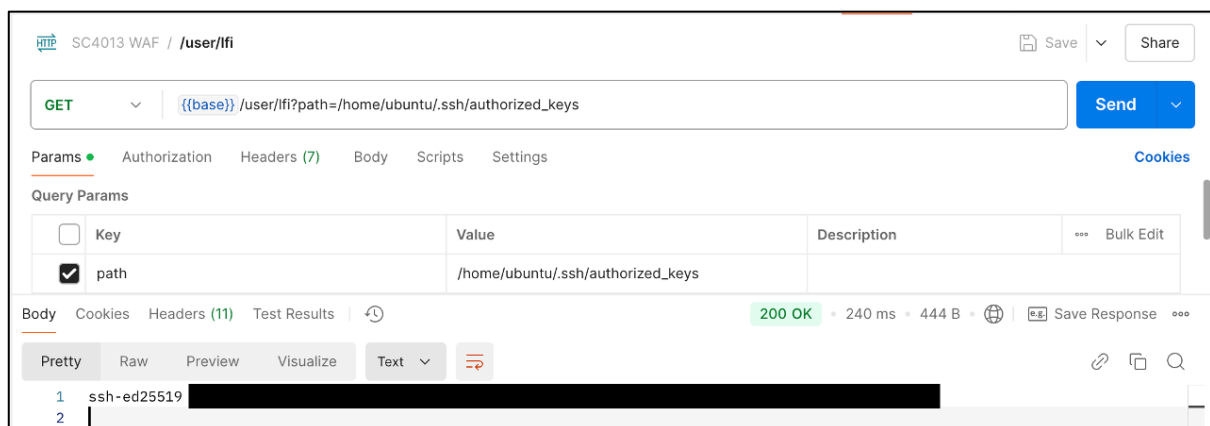


Fig 9: Successful Retrieval of ssh keys via exploiting the LFI vulnerability

Fig 9 shows the successful retrieval of the authorized ssh keys, which can give unauthorized users access to the server.

If a request matches all the statements (AND)		
Statement 1	Statement 2	Action
Field to match URI path Positional constraint Contains string Search string /user/lfi Text transformations • None (Priority 0)	Field to match All query parameters Regular Expression ^\/(?[a-zA-Z0-9_\.]+\?)*[a-zA-Z0-9_\.]+\$ Text transformations • None (Priority 0)	The action to take when a web request matches the rule statement. Action Block Custom response code 404 Add labels -

Fig 10: LFI Rule

Fig 10 shows the WAF rule created to target this vulnerability. If the request's URI contains /user/lfi and matches the regex expression provided, which checks for Linux file paths of the form (eg /etc/passwd or ~/.ssh/authorized_keys), then the request will be rejected and given a custom response code of 404 (for debugging purposes).

4. Remote File Inclusion

```
import { NextFunction, Request, Response } from "express"
import fetch from "node-fetch"

export const remoteFileInclusionHandler = async (req: Request, res: Response, next: NextFunction) => {
  try {
    const { url: url } = req.query
    // console.log("url", url)
    if (!url || url === "") {
      return res.status(400).json({ "status": "Fail: url not provided" })
    }

    // Fetch external resource
    // VULNERABLE CODE: Using unvalidated URL from user input, which leads to RFI vulnerability
    const response = await fetch(url as string);
    // console.log("response", response.body)
    const data = await response.json()
    console.log("data", data)

    return res.status(200).json({ "data": data });
  } catch (error) {
    console.error(error)
    return res.status(200).json({ "status": "Internal Server Error" })
  }
}
```

Fig 11: RFI Vulnerability

The code snippet shown above is for the /user/rfi endpoint. This hypothetical endpoint takes in any user-supplied URL as a query string and executes an API call to the URL, without carrying out any input sanitization. Attackers can exploit this endpoint by passing in a malicious URL, which will cause the server to fetch a malicious script and then send it to the client. Should the client execute the malicious script unintentionally, this can result in the installation of malware or other malicious agents on the client's device.

If a request matches all the statements (AND)		
Statement 1	Statement 2	Action
Field to match URI path Positional constraint Contains string Search string /user/rfi Text transformations • None (Priority 0)	Field to match All query parameters Positional constraint Contains string Search string https:// Text transformations • None (Priority 0)	The action to take when a web request matches the rule statement. Action Block Custom response code 407 Add labels -

If a request matches all the statements (AND)		
Statement 1	Statement 2	Action
Field to match URI path Positional constraint Contains string Search string /user/rfi Text transformations • None (Priority 0)	Field to match All query parameters Positional constraint Contains string Search string http:// Text transformations • None (Priority 0)	The action to take when a web request matches the rule statement. Action Block Custom response code 405 Add labels -

Fig 12: Top) Rule blocking https requests, Bottom) Rule blocking http requests

Fig 12 shows 2 WAF rules written intentionally to prevent such vulnerabilities whereby malicious http or https URLs are provided to the server. If the requests are sent to the /user/rfi endpoint and contain either “http://” or “https://”, the requests will be blocked. Given that remote fetches were not a key functionality of the online book repository, and the endpoint was created merely to test the capabilities of the WAF, the proposed 2 WAF rules are likely to be acceptable.

However, should the application require the functionality to make remote fetches, then the proposed rules would not be appropriate as it would prevent the application from receiving any such requests. In such a scenario, server-side input sanitization and even malware detection would be a more appropriate solution.

5. SQL Injection

```
export const uploadHandler = async (req: Request, res: Response, next: NextFunction) => {
  let uploadedFile: UploadedFile | null = null
  let id = ""
  try {
    // check if file was submitted
    console.log("req.files", req.files)
    if (!req.files || !req.files.file) return res.status(400).json({ "status": "No files were uploaded" });

    // check if title was provided
    const { title, author } = req.body
    if (!title || title === "") return res.status(400).json({ "status": "Title not provided" });
    if (!author || author === "") return res.status(400).json({ "status": "Author not provided" });

    uploadedFile = req.files.file as UploadedFile

    id = Book.genId(title, author)

    // check if the file exists
    const isExist = await Book.findOne({ where: { id: id } })
    if (isExist) return res.status(400).json({ "status": "Book already exists" });

    // insert the file into the db
    // VULNERABLE CODE: SQL Injection
    await sequelize.query(`
      INSERT INTO "books" ("id", "title", "author")
      VALUES('${id}', '${title}', '${author}')
    `, {
      raw: true,
      type: QueryTypes.INSERT
    })
  }
}
```

SQL Copy Caption ...

```
INSERT INTO "books" ("id", "title", "author")
VALUES('abc', 'placeholder', 'placeholder'); DROP TABLE books; --')
```

Fig 13: Top) Code containing SQL Injection vulnerability.
Bottom) Sample SQL injected code

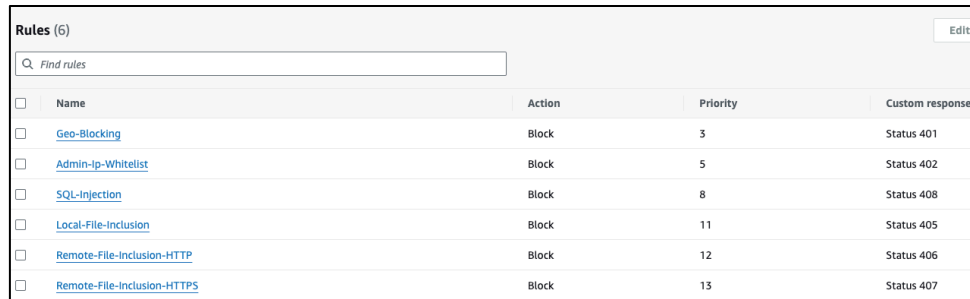
Fig 13 shows the code snippet for the `/user/upload` endpoint. The endpoint is vulnerable to SQL Injection attacks, as it does not sanitize the user inputs, and uses the raw inputs to craft the SQL query. An example of a malicious payload is also shown above.

If a request matches all the statements (AND)		
Statement 1	Statement 2	Action
Field to match URI path	Field to match Body	The action to take when a web request matches the rule statement.
Positional constraint Contains string	Attack Type Contains SQL injection attacks	Action Block
Search string /user/upload	Sensitivity Level Low	Custom response code 408
Text transformations • None (Priority 0)	Text transformations • None (Priority 0)	Add labels -
	Override handling Match - Treat the web request as matching the rule statement	

Fig 14: SQL Injection rule

To prevent SQL injection attacks, a WAF rule is to check for any SQL injection attacks on the /user/upload endpoint.

Rule Priority



<input type="checkbox"/>	Name	Action	Priority	Custom response
<input type="checkbox"/>	Geo-Blocking	Block	3	Status 401
<input type="checkbox"/>	Admin-ip-Whitelist	Block	5	Status 402
<input type="checkbox"/>	SQL-Injection	Block	8	Status 408
<input type="checkbox"/>	Local-File-Inclusion	Block	11	Status 405
<input type="checkbox"/>	Remote-File-Inclusion-HTTP	Block	12	Status 406
<input type="checkbox"/>	Remote-File-Inclusion-HTTPS	Block	13	Status 407

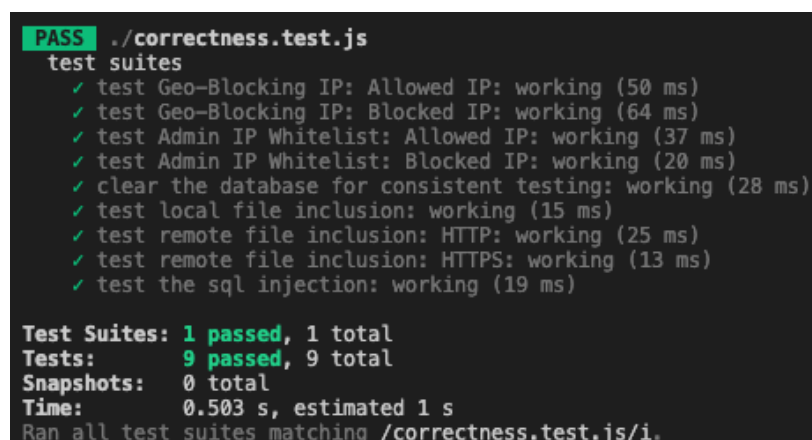
Fig 15: All Rules

Fig 15 shows the order of the rules. Rules with higher priority are applied first, followed by rules with lower priority. The rules were ordered such that more “general” rules are applied first, to reduce the attack surface as quickly as possible. Hence, rules for Geo-Blocking and Admin IP whitelist are applied first, before specific rules that target individual endpoints.

Testing

After developing the server code and WAF rules, test suites were created to test for the correctness and performance impact of the WAF rules. These test suites were written using Jest.

Correctness Test



```
PASS ./correctness.test.js
  test suites
    ✓ test Geo-Blocking IP: Allowed IP: working (50 ms)
    ✓ test Geo-Blocking IP: Blocked IP: working (64 ms)
    ✓ test Admin IP Whitelist: Allowed IP: working (37 ms)
    ✓ test Admin IP Whitelist: Blocked IP: working (20 ms)
    ✓ clear the database for consistent testing: working (28 ms)
    ✓ test local file inclusion: working (15 ms)
    ✓ test remote file inclusion: HTTP: working (25 ms)
    ✓ test remote file inclusion: HTTPS: working (13 ms)
    ✓ test the sql injection: working (19 ms)

Test Suites: 1 passed, 1 total
Tests: 9 passed, 9 total
Snapshots: 0 total
Time: 0.503 s, estimated 1 s
Ran all test suites matching /correctness.test.js/i.
```

Fig 16: Correctness Test

The correctness test suites were written for each individual WAF rules. For each WAF rule, success and fail test cases were written. For instance, for the Geo-Blocking rule, IP addresses for Singapore were tested in the success test case, and US IP addresses were tested in the fail test case. To check if the WAF rules were applied correctly, the received response code was

checked against the desired response code. Each rule was assigned a custom response code for testing purposes as seen in Fig 15.

Performance Impact Analysis

```
> sc4013-server@1.0.0 test
> cd ./test && npx jest performance.test.js

console.log
  duration (with ec2): 70.18 ms
    at Object.log (test/performance.test.js:35:13)

console.log
  duration (with waf): 82.32 ms
    at Object.log (test/performance.test.js:54:13)

PASS ./performance.test.js (15.489 s)
  test suites
    ✓ ec2 (7053 ms)
    ✓ waf (8237 ms)

Test Suites: 1 passed, 1 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 15.532 s, estimated 28 s
Ran all test suites matching /performance.test.js/i.

var timings = 0

for (var i = 0; i < 100; i++) {
  const start = Date.now()
  const { body, status: uploadStatus } = await uploadFile(ec2, Number(i))
  expect(uploadStatus).toEqual(200)
  const { status: downloadStatus } = await downloadFile(ec2, body["id"])
  expect(downloadStatus).toEqual(200)
  const { status: deleteStatus } = await deleteBook(ec2, body["id"])
  expect(deleteStatus).toEqual(200)
  const end = Date.now()
  timings += end - start
}

console.log("duration (with ec2):", timings / 100, "ms")
```

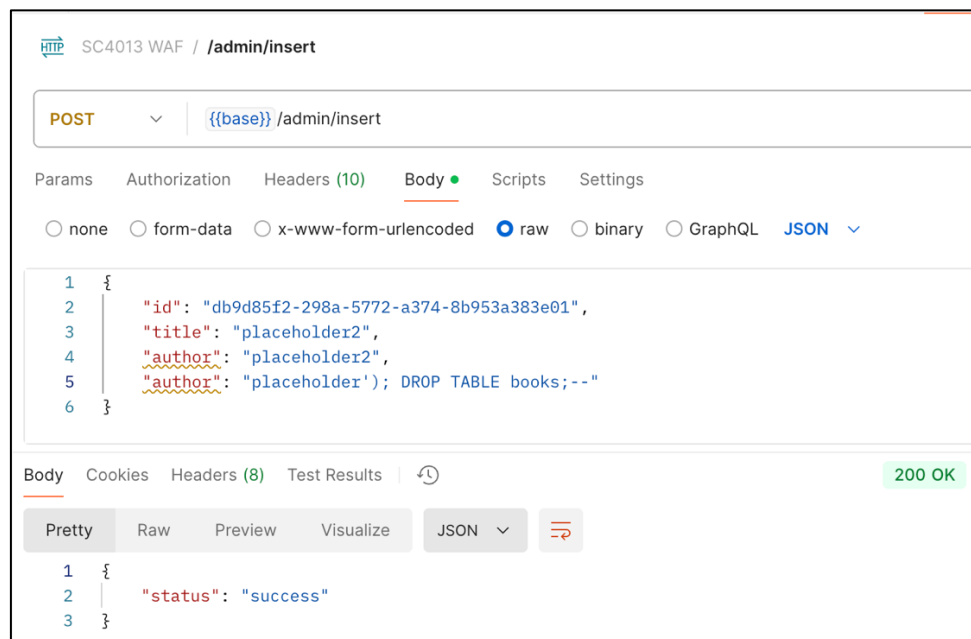
Fig 17: Top) Performance Test Result,
Bottom) Snippet of Performance test suit

To test the impact of the WAF rules on performance, code was written to compare the time taken to call APIs via the WAF and directly on the EC2 instance. For both the WAF and EC2 instance, the average time taken to complete 100 iterations of uploading a book, downloading a book, and deleting the book by calling the respective APIs was measured.

The code above shows the results obtained. The average time taken without the WAF is 70.18ms, whereas the average time taken with the WAF is 82.32ms. This result suggests that the WAF causes a 14.74% increase in the time required.

Rule Bypass and Mitigation:

Invalid JSON attack



```
0|sc4013-w | Executing (default): INSERT INTO "books" ("id", "title", "author")
0|sc4013-w | VALUES('db9d85f2-298a-5772-a374-8b953a383e01', 'placeholder2', 'placeholder'); DROP TABLE books;--'
```

Fig 18: Top) Success response from the server when attempting an Invalid JSON attack, Bottom) Server-side logs showing the SQL command that was executed on the server side

During an invalid JSON attack, the attacker includes an invalid JSON in the request body. One way to create an invalid JSON is to create duplicate fields in a JSON object, as shown in the figure above.

The mechanism of the attack can be analyzed from 2 perspectives: 1) the WAF, 2) the server. From the WAF perspective, the default behaviour of AWS WAF when dealing with invalid request bodies is to not apply the WAF rules, therefore, such requests will pass through the WAF. From the server perspective, backend frameworks such as Node.js Express and Python Flask use the last value of the duplicated keys when retrieving the request body. Hence, when the request shown in Fig 18 is received by the server, the malicious SQL injected code will be used and executed, causing damage to the server (Menin, 2023).

Mitigation

How AWS WAF should handle the request if the JSON in the request body is invalid

☐ None
Evaluate the tokens that AWS WAF parsed before encountering the invalid JSON

☐ Evaluate as plain text
Apply transformations and matching criteria to the request body as plain text

☐ Match
Return a match for the request

☒ No match
Return no match for the request

Fig 19: AWS WAF configuration for invalid request body behaviour

To prevent invalid JSON attacks, AWS WAF allows users to configure the behaviour when invalid request bodies are received. Depending on the rule that was written, users can set the default to either match or not match the rule. To prevent these malicious requests from getting through, users should set “Match” if the rule is not a negation and if matching the rule allows for the request to pass through, vice versa.

WAF Strengths and Limitations:

The strength of the WAF is that it can apply rules to all requests that are reaching the application all at once, rather than having to implement the rules in code on individual endpoints. In addition, it covers a wide range of attacks such as SQL Injection and XSS.

However, there are several limitations to the WAF. The following are some of the notable ones. Firstly, WAF rules that apply to specific API endpoints will need to be updated once the APIs change. Depending on the use case, if there are frequent changes to the APIs, it can be tedious to constantly update and test the WAF rules. Secondly, AWS WAF can only inspect request bodies of Content-Type “*application/json*” and “*plain/text*”. Requests with other Content-Type, such as “*multi-part/form-data*” will be treated as an invalid request body. Thirdly, AWS WAF only able to inspect the request body up till 8KB. Lastly, it is unable to perform certain security checks, such as malware analysis if the application is designed to allow users to upload files.

Conclusion and Recommendations

Despite the limitations mentioned above, WAF remains a useful first layer of defense in any security architecture. However, it should not be the only layer of defense, given that there are ways in which it can be bypassed. It is best used to enforce general rules that do not change frequently, to reduce the number of further updates required when the API endpoints change.

Additional security features should be implemented on the application level to suit the use case, especially if they are endpoint specific. For example, input sanitization, authentication and authorization systems, etc. Only by having a comprehensive suite of security tools implemented would we be able to achieve “security in depth”.

In addition, users should also be mindful of the need to balance between performance and security. As shown earlier, the usage of WAF rules will result in a slight impact on performance.

References

Menin, A. (2023, July 26). *AWS WAF Bypass: invalid JSON object and unicode escape sequences*. Sicuranext Blog. <https://blog.sicuranext.com/aws-waf-bypass/>

Annex

The following annex contains WAF rules that were experimented with but did not work as intended, and were thus removed eventually.

1. File upload size restriction

If a request matches all the statements (AND)		Action The action to take when a web request matches the rule statement.
Statement 1	Statement 2	
Field to match URI path Positional constraint Contains string Search string /user/upload Text transformations • None (Priority 0)	Field to match Body Match type Size greater than Size 1,048,576 Text transformations • None (Priority 0)	Action Block Custom response code 402 Add labels -

2. File upload extension check

If a request matches all the statements (AND)		Action The action to take when a web request matches the rule statement.
Statement 1	Statement 2	
Field to match URI path Positional constraint Contains string Search string /user/upload Text transformations • None (Priority 0)	Field to match Body Regular Expression (?:.pdf) Text transformations • None (Priority 0)	Action Allow Custom request - Add labels -

3. Rate Limiting

Rate-limiting criteria	Scope-down statement	Action The action to take when a web request matches the rule statement.
Request aggregation IP address in header Rate limit 30 Evaluation window 1 minute (60 seconds) IP address in header X-Forwarded-For Fallback for missing IP address No match	Field to match URI path Positional constraint Contains string Search string http://sc4013-alb-257362673.ap-southeast-1.elb.amazonaws.com:80/ Text transformations • None (Priority 0)	Action Block Custom response code 429 Add labels -