Insecure Direct Object References

Talha Eroglu, Ilgaz Senyuz, Okan Yıldız

January 4, 2023

Abstract

Insecure Direct Object References (IDOR) are a type of vulnerability that occurs when an application exposes direct object references, such as a file path or database key, to unauthorized users. This can allow attackers to bypass security controls and access sensitive information, such as user data or financial records, without proper authentication. IDOR vulnerabilities can arise due to a lack of proper access controls or when an application trusts user-supplied input without adequately validating it. In this article, we will provide examples of unsecure code that is vulnerable to IDOR attacks, and demonstrate how these vulnerabilities can be exploited. To prevent IDOR vulnerabilities, it is important to implement robust access controls and sanitize user input to ensure that only authorized users can access sensitive objects. Additionally, regularly testing and monitoring applications for IDOR vulnerabilities can help to identify and mitigate potential threats.

Contents

1 Introduction to Insecure Direct Object References (IDOR)

IDOR vulnerabilities are a serious concern for organizations and individuals alike, as they can result in the compromise of sensitive data and financial loss. These vulnerabilities occur when an application exposes direct object references, such as file names or database keys, in its user interface, allowing attackers to manipulate them to access unauthorized data or functionality. IDOR vulnerabilities can be found in a variety of applications, including web applications, mobile apps, and desktop software.

To prevent IDOR vulnerabilities, developers must ensure that proper security measures are in place for object references within their applications. This includes implementing access controls to verify that a user is authorized to access a particular object, as well as encrypting object references to make them more difficult to manipulate. It is also crucial for developers to keep their software up to date, as new vulnerabilities may be discovered and patches released to fix them.

The consequences of IDOR vulnerabilities can be severe, including unauthorized access to sensitive data, financial loss, and damage to an organization's reputation. It is therefore essential for organizations to take proactive measures to secure their applications and protect against IDOR attacks. This includes implementing strong security measures, regularly updating software, and providing education and training to employees to help them identify and prevent potential vulnerabilities. By taking these steps, organizations can effectively safe-

guard their data and reduce the risk of suffering from the consequences of IDOR vulnerabilities.

2 Common Types of IDOR

IDOR attacks can be classified in various ways, and it is possible to divide them into different categories based on the method of attack (URL tampering, body manipulation, etc.) and the types of direct reference objects. In our case, We have chosen to divide IDOR attacks into two categories: Direct Reference to Database Objects and Direct Reference to Static Files.

2.1 Direct Reference to Database Objects

Direct reference to database objects is a type of (IDOR) vulnerability that occurs when an application references database objects using user-supplied input without proper validation. This can allow an attacker to access sensitive data or manipulate database objects by altering the user-supplied input.

Imagine that you have registered on a shopping website and have provided your personal information such as your address, email, and phone number. You are then redirected to a page where you can review and edit your information. On this page, you may see a link in the browser bar that looks something like this:

`example.com/user/details?id=2023`

The "2023" ID is a reference to an object that is stored in the database

when your account is created. In this example, your account is associated with the ID "2023".

However, what happens if you try to change the ID in the URL? It is possible to access the details of other users on the website simply by altering the ID in the URL. For instance, if you change the ID to "2022", you may be able to view the details of another user's account. This demonstrates that there is an "Insecure Direct Object References" vulnerability on the website.

To further illustrate this vulnerability, consider the following URLs:

```
example.com/user/profile?id=2023 (Your Account)

example.com/user/profile?id=2022 (Victim's Account)
```

By exploiting the IDOR vulnerability, it is possible to retrieve information from the victim's account simply by modifying the ID in the URL. The server does not check user permissions when requests are made, meaning that there is a broken access control on the website. As a result, it is possible to read, edit, and delete the personal information of all registered users by altering the ID in the URL.

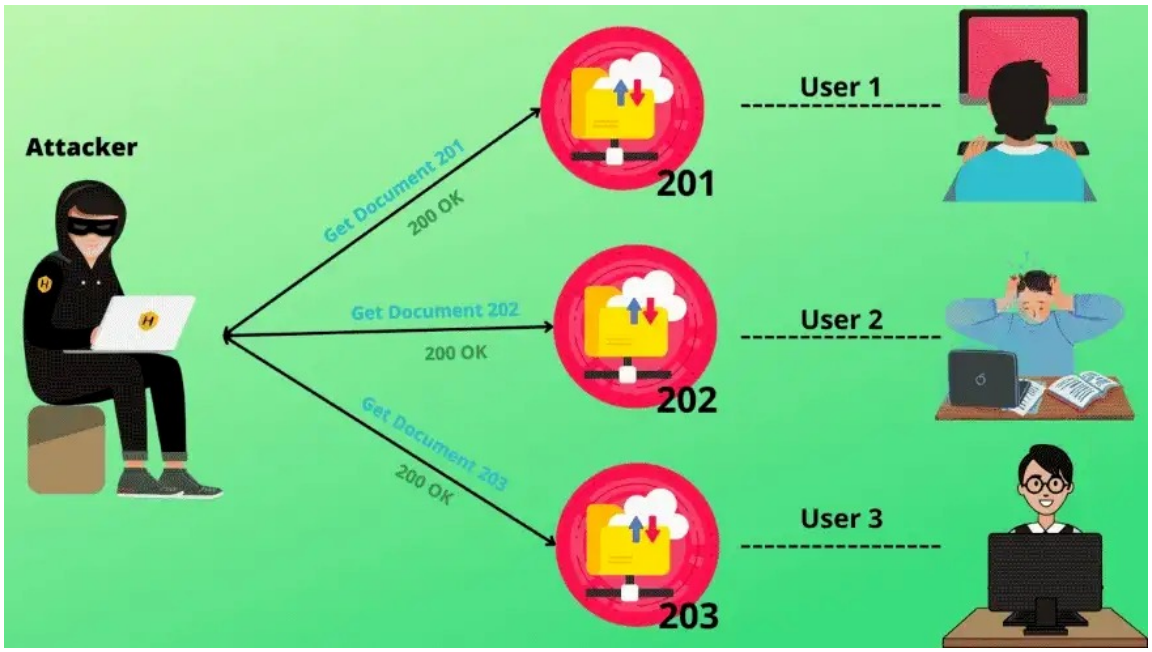

Figure 1: Direct Reference to Database Objects Scenario

2.2 Direct Reference to Static Files

Direct reference to static files is another type of IDOR vulnerability that occurs when an application references static files using user-supplied input without proper validation. This can allow an attacker to access sensitive files.

Now assume that a website stores sensitive information in /static/ files on the server-side file system. For example, when you make a purchase on the website, a receipt.pdf file may be created. If you want to retrieve this file, you can download it using a link that looks something like the following:

```
example.com/static/receipt/205.pdf
```

However, an attacker can exploit this vulnerability by simply modifying the filename in the URL like below:

`example.com/static/receipt/200.pdf`

By doing so, the attacker may be able to retrieve a receipt created by another user and potentially gain access to sensitive information such as user credentials.

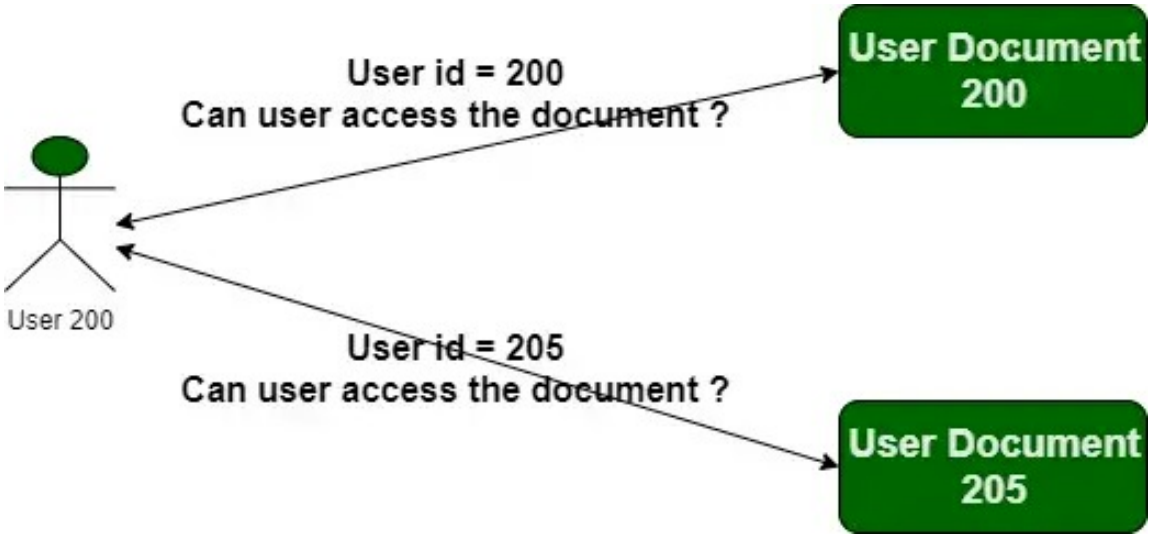

Figure 2: Direct Reference to Static Files

3 Risks Caused by IDOR

IDOR attacks can have serious consequences for both website owners and users. Some of the risks associated with IDOR include:

3.1 Unauthorized Access to Sensitive Information

Insecure direct object references (IDOR) can occur when an application references database objects or static files using user-supplied input without proper validation. This can allow an attacker to access sensitive information by altering the user-supplied input. For example, an attacker could use an IDOR vulnerability to view other users' account details, financial information, or personal data. This can lead to serious privacy breaches and harm to individuals affected by the breach.

3.2 Data Manipulation

In addition to allowing unauthorized access to sensitive information, IDOR vulnerabilities can also allow attackers to manipulate data stored in the application's database. This can include modifying, deleting, or adding new data. Data manipulation attacks can have serious consequences, such as corrupting important records or disrupting business operations. For example, an attacker could use an IDOR vulnerability to change a user's account balance or personal information, or delete critical data from the database.

3.3 Direct File Access

IDOR vulnerabilities can also allow attackers to directly access files stored on the application's server. This can include sensitive files such as configuration files or log files, as well as static files such as images or documents. Direct file access attacks can lead to sensitive information leakage and potentially allow attackers to execute malicious code on the server. For example, an attacker could use an IDOR vulnerability to download a server's configuration file, which could contain sensitive information such as database passwords. Alternatively, an attacker could use an IDOR vulnerability to upload a malicious script and execute it on the server.

3.4 Account Takeover

Finally, IDOR vulnerabilities can be exploited to gain unauthorized access to user accounts. This can allow attackers to perform actions on behalf of the compromised user, such as making purchases or accessing sensitive information. Account takeover attacks can have serious consequences for both the affected user and the application. For example, an attacker could use an IDOR vulnerability to take over a user's account and make unauthorized purchases using the user's financial information. Alternatively, an attacker could use a compromised account to access sensitive information or perform actions that damage the user's reputation.

4 How to Exploit IDOR Vulnerabilities

4.1 Direct Reference to Database Objects

We have an e-commerce application built with React and Django stack running on localhost with ports 3000 and 8000. Almost all of the code is taken from the GitHub account JustDjango, you can find the relevant repository in the references section. The website includes basic features such as registering, logging in, adding products to the cart, and getting billing and delivery information from the users.

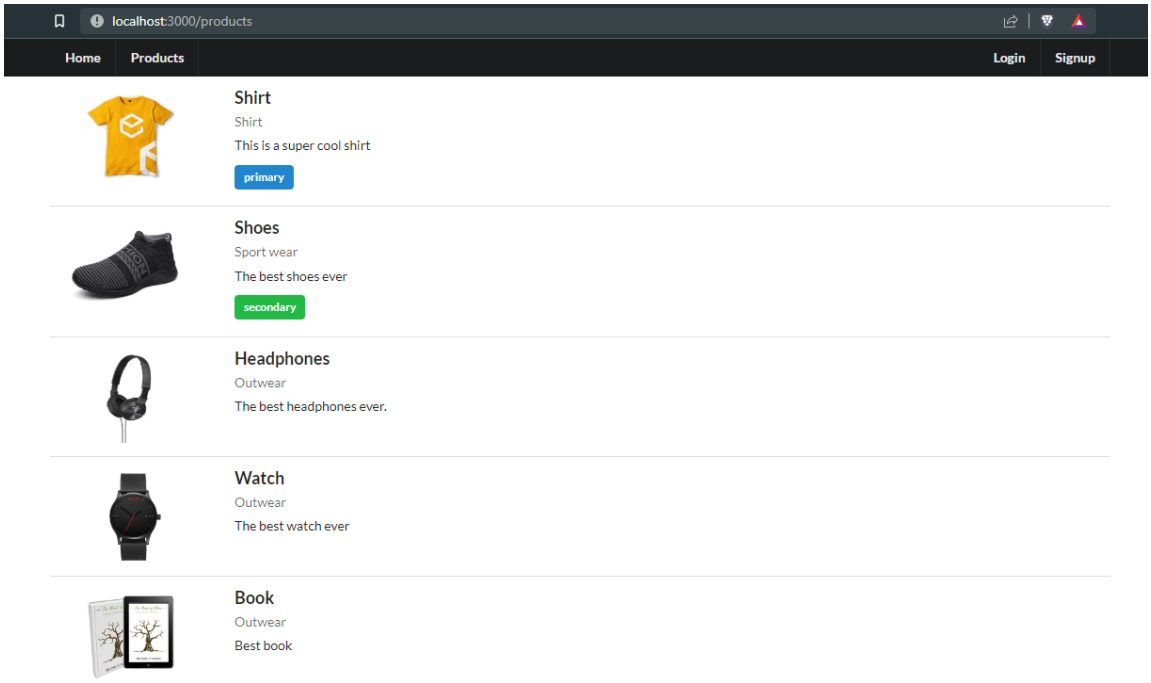

Figure 3: E-commerce website

We begin by creating two accounts in order to verify that we can access restricted information that we should not. We registered to the e-commerce platform by creating 2 different users named talha and

ilgaz. After creating the accounts, We logged in as user "ilgaz, and start examining the requests and responses from the network tab. In network tab we can easily obtain the user's ID by accessing the

`http://127.0.0.1:8000/api/user-id/`

endpoint in the profile menu on the website.

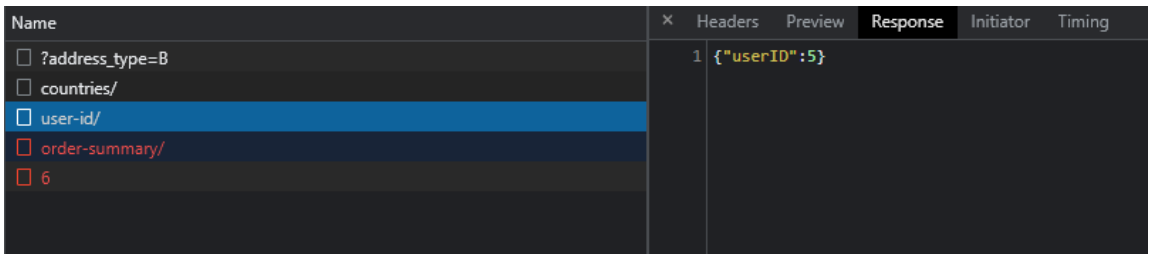

Figure 4: UserID of "ilgaz" obtained from network tab

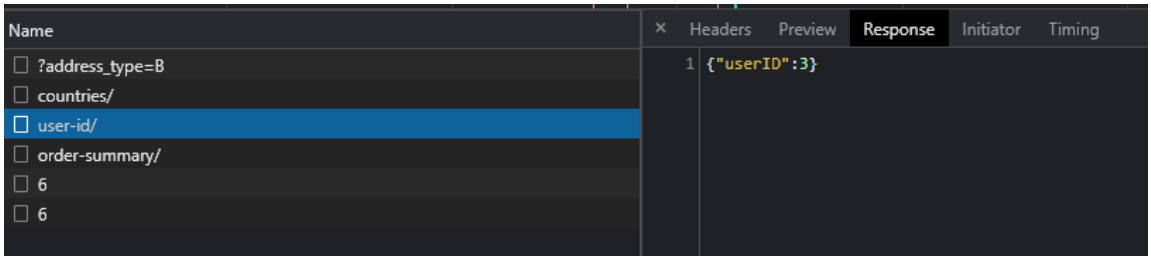

Figure 5: UserID of "talha" obtained from network tab

With the user ID of the "ilgaz" user in hand, we also retrieved the user ID of the "talha" user by repeating the same steps. By understanding the way user IDs are assigned in an auto-incremented fashion, we were able to formulate strategies for conducting an attack. Even if we could not detect any pattern, we could find valid users using an enumeration attack, which basically is a brute-force method to check if certain data exists on a web server database.

Let's go back to our topic. As shown in the above figures, when we retrieve the user ID for the "talha" user, we can see that user IDs are assigned to users in an auto-incremented way.

From now on, we will use the features of our website to create different shipping and billing addresses and orders for two users, and begin searching for IDOR vulnerabilities.

As can be seen in the following figures, we have created different addresses and orders.

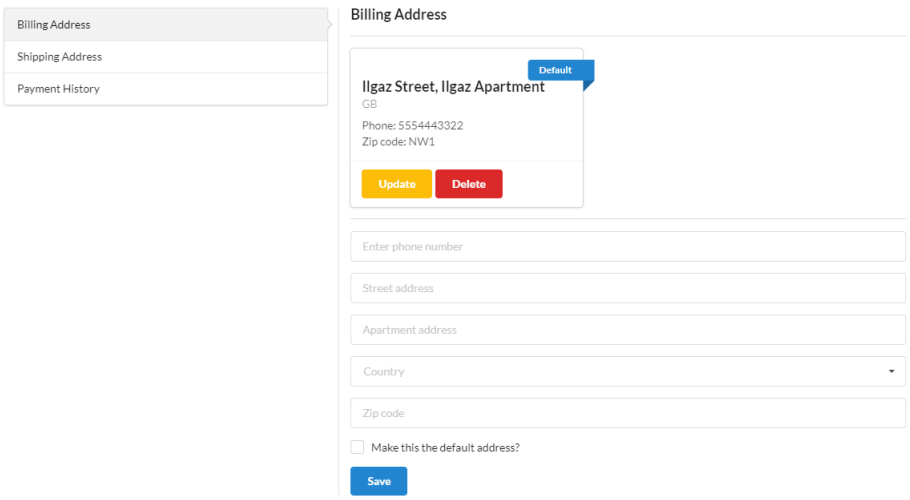

Figure 6: Billing Addresses of Ilgaz

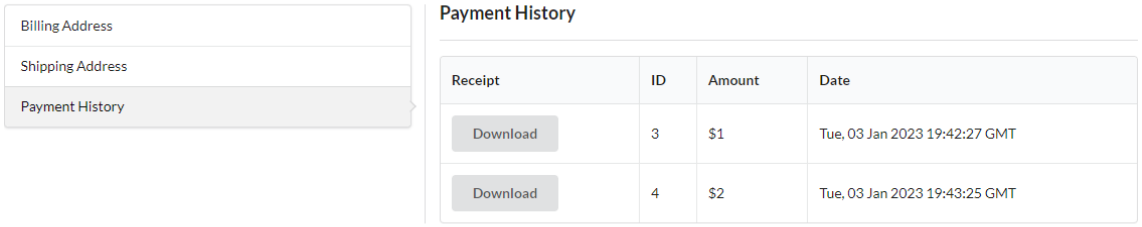

Figure 7: Payment History of Ilgaz

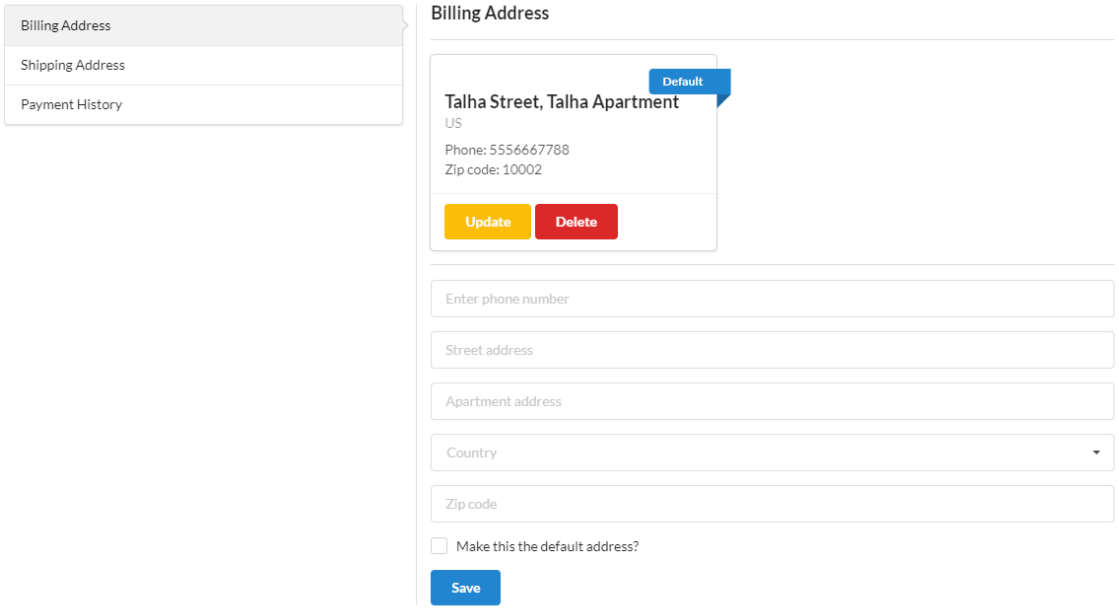

Figure 8: Billing Addresses of Talha

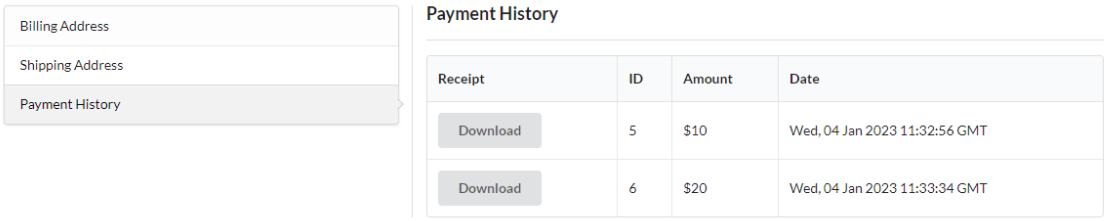

Figure 9: Payment History of Talha

After logging in as the "talha" user, we examine the requests made using Burp Suite. We attempt to access "ilgaz" users' data using related user ID. We open the intercept in the proxy tab in Burp Suite and click on the profile menu on the website. We see that the data in the profile is retrieved by a GET request to

`localhost:8000/api/addresses`

endpoint As shown in the figure, when we examine the request, we see that no data is sent except for the CSRF token and address type. We understand that the relevant user is matched with his/her own profile via CSRF Token and a response is sent accordingly.

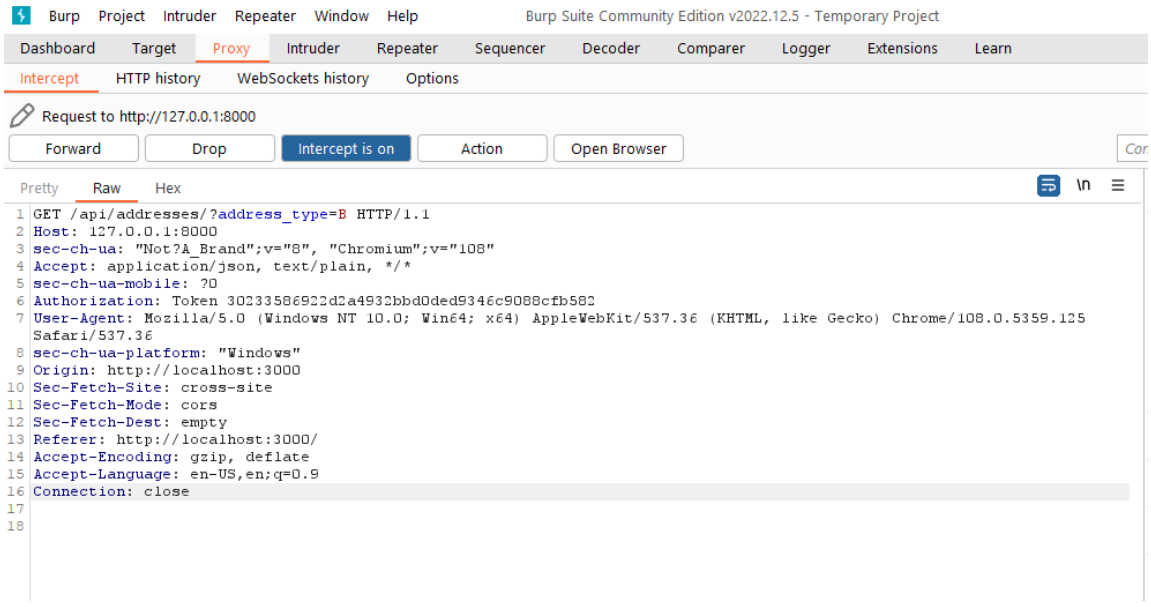

Figure 10: Address request

Leaving the address endpoint behind, we continue by examining other requests via burp and browser. As shown in the figure below, we can see that the requests made to the "order-summary" and "address" endpoints are validated using the CSRF token and the response is sent in the same way.

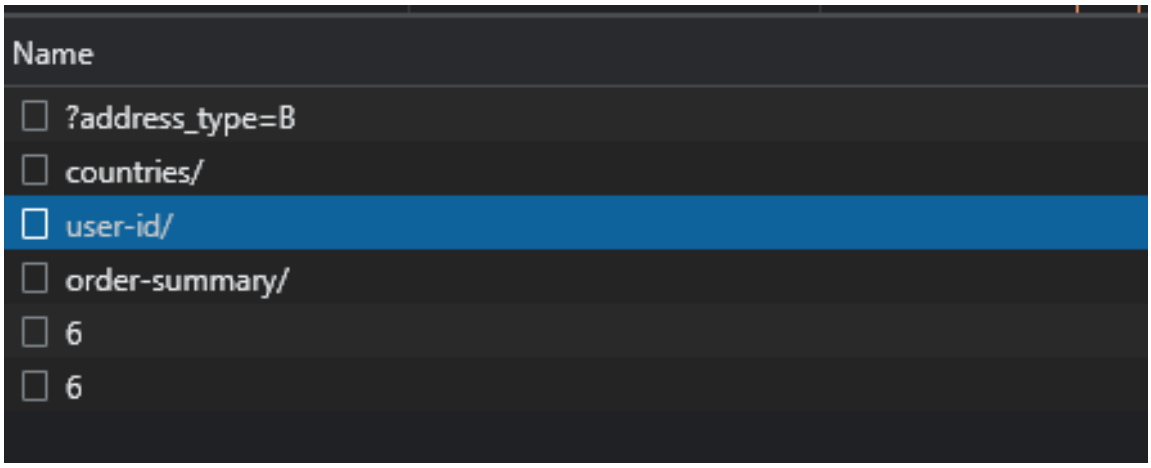

Figure 11: Other requests

At this point, we will create and update billing addresses while Burpsuite intercept is active.

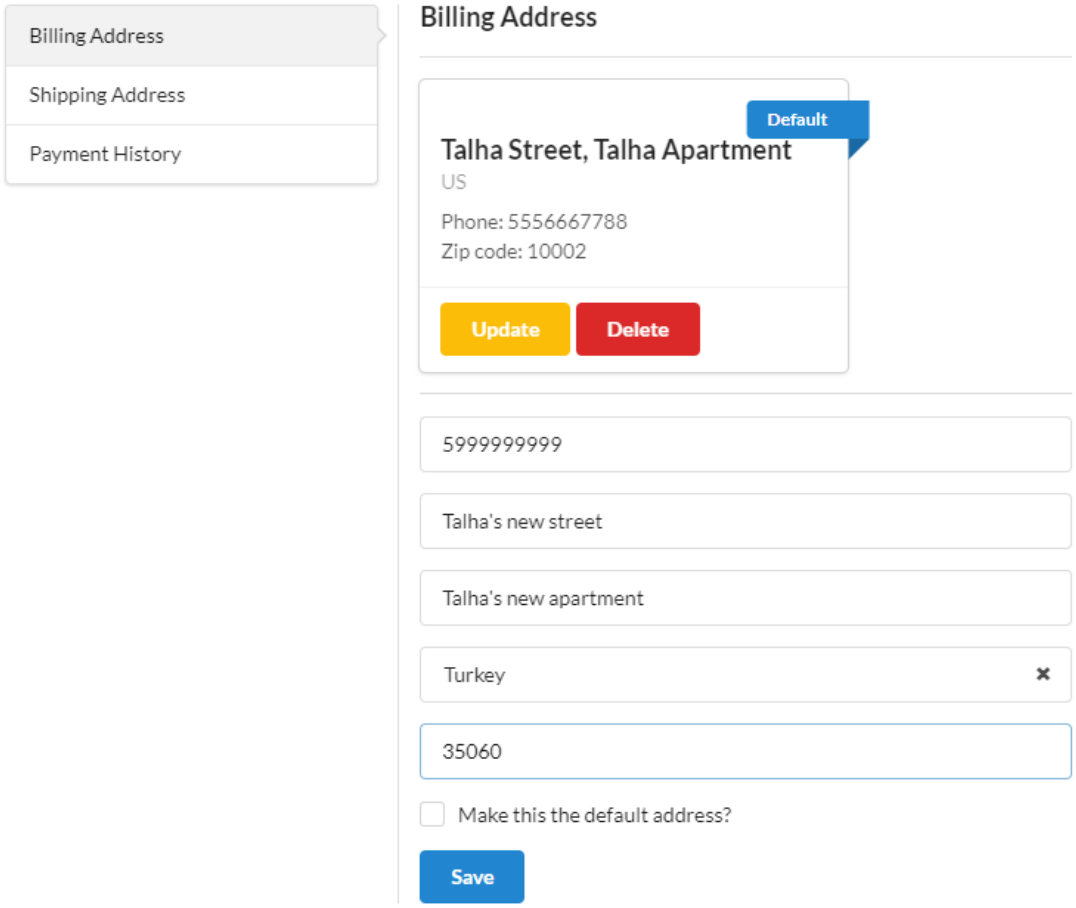

Figure 12: Creating new billing address for user "talha"

Let's turn intercept on and try to create a new Billing Address for user "talha". As you can see in the below figure we are passing user and related address information to the request we sent to /api/addresses/create/.

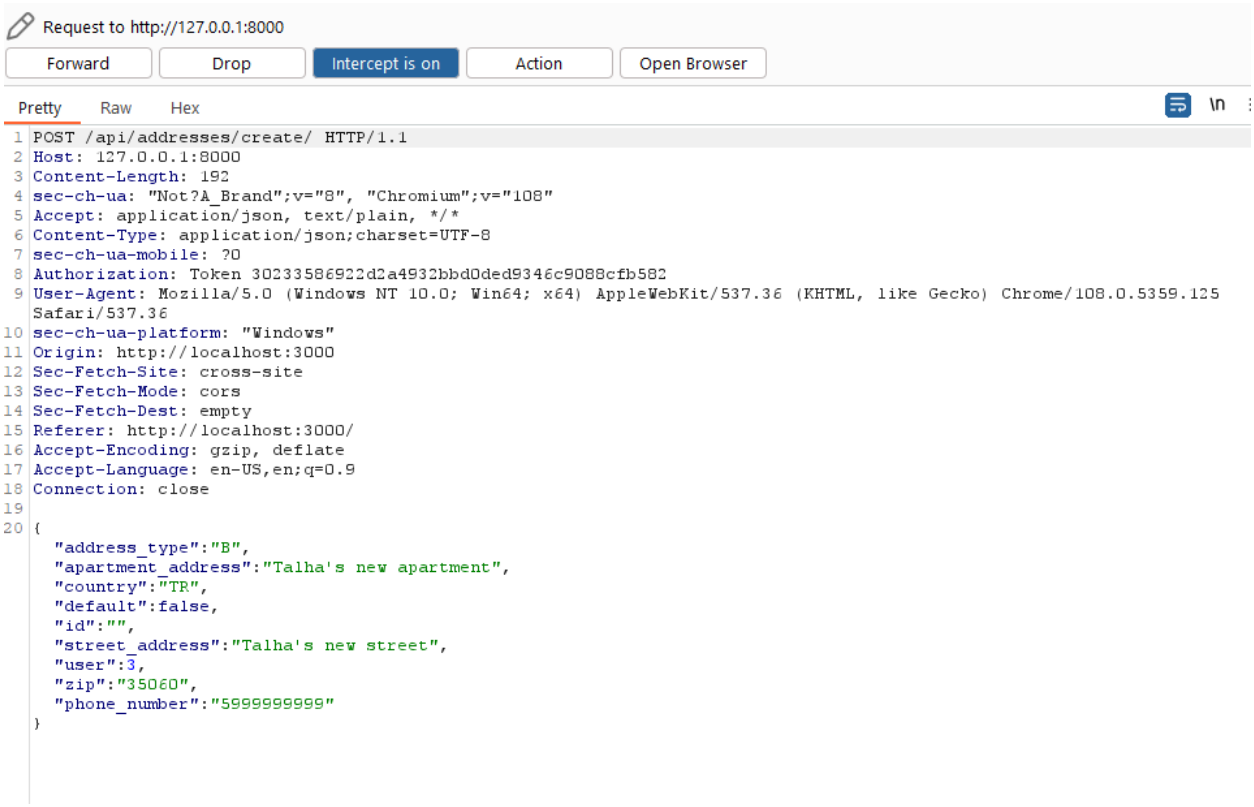

Figure 13: Intercepted request of new billing address

Knowing the user ID of "ilgaz" (which we had determined was assigned incrementally), we changed the user ID for "talha" (3) to the user ID for "ilgaz" (5). After making this change, we submitted the request from the top left button on the website. While we were able to see a success message on the website, the newly created address did not appear for the "talha" user. This suggests that the new billing address is indeed created, but for the user "ilgaz".

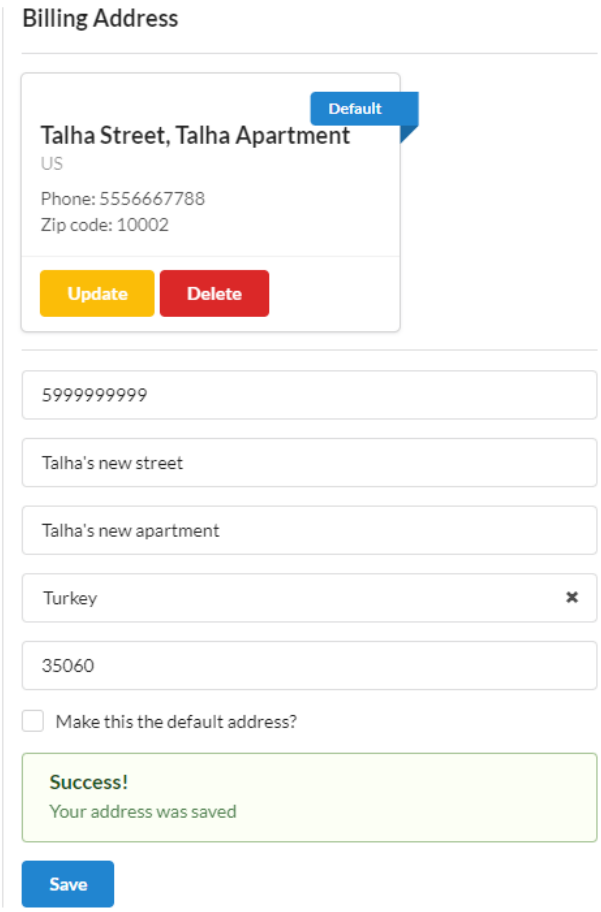

Figure 14: Successfuly created message

Now let's log into the other account and see if our new address is created under the other user's profile.

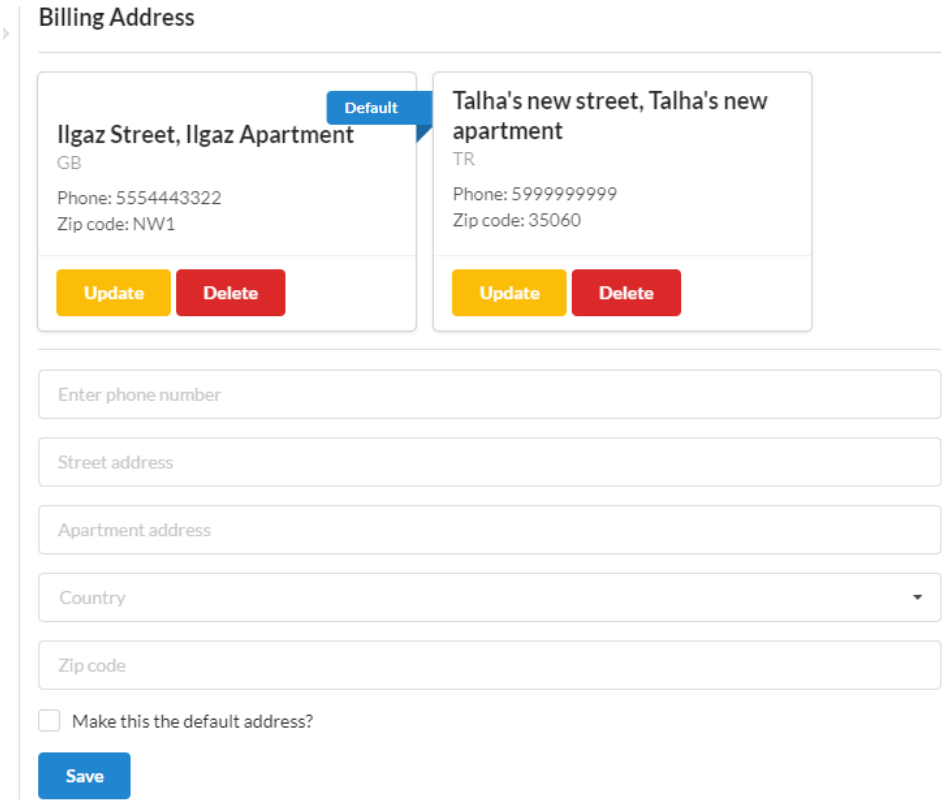

Figure 15: Ilgaz's account, address section

Bingo! It's important to note that we were able to manipulate the addresses in the "ilgaz" user's profile from "talha" user without using any private information. By providing integer payloads through Burp Suite and blindly using the UserId attribute, we could create this address information for any user.

At the beginning of the exploit part, we could not access the profiles of other users. But right now, we have a much more powerful weapon than reading data. It is writing data! For the time being,

we can create profile data under another user. What if we could also update the address information? If we can update address information, we can change the user ID of any address to our own user ID and may associate any information with our "talha" user. This would allow us to access all email addresses, phone numbers, payment, and billing addresses in the system.

Let's try to update random addresses so that they belgons to the user "talha", by using enumeration method. We will demonstrate how to use enumerated payloads to associate each address found in the system, with the "talha" user."

First, to use update address endpoint, we will simply intercept update billing address operation in "talha" user.

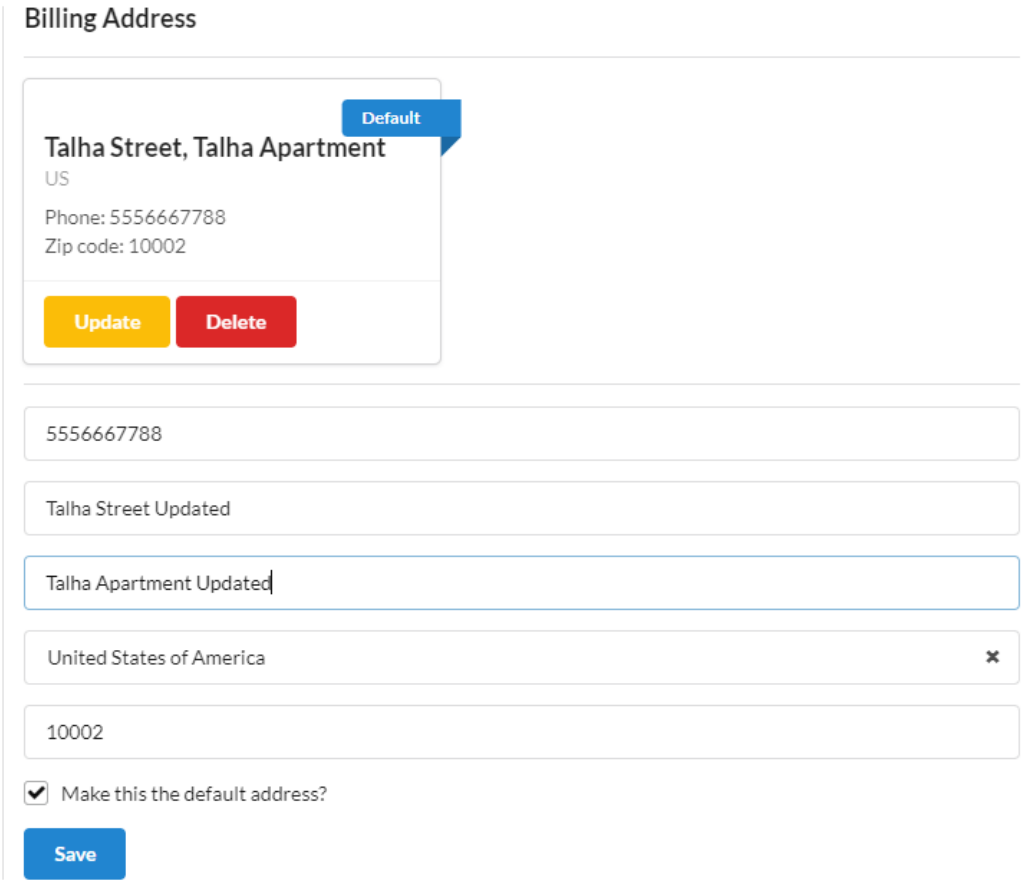

Figure 16: Updating billing address of user "talha"

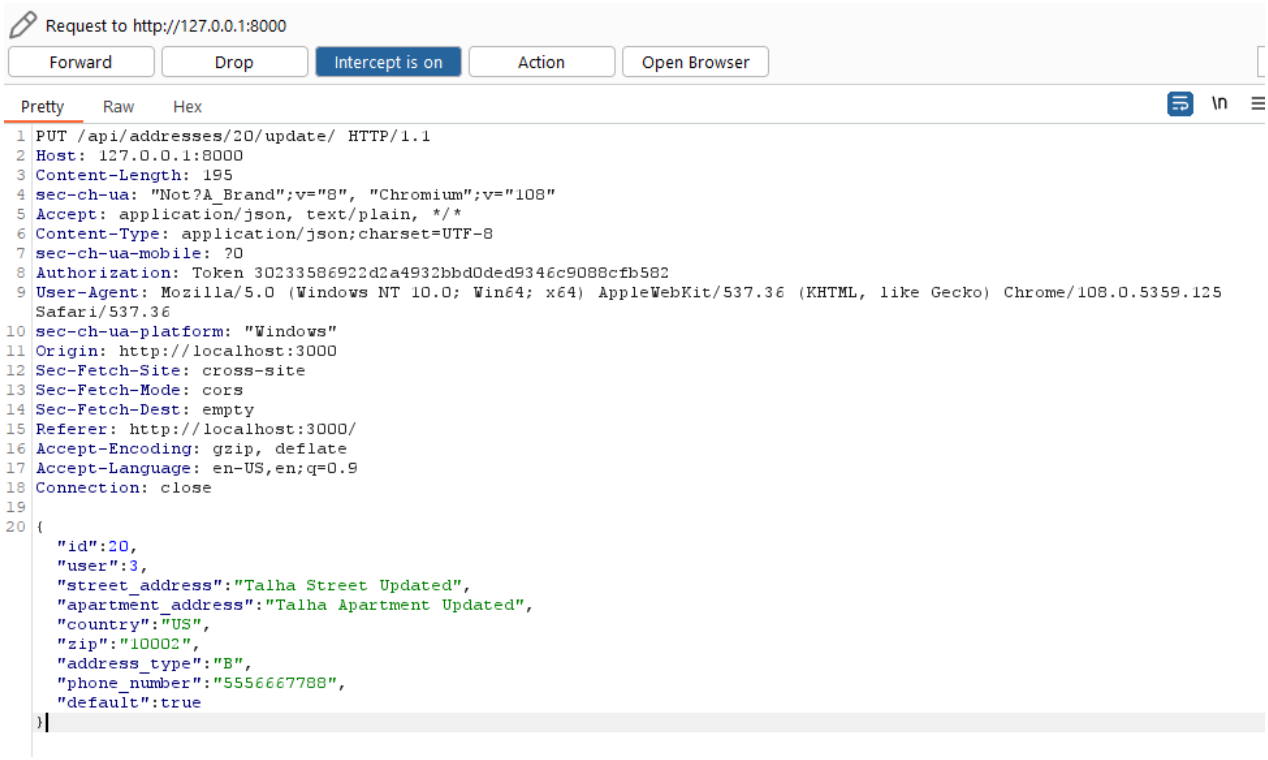

Figure 17: Intercepted request of updating billing address

In intercepted request, we will right-click and send it to the intruder. Since we don't want to update other users' address fields. we delete all of the parameters except ID and userID. We will relate the following ID with our userId (3).

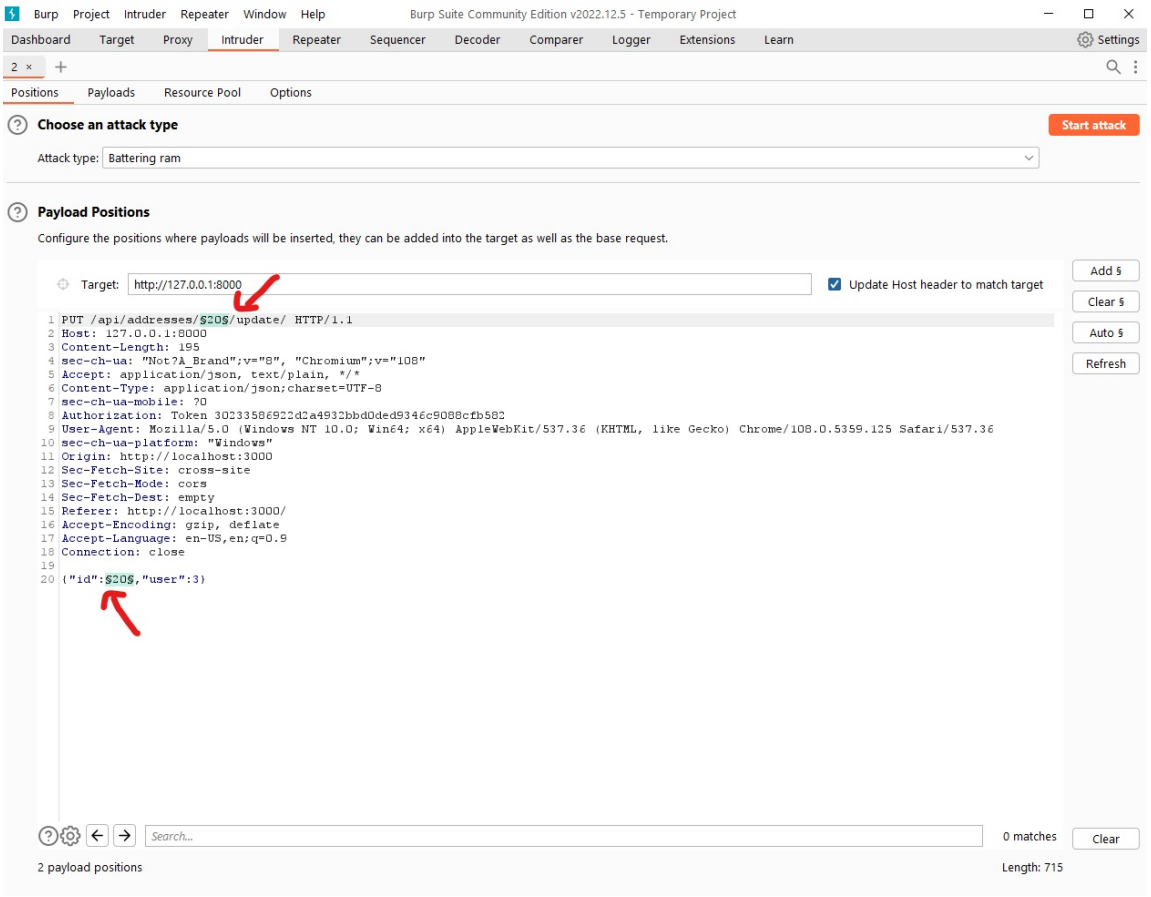

Figure 18: Intruder configurations

Since we are sending a PUT request to

`api/addresses/"addressID"/update`

,we need to set the AddressID part in the request URL and the ID part in the parameters section as the "payload position". Since we want to put the same payload in every payload position we use the Battering Ram attack type, you can see the attack type and payload positions in the above figure.

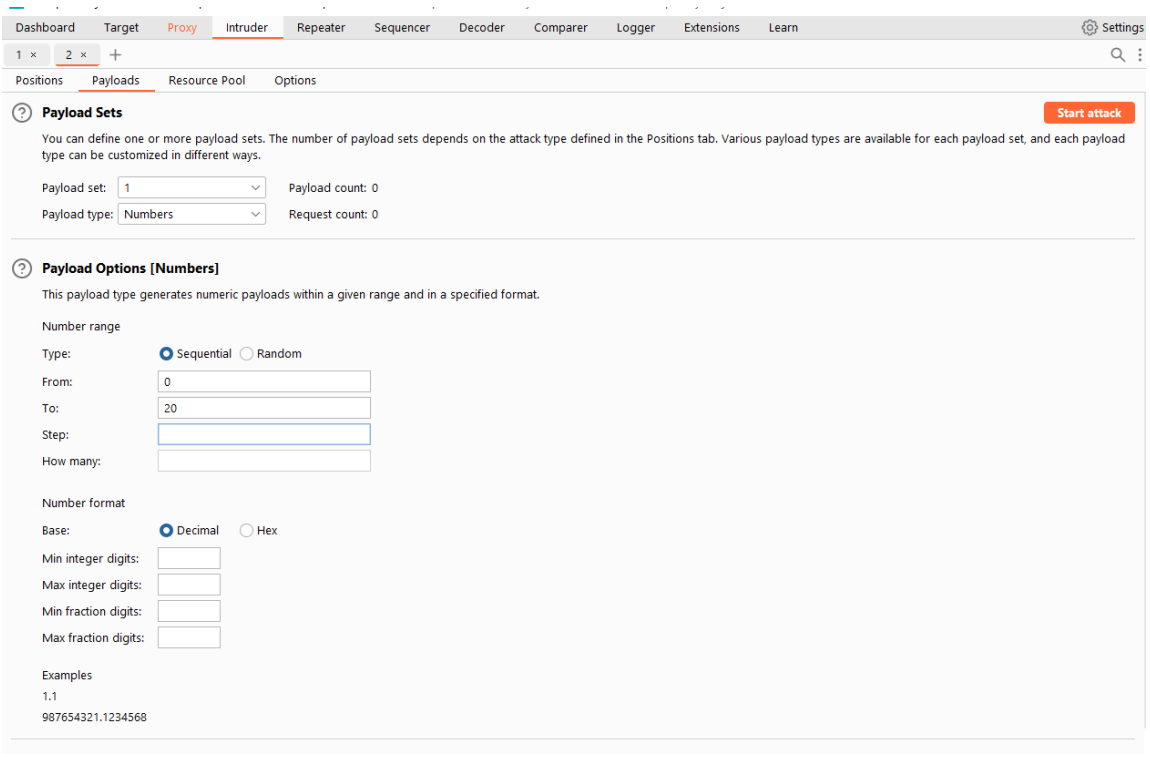

Figure 19: Payload configurations

From the Payloads Tab, we choose the Payload type as Numbers. Since we don't have too many addresses for this demo, we specify a Number range of 0 to 20 with 1 step each. But in a real life case, an attacker's range will include millions! Without further talk, let's start our attack.

As seen in the figures below, Burpsuite sent 20 different payloads and some of them resulted in a status of 200. When we visited the profile of the user "talha," we were able to see the addresses of all other users under "talha". What a shame!

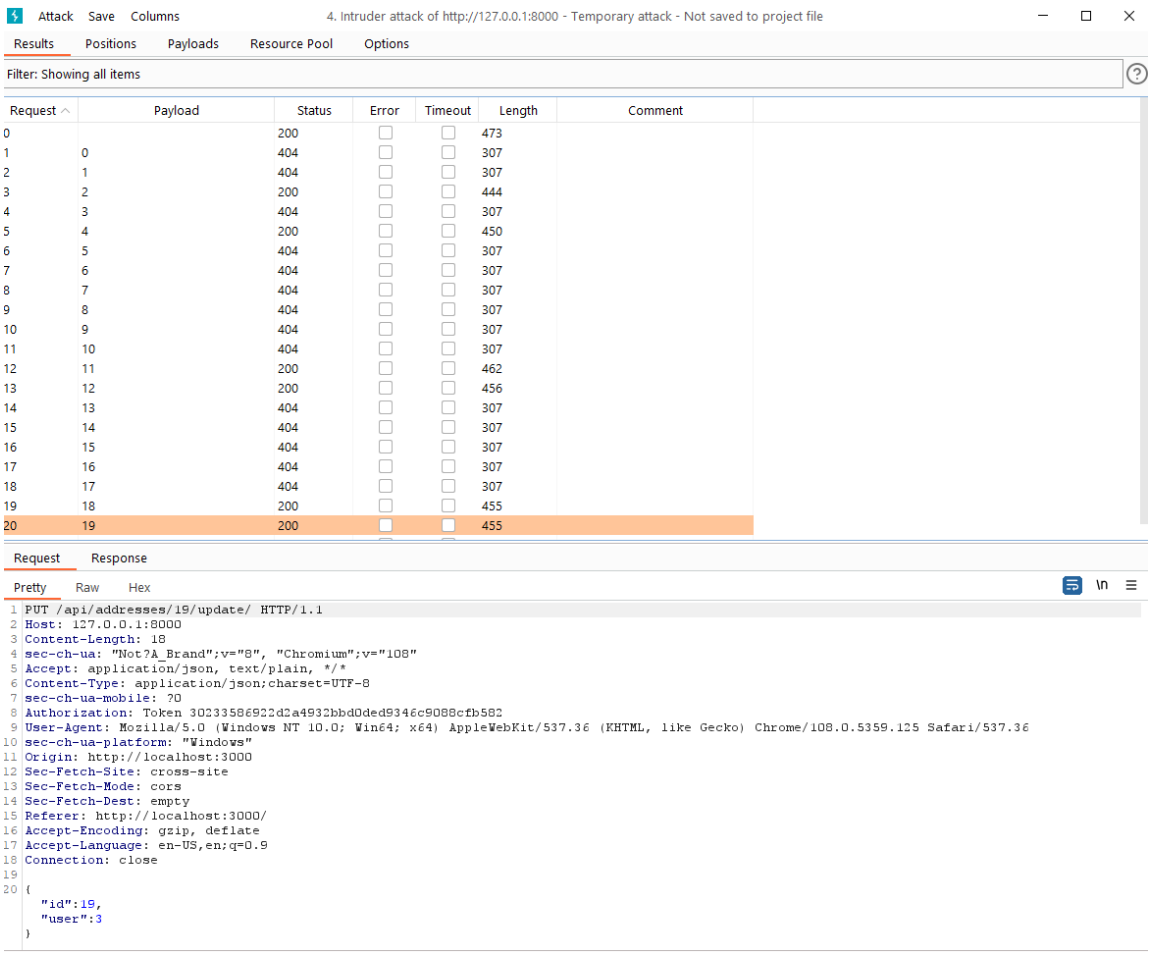

Figure 20: Payload results

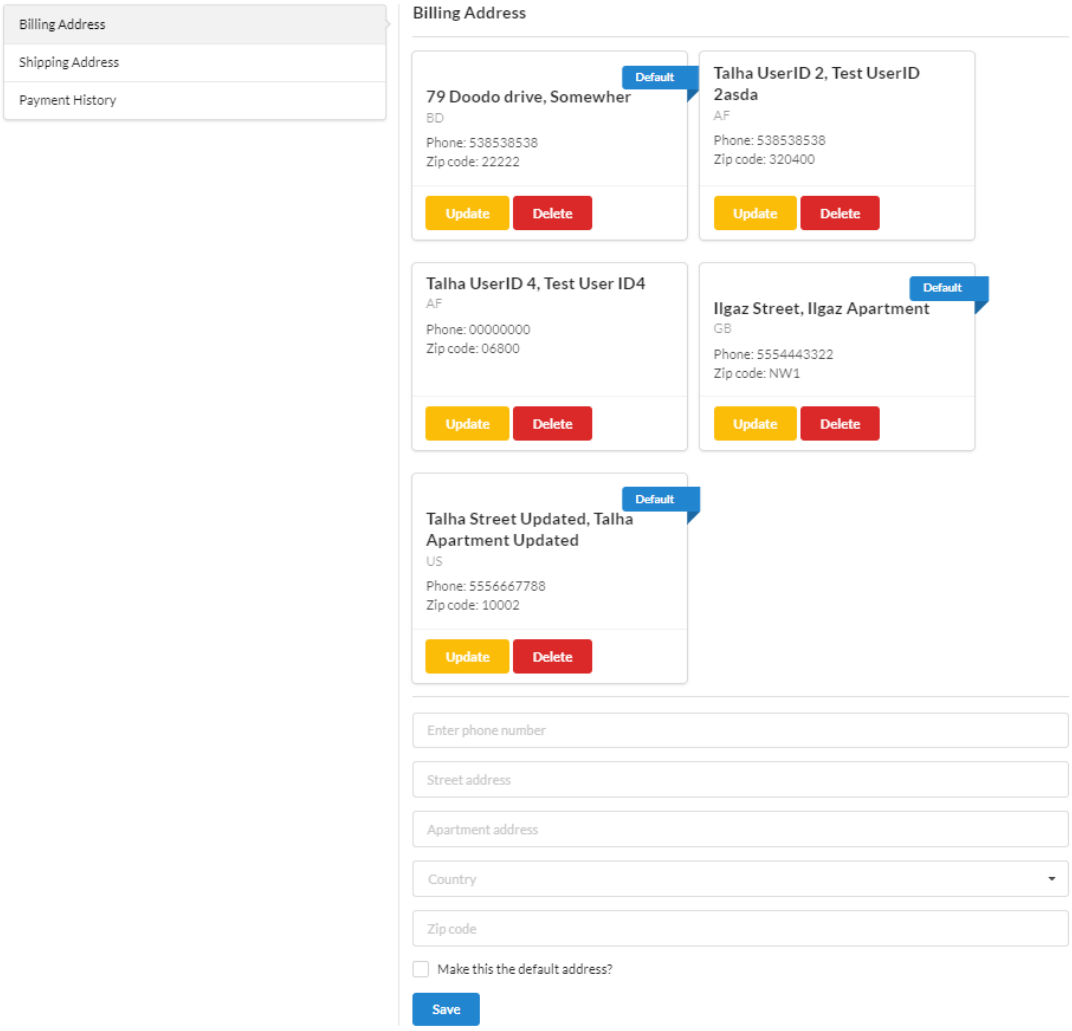

Figure 21: Profile of the user "talha"

4.2 Direct Reference to Static Files

We will continue with the example of direct reference to static files. As you may remember from the previous figures, there is a section in the payment history menu where the receipts of the payments can be downloaded. If we download receipt number 5, we get the receipt for our own order, as can be seen in the figures below .

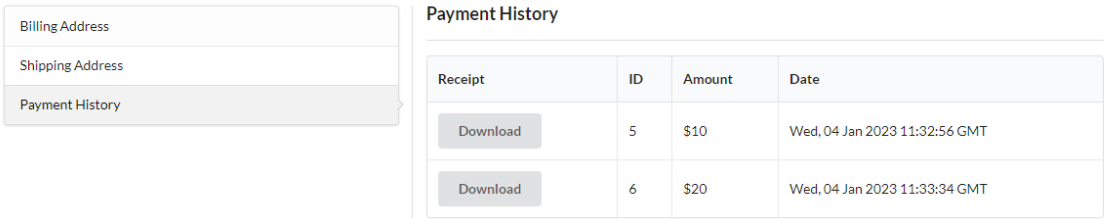

Figure 22: Payment history of "talha"

Figure 23: Receipt with ID=5

Let's activate the intercept again in Burpsuite and examine the requests and confirm whether we can access other users' receipts. We retrieve the relevant receipt using a get request send to

`/127.0.0.1:8000/api/download/5`

Here, we can also access other receipts by enumerating the receiptIDs, but to save time, we will modify the request with an ID (3) that we already know exists. When we look at the downloaded receipt, we see that we have obtained the receipt that belongs to the "ilgaz" user, from the user "talha". Dangerous! (Below Figure)

```
ReceiptID: 3
Name: Ilgaz
Surname: Senyuz
Email: ilgazsenyuz@hotmail.com
Phone: 5554443322
Address: Ilgaz Street, Ilgaz Apartment United Kingdom NW1
Amount Received: 1$
Items: Watch
Payment Method: Credit Card (5555 **** **** 4444) Exp: 07/28 CVV: 222
Date: Tue, 03 Jan 2023 19:42:27 GMT
```

Figure 24: Receipt with ID=3

I would like to remind you that we only reach this information by guessing the receipt ID. In this way, we can access other users' addresses, phone numbers and payment information along with random receipt IDs via Burpsuite.

In the previous section, we classified IDOR into two common types and in this section, we demonstrated how to exploit these types of IDOR vulnerabilities in our test environment.

5 Vulnerable Code Examples

The three classes and one function in the example are taken from our test environment. As shown in the below figures, although the user's authentication status is checked, it is not verified whether the user has access to the relevant address in address creation, updating, and deletion.

```
278
279  class AddressCreateView(CreateAPIView):
280      permission_classes = (IsAuthenticated, )
281      serializer_class = AddressSerializer
282      queryset = Address.objects.all()
283
284
285  class AddressUpdateView(UpdateAPIView):
286      permission_classes = (IsAuthenticated, )
287      serializer_class = AddressSerializer
288      queryset = Address.objects.all()
289
290
291  class AddressDeleteView(DestroyAPIView):
292      permission_classes = (IsAuthenticated, )
293      queryset = Address.objects.all()
294
```

Figure 25: Address views of Django

The authentication process is handled by Django as default. But unfortunately download_file function does not check whether the receipt being downloaded belongs to the user attempting to download it.

```
303
304  def download_file(request, order_id):
305      # use the order_id to determine the file path
306      file_path = f'{os.getcwd()}/receipts/{order_id}.txt'
307
308      # read the file and get its content
309      with open(file_path, 'rb') as f:
310          file_content = f.read()
311
312      # create the HttpResponse object with the file content
313      response = HttpResponse(file_content, content_type='text/plain')
314      response['Content-Disposition'] = f'attachment; filename={order_id}.txt'
315      return response
316
```

Figure 26: Download function

6 How to Prevent IDOR Attacks

6.1 Attempt to fix IDOR on the test environment

To fix this vulnerability, we can add a check to ensure that the user making the request is authorized to update the specific address object being modified. One way to do this is to override the perform update method in our view and add a check to ensure that the user making the request is the owner of the address being updated. This can be done by adding a line like the following at the beginning of the perform update method:

```
341  class AddressUpdateView(UpdateAPIView):
342      permission_classes = (IsAuthenticated, )
343      serializer_class = AddressSerializer
344      queryset = Address.objects.all()
345
346      def perform_update(self, serializer):
347          instance = serializer.save()
348          if self.request.user != instance.user:
349              raise PermissionDenied
```

Figure 27: Possible way to check for user

An alternative approach is to use Python's built-in "request.user" method. By filtering user objects with this method, we can retrieve the relevant data for that user from the database. As shown in the figures below, we have modified all functions that were vulnerable to IDOR.

```
278
279 class AddressCreateView(CreateAPIView):
280     permission_classes = (IsAuthenticated, )
281     serializer_class = AddressSerializer
282
283     def get_queryset(self):
284         if self.request.user == self.request.data['user']:
285             return Address.objects.all().filter(user=self.request.user)
286         else:
287             return Address.objects.none
288
289
290 class AddressUpdateView(UpdateAPIView):
291     permission_classes = (IsAuthenticated, )
292     serializer_class = AddressSerializer
293
294     def get_queryset(self):
295         if self.request.user == self.request.data['user']:
296             return Address.objects.all().filter(user=self.request.user)
297         else:
298             return Address.objects.none
299
300
301 class AddressDeleteView(DestroyAPIView):
302     permission_classes = (IsAuthenticated, )
303
304     def get_queryset(self):
305         if self.request.user == self.request.data['user']:
306             return Address.objects.all().filter(user=self.request.user)
307         else:
308             return Address.objects.none
309
```

Figure 28: Second way to check for user

To ensure that the download receipts function is secure, we can filter orders that belong to the requesting user and also have requested order ID. This will guarantee that the user can only access their own receipts.

```python
class download_file(APIView):

    # use the order_id to determine the file path

    def get(self, request, order_id):
        try:
            order = Order.objects.get(user=self.request.user, ordered=True, id=order_id)
            if len(order) > 0:
                file_path = f'{os.getcwd()}/receipts/{order_id}.txt'
                with open(file_path, 'rb') as f:
                    file_content = f.read()

                response = HttpResponse(
                    file_content, content_type='text/plain')
                response['Content-Disposition'] = f'attachment; filename={order_id}.txt'
                return response
            else:
                return Response({"message": "No active order or unauthorized"}, status=HTTP_400_BAD_REQUEST)
        except:
            return Response({"message": "No active order or unauthorized"}, status=HTTP_400_BAD_REQUEST)
```

Figure 29: Download receipt function

6.2 General Methods for preventing IDOR

Randomizing the numbers assigned to reference objects, rather than using a sequential numbering system, can help to reduce the risk of IDOR vulnerabilities. However, this is not a complete solution and other measures must be taken to fully protect against these types of attacks. Even using unique identifiers like UUIDs, which are designed to have a high level of randomness, may not be sufficient if a company's list of user IDs is leaked. In this case, attackers could potentially use the leaked list to execute IDOR attacks if the web application does not have proper access control measures in place.

To effectively prevent IDOR vulnerabilities, businesses should implement a robust approach that addresses all potential sources of risk. This might include using automated tools or manual testing methods to regularly scan and test for IDOR vulnerabilities, implementing strong access control measures to verify user permissions, and educating employees on how to identify and report potential security issues. In addition, it is important to ensure that any leaked lists of user IDs or other unique identifiers are promptly invalidated to prevent unauthorized access.

6.3 Indirect Reference Maps

Indirect reference maps involve replacing the direct reference to an object with an indirect reference that is much more difficult to guess. This can help prevent IDOR vulnerabilities by making it harder for

attackers to access sensitive data or manipulate objects by altering the user-supplied input. To implement an indirect reference map, the application should replace the direct reference to an object with an identifier such as a UUID (Universally Unique Identifier). Internally, the application should maintain a mapping between the UUIDs and the corresponding objects, so that it can translate the indirect reference back to the original form. It is important to note that while UUIDs and other methods of generating randomly assigned IDs can make it more difficult for attackers to guess the direct reference, they are not a foolproof solution. Indirect reference maps should be used in combination with other methods to effectively prevent IDOR vulnerabilities.

6.4 Fuzz Testing

Fuzz testing is a software testing technique that involves entering random or unexpected inputs into the application in an attempt to discover bugs and vulnerabilities. By testing the application's ability to handle these strange inputs, organizations can help detect IDOR vulnerabilities. Fuzz testing can be automated using tools such as the fuzz-lightyear framework developed by Yelp (Loo, 2020). It is important to note that while fuzz testing can help detect IDOR vulnerabilities, it does not prevent them. Organizations should use other methods in addition to fuzz testing to effectively prevent IDOR vulnerabilities.

6.5 Parameter Verification

Parameter verification involves checking user-supplied input to ensure it is of the correct length, type, and does not contain unacceptable characters. This can help prevent IDOR vulnerabilities by making it harder for attackers to access sensitive data or manipulate objects by altering the user-supplied input. Some checks that can be performed as part of parameter verification include:

- Verifying that a string is within the minimum and maximum length required
- Verifying that a string does not contain unacceptable characters
- Verifying that a numeric value is within the minimum and maximum boundaries
- Verifying that input is of the proper data type (e.g., strings, numbers, dates, etc.) It is important to note that while parameter verification can help prevent IDOR vulnerabilities, it is not a foolproof solution. Organizations should use other methods in addition to parameter verification to effectively prevent IDOR vulnerabilities.

6.6 Access Validation

The most effective way to prevent IDOR vulnerabilities is through access validation, which involves verifying that the user has the proper credentials to access the requested object or data. This can be done through server-side access control, which involves performing checks

on the server to ensure that the user is authorized to access the requested data. It is important to perform access validation at multiple levels, including the data or object level, to ensure that there are no holes in the process. By performing access validation, organizations can effectively prevent IDOR vulnerabilities and protect sensitive data from unauthorized access.

7 Attempt to exploit more secure code

Let's revisit the steps we took previously to exploit the vulnerability. The following figures illustrate the actions taken in the correct order.

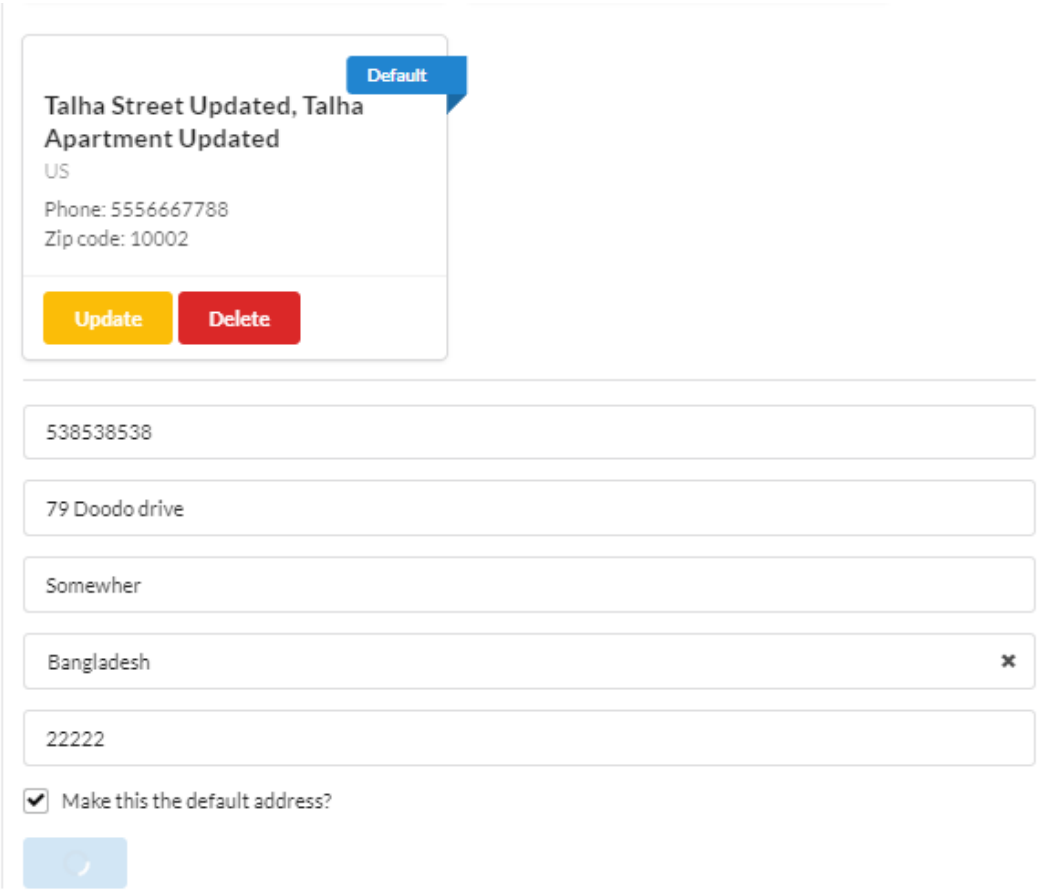

Figure 30: Updating address of User

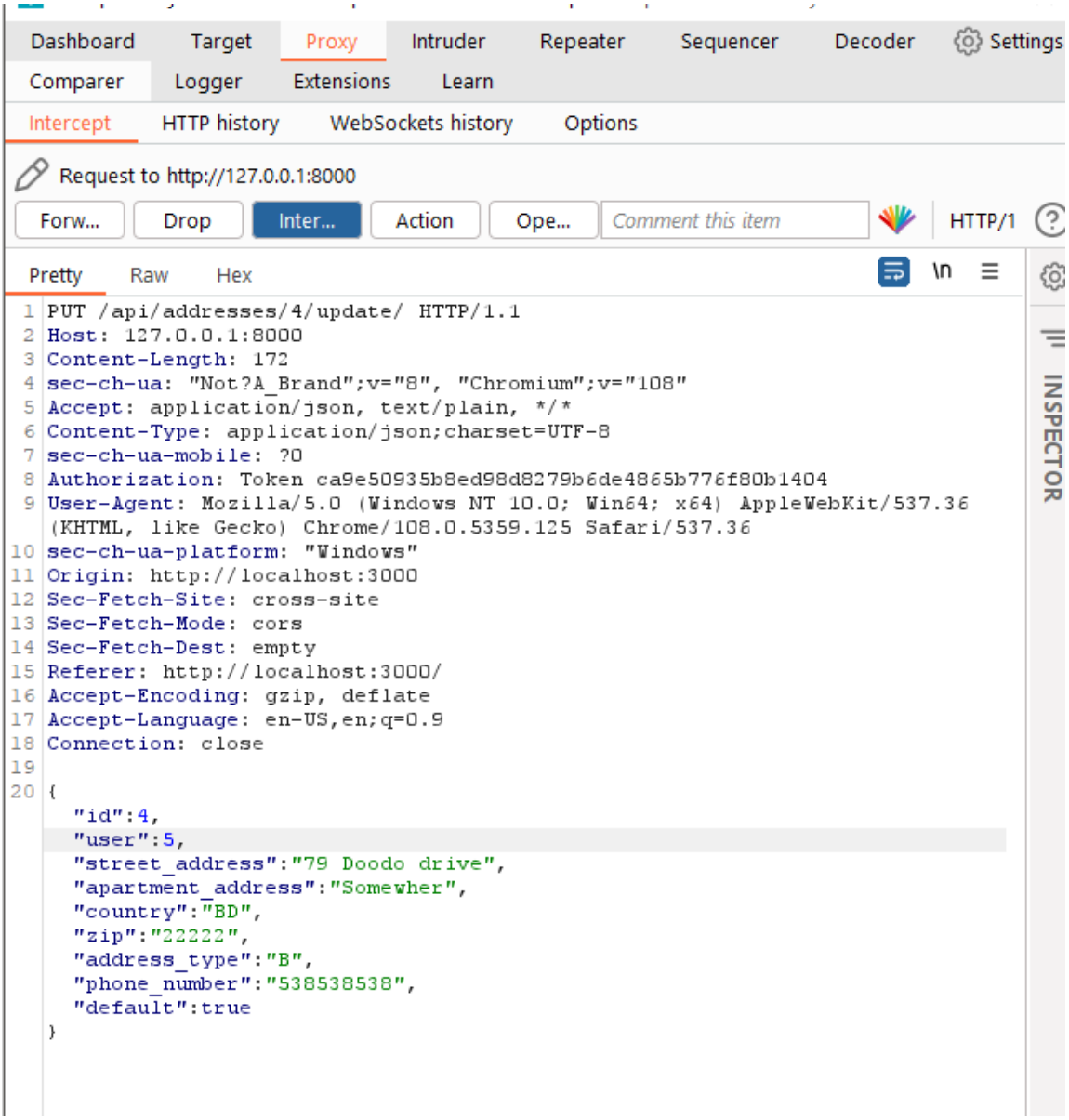

Figure 31: Intercepting request and sending it to the intruder.

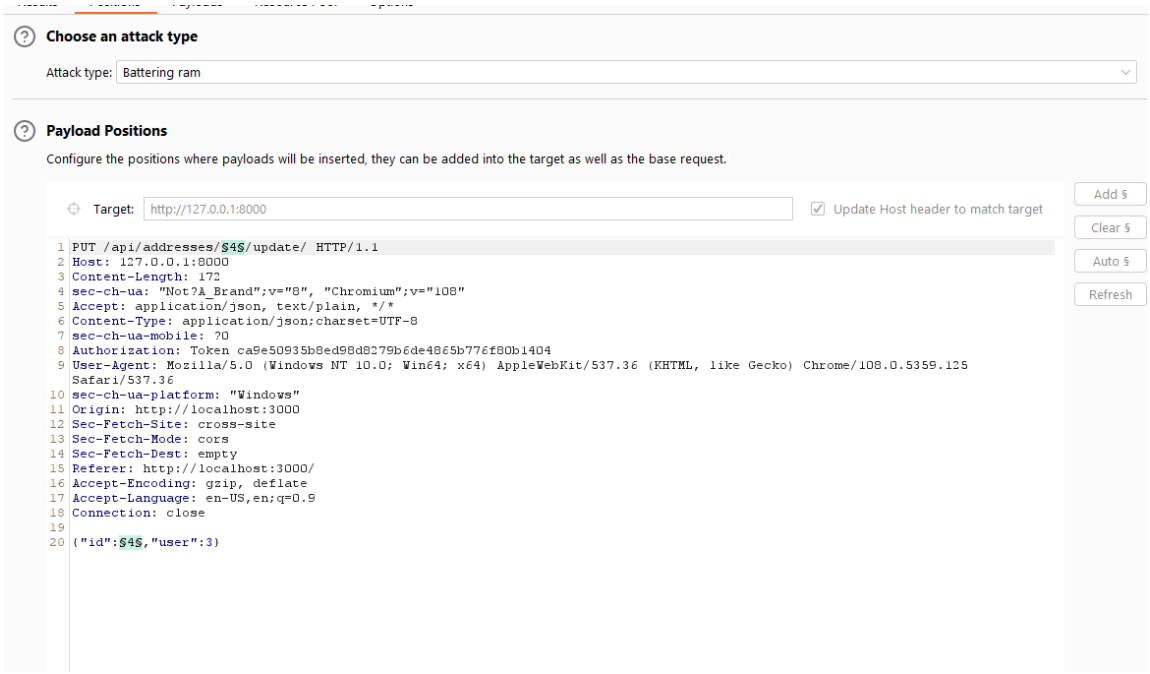

Figure 32: Clearing unrelated fields and changing user ID, configuring payloads as before

As can be seen in the figure below, all of the requests we sent resulted in 400 and we got the unauthorized message as a response.

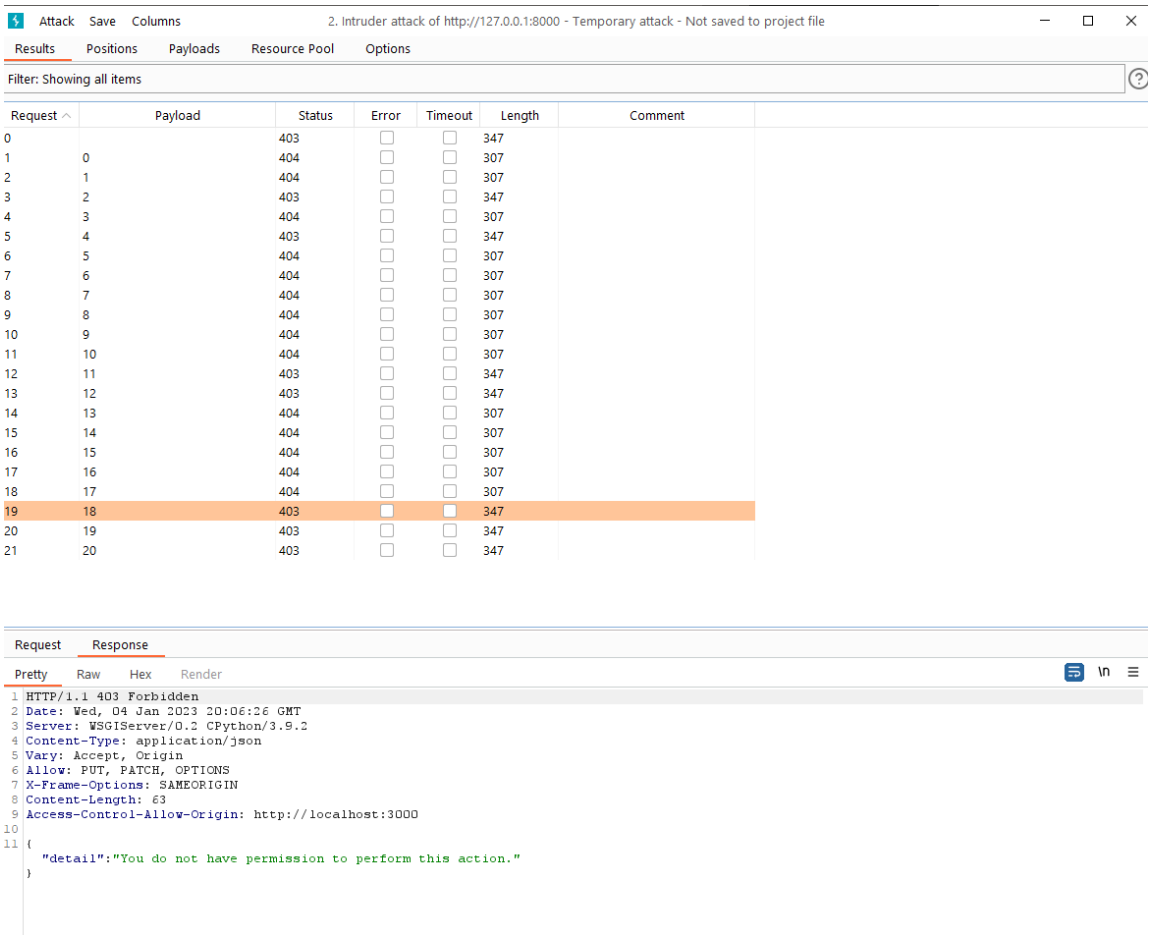

Figure 33: Starting attack

We also got 400 and we got the unauthorized message when trying to manipulate BurpSuite request.

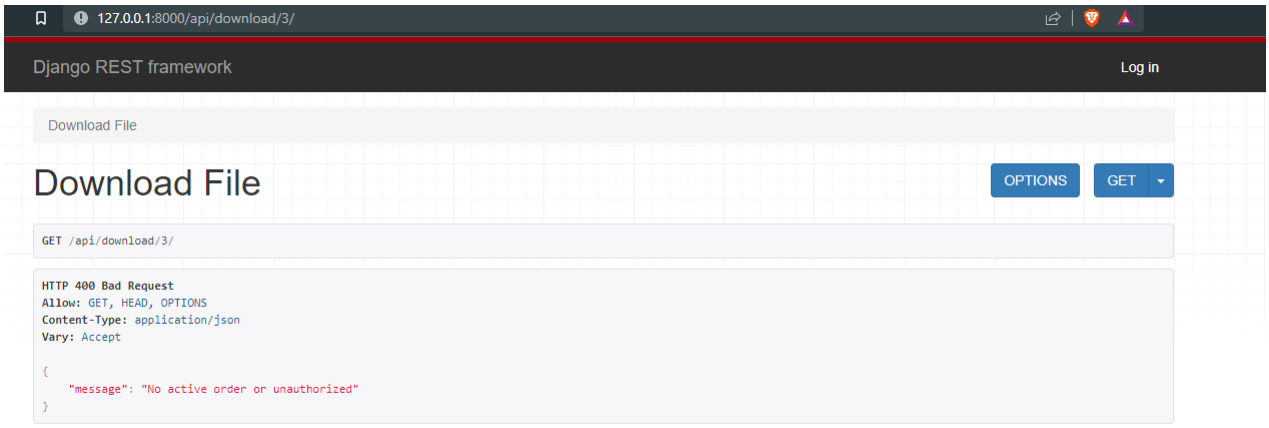

Figure 34: Attempting to download other receipts.

8 References

- Github E-Commerce Repository – `https://github.com/justdjango/django-ecommerce`
- `https://shellmates.medium.com/insecure-direct-object-references-i`
- `https://spanning.com/blog/insecure-direct-object-reference-web-ba`
- `https://portswigger.net/web-security/access-control/idor`
- Mehmet Ince – `https://www.youtube.com/watch?v=TsJ2XPuGe1k&t=5156s`
- `https://www.eccouncil.org/cybersecurity-exchange/web-application-idor-vulnerability-detection-prevention/`

- https://www.invicti.com/blog/web-security/insecure-direct-object-
- https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html