INTRODUCTION

The aim of this study is improve knowledge about Business Logic Vulnerabilities.

DETAILS

**What are business logic vulnerabilities?**

Business logic vulnerabilities are flaws in the design and implementation of an application that may allow attacker to execute unintended behavior. This generally allow attacker to use legitimate function of an application for his/her malicious goal. These kind of flaws generally result of failing to anticipate unusual application states that may occur and failing to handle them safely (In Sc; unexpected input causes info disclosure about libraries that has been used.)

This kind of flaws generally invisible to normal people but attacker can exploit these behavioral quirks by interacting application in a way that developers never expected.

\*Business logic’s purpose is dictate how application whill react when a give scenario occurs. This also includes preventing users from doing malicious things that will have negative impact on the business or things that just don’t make sense.

Flaws in the logic allow attacker to circumvent these rules. For example, they might be able to complete transaction without going through the intended purchase workflow. In other cases, broken or non-existent validation of user-supplied data might alllow user to make arbitrary changes in critical transaction values or they can submit nonsencial input (In Juice shop price of a product can be changed as negative like -5 $.) By passing an unexpected values into server-side logic, an attacker can potentially enforce the application to do something that isn’t supposed to do.

These kind of vulnerabilities often unique to the application and it’s specific functionality. For this reason this kind of vulnerabilities are great target for bug bounty hunters and manual testers because they are so difficult to detect by automated softwares.

**How do business logic vulnerabilities arise?**

Business logic flaws often arise because the design and development teams make flawed assumptions about how user will interact with the application. These kind of assumptions can lead to insufficient validation of user input. For example if developer assume that users will pass data exclusively via web browser, the application may rely on entirely client-side control and this a big mistake. An attacker can easily bypass client-side controls by using proxies like Burp Suite. This means that attacker deviates from expected user behaviour. The application fails to take appropriate steps to prevent this and fails to handle the situation safely.

Logic flaws are specifically common in complicated systems that even development team fully understand the logic of the application. To avoid these flaws, developers must understand the application as a whole. This includes, being aware of how different functions can be combined in unexpected ways. Developer that works on one component could make flawed assumption about how another component of an application works and as a result this can introduce serious logic flaws. If developers did not document the assumptions they are made, logic flaws will occur commonly.

**What is the impact of business logic vulnerabilities?**

Impact of the business flaws can differ. Sometimes fairly trivial, sometimes huge consequences can occur. Unexpected behavior can potentially lead to high-severity attack if attacker able to manipulate the application in the right way. Because of that reason, even if you cant work out exploit a business logic vuln. you must fix it as soon as possible because there will always be a risk that someone else will be able to.

Generally, impact of logic flaws depend on what functionality of application is related to. If flaw is in the authentication mechanism, this could have serious impact on overall securty. Attacker could potentially exploit this for privilege escalation, or to bypass authentication entirely, gaining access to sensitive data and functionality. This also exposes an increased attack surface for other exploit. (Generally business logic flaws is a starting point of an attack vektor. Attacker can create attack vector according to the logic flaw in the app).

Even if the logic flaw may not allow an attacker to benefit directly, they could still allow a malicious party to damage the business in some way. (F.E if logic in the application does not check the third party libraries that have been using, A supply chain attack may occur.)

**Examples of business logic vulnerabilities**

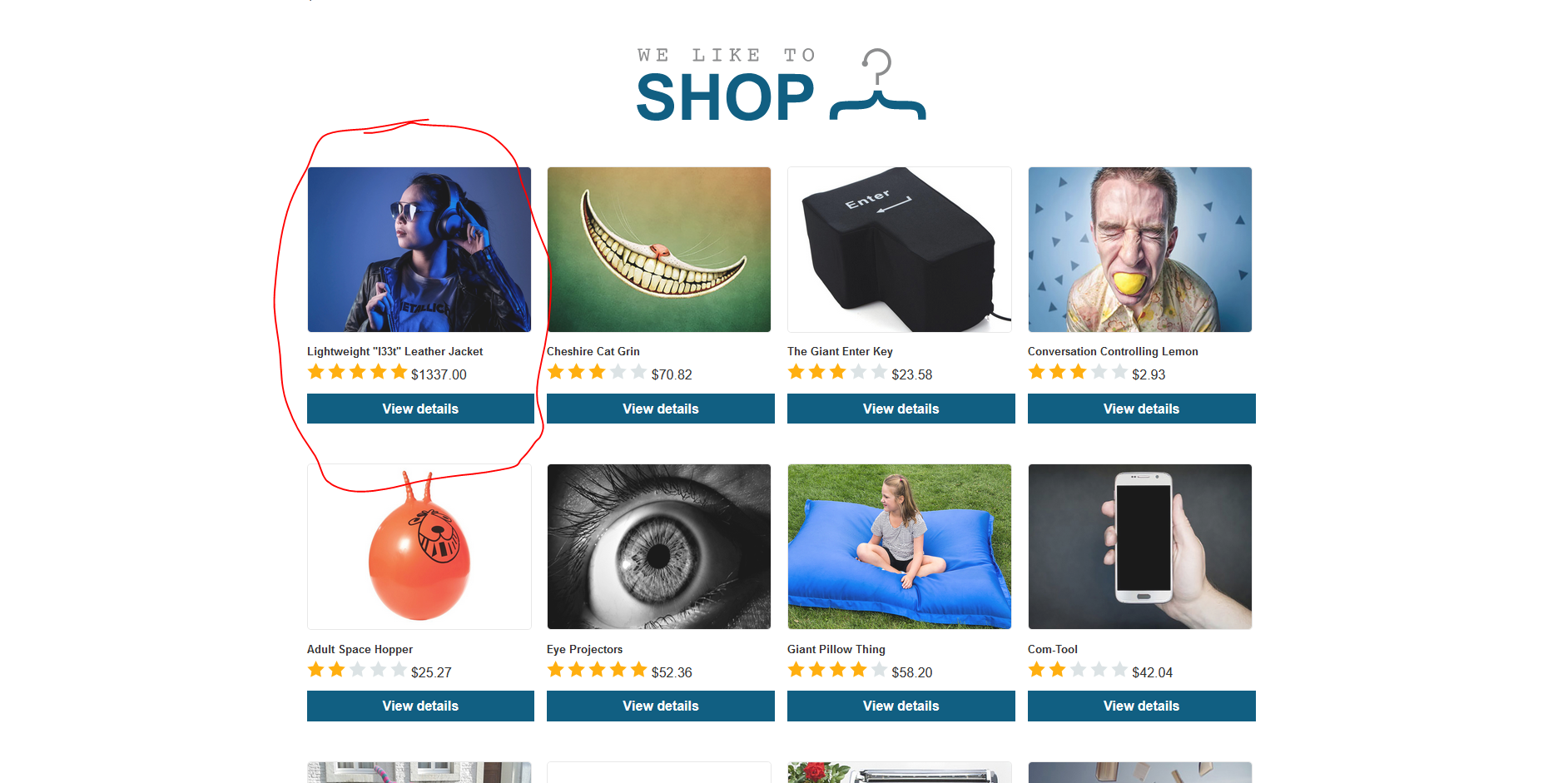
**1-) Excessive trust in client-side controls**

This assumption is flawed because developers think even only client-side protection will prevent malicious input to the application. But attacker can use proxy tools such as burpsuite to tamper with the data after it has been sent by the browser but before it is passed into the server-side logic. This effectively renders the client-side controls useless.

Accepting data at face value, without performing proper integrity checks and server-side validation, can allow an attacker to do all kinds of damage with relatively minimal effort. Exactly what they are able to achieve is dependent on the functionality and what it is doing with the controllable data. In the right context, this kind of flaw can have devastating consequences for both business-related functionality and the security of the website itself.

**LAB 1 : Excessive trust in client-side controls**

In this lab we need to buy 1337$ jacket. But we have only 100 dollars. So lets downgrade the price a bit in burp because they are not checking the price of the products in the serverside.



Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Chart

Description automatically generated with low confidence

**LAB 2: 2FA broken logic**

This application’s 2-FA is flawed. Access Carlos’s account page.

1-) First we tried to enter the application with wiener’s credentials but in cookies there were field such as verify=wiener, so I changed it as carlos and 2FA provided 2FA code for carlos’s account

Graphical user interface, text, application

Description automatically generated

2-) So even I entered application with wiener’s account credentials, by changing verify value from cookies I managed to enter Carlos’s account.

Graphical user interface, application

Description automatically generated

**2-) Failing to handle unconventional input**

One aim of the app logic is restrict user input to values that suitable for the business rules. F.E app is designed to accept arbitrary values of a certain data type but logic determines wheter or not this value is acceptable.

For example, in a transaction site, nobody can change price values with the negative ones. Theoretically integer values can be also negative but this values does not suitable for business logics. So developer must set rules for the unexpected inputs that can come from users to prevent malicious goals. If they dont set propriate business logic rules for a transaction site, user may make a purchase with -1000$ than this means app bank account will lose 1000 $.

When auditing an application we can use Burpsuite Repeater and Proxy components to try unexpected values as inputs in the application. Try input in ranges that legitimate users are unlikely to ever enter. This includes exceptionally high or exceptionally low numeric inputs and abnormally long strings for text-based fields. We can also try unexpected data types. By observing the application's response, you should try and answer the following questions:

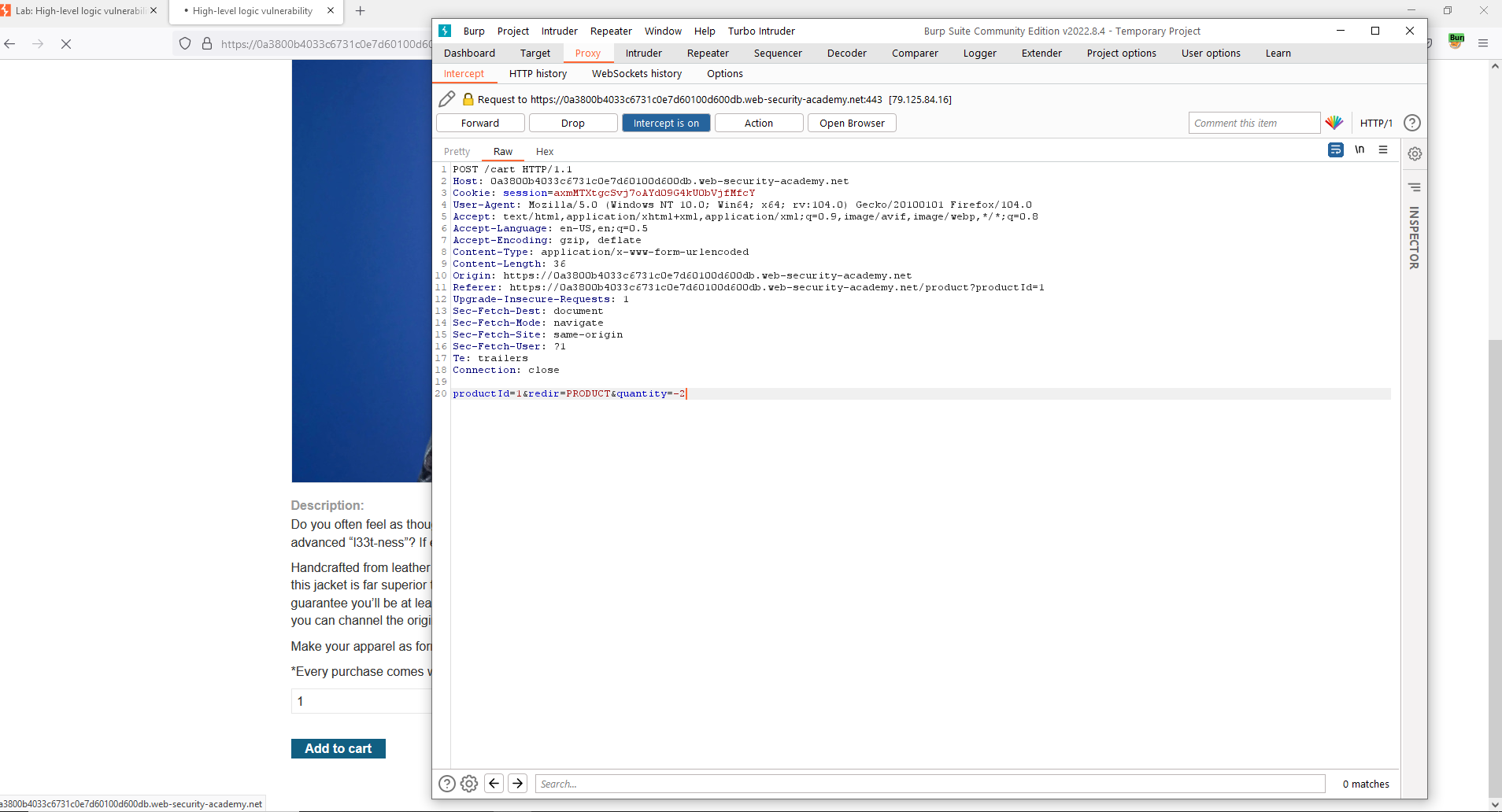
* Are there any limits that are imposed on the data?
* What happens when you reach those limits?
* Is any transformation or normalization being performed on your input?

\*\*\*This may expose weak input validation that allows you to manipulate the application in unusual ways. Keep in mind that if you find one form on the target website that fails to safely handle unconventional input, it's likely that other forms will have the same issues.

**LAB 3: High-level logic vulnerability**

This lab doesn't adequately validate user input. You can exploit a logic flaw in its purchasing workflow to buy items for an unintended price. To solve the lab, buy a "Lightweight l33t leather jacket".

1-) Lets make the quantity of the jacket -1. So we can easily buy it.



2-) Application does not allow negative price of product. We have $100 dollar in the app. So we can add +1 jacket and -24 Hydrated Cracker (1260$or smth)

Graphical user interface, text, application

Description automatically generated

3-) This website is so cheap.

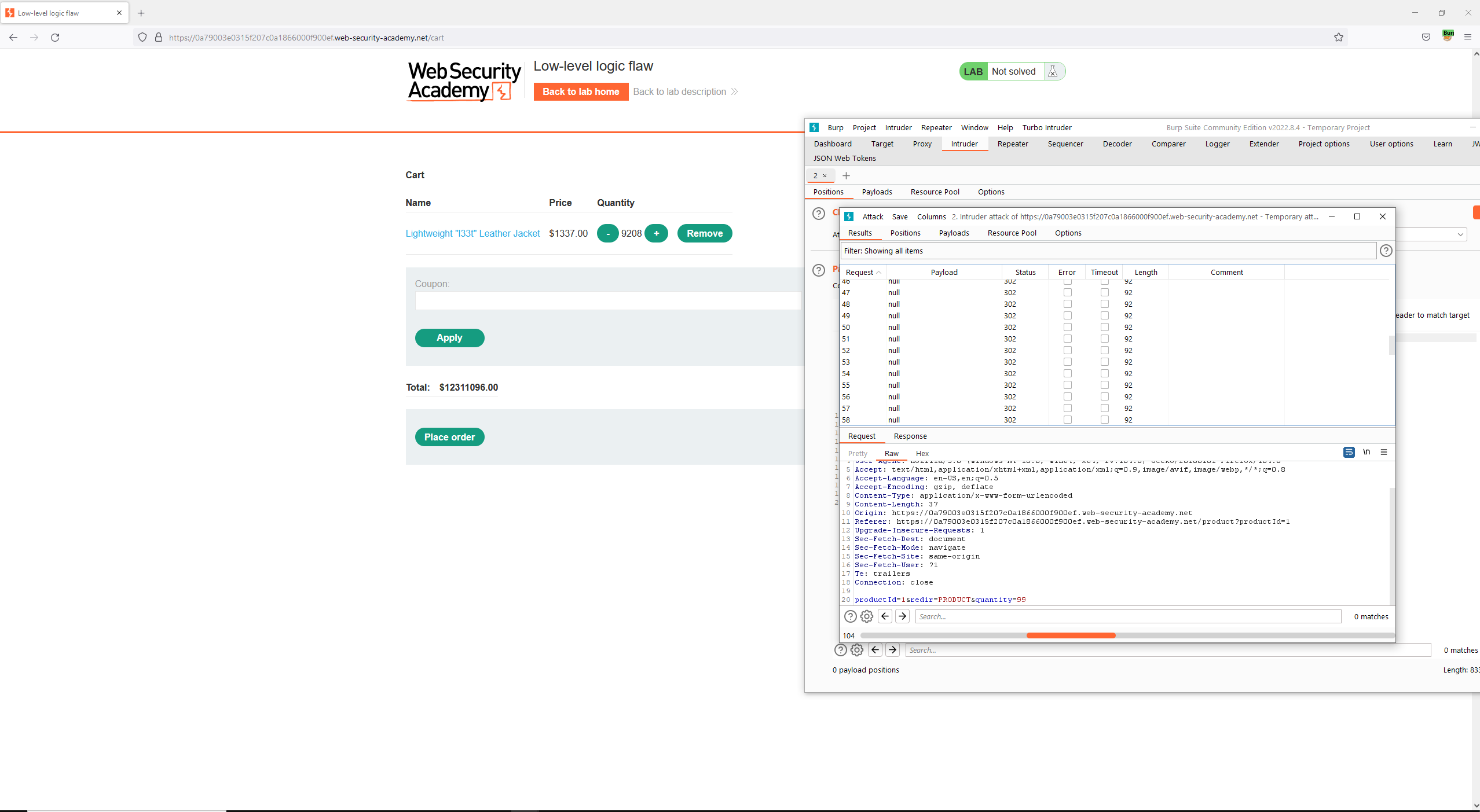
Graphical user interface, application

Description automatically generated

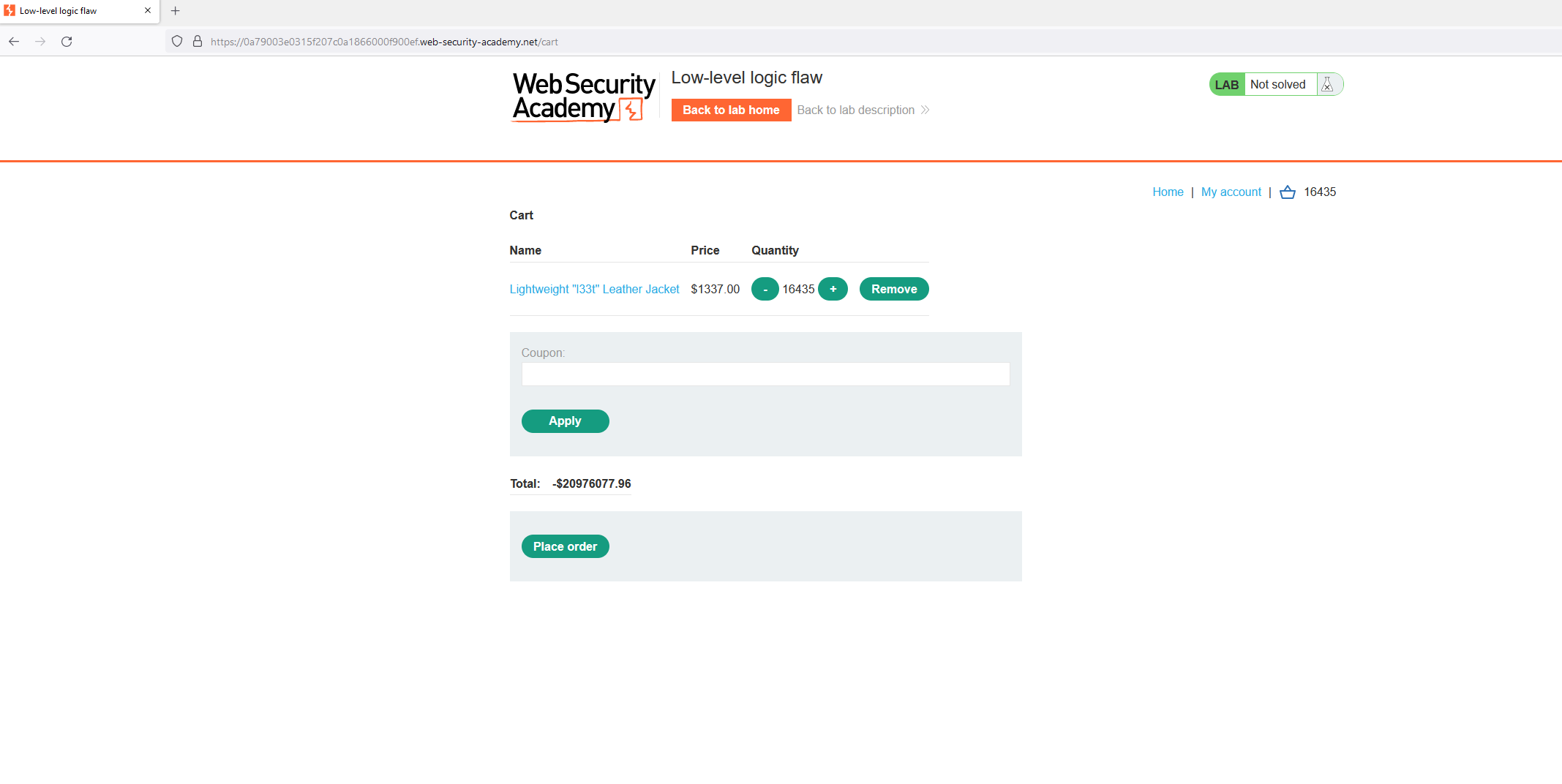
**LAB 4: Low-level logic flaw**

This lab doesn't adequately validate user input. You can exploit a logic flaw in its purchasing workflow to buy items for an unintended price. To solve the lab, buy a "Lightweight l33t leather jacket".

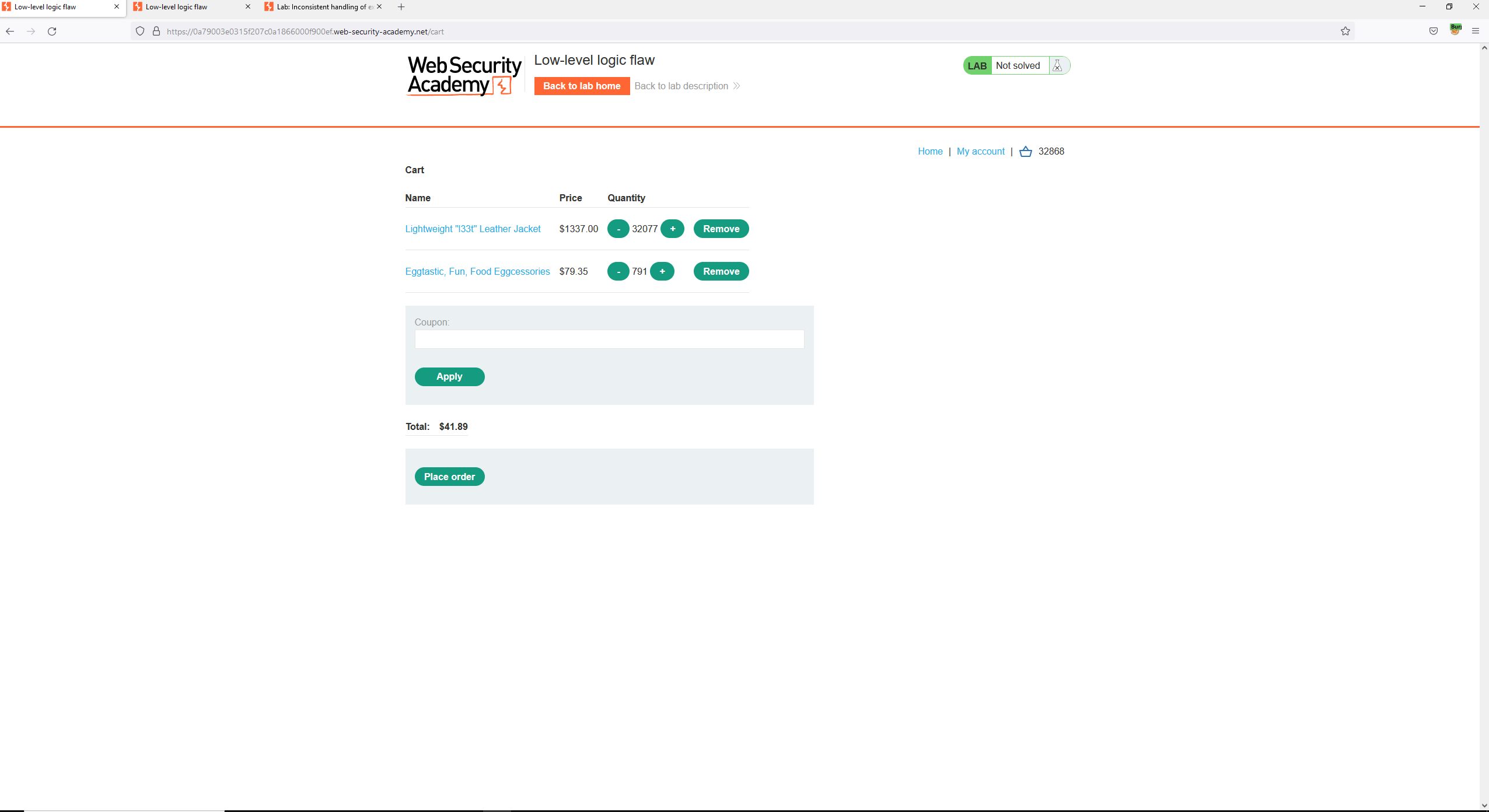
1-) We will add so many jacket to reach and pass the limit of the total value variable.(hypothesis is that when we pass the limit on total value integer, integer will go from the positive numbers to negaitive numbers after limit)



2-) Hypothesis seems true. No control about negative values after passing the limit.



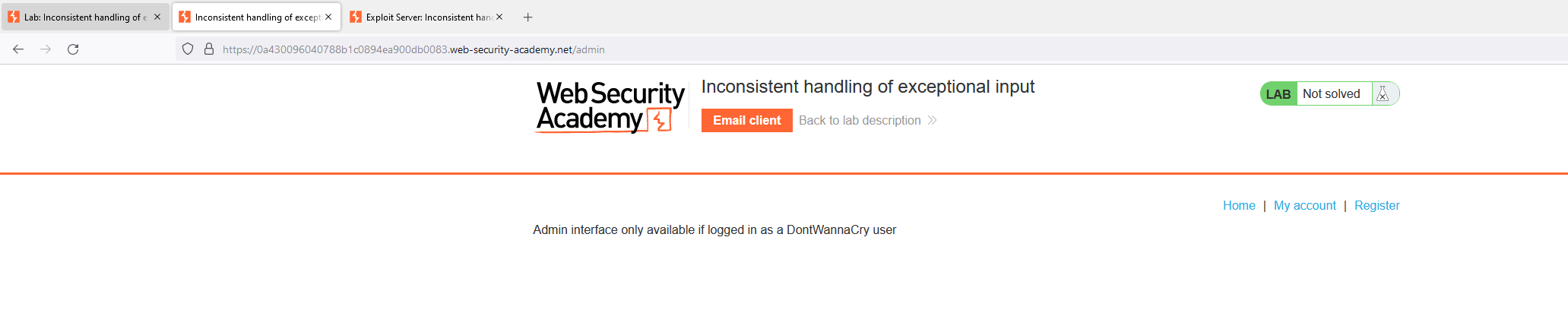
3-) And done. Lab Completed.



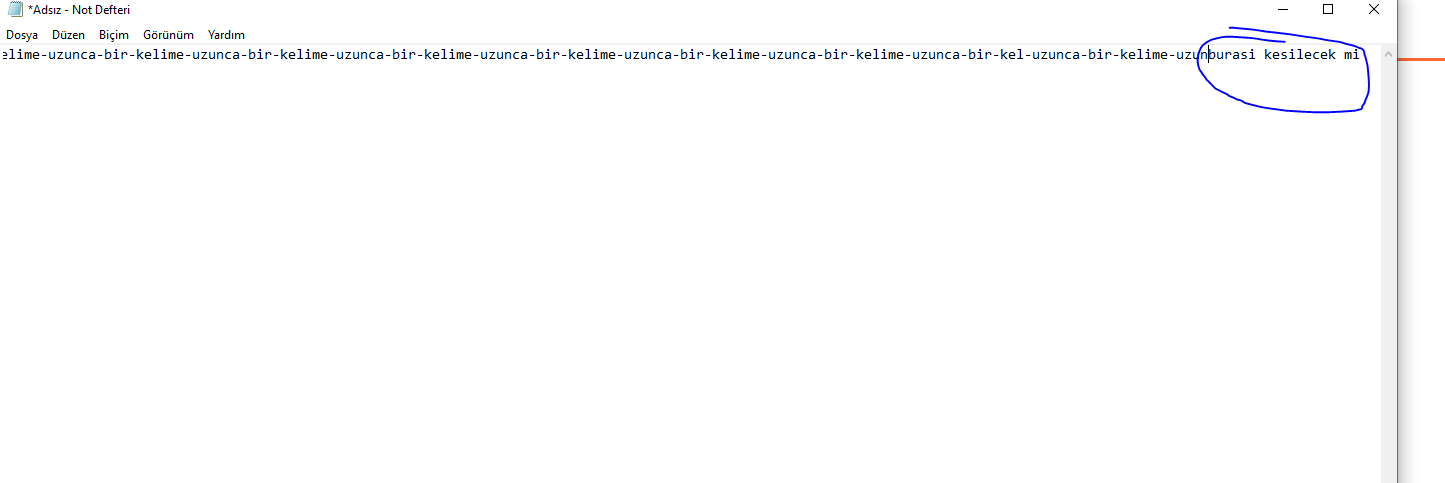
**LAB 5: Inconsistent handling of exceptional input**

This lab doesn't adequately validate user input. You can exploit a logic flaw in its account registration process to gain access to administrative functionality. To solve the lab, access the admin panel and delete Carlos.

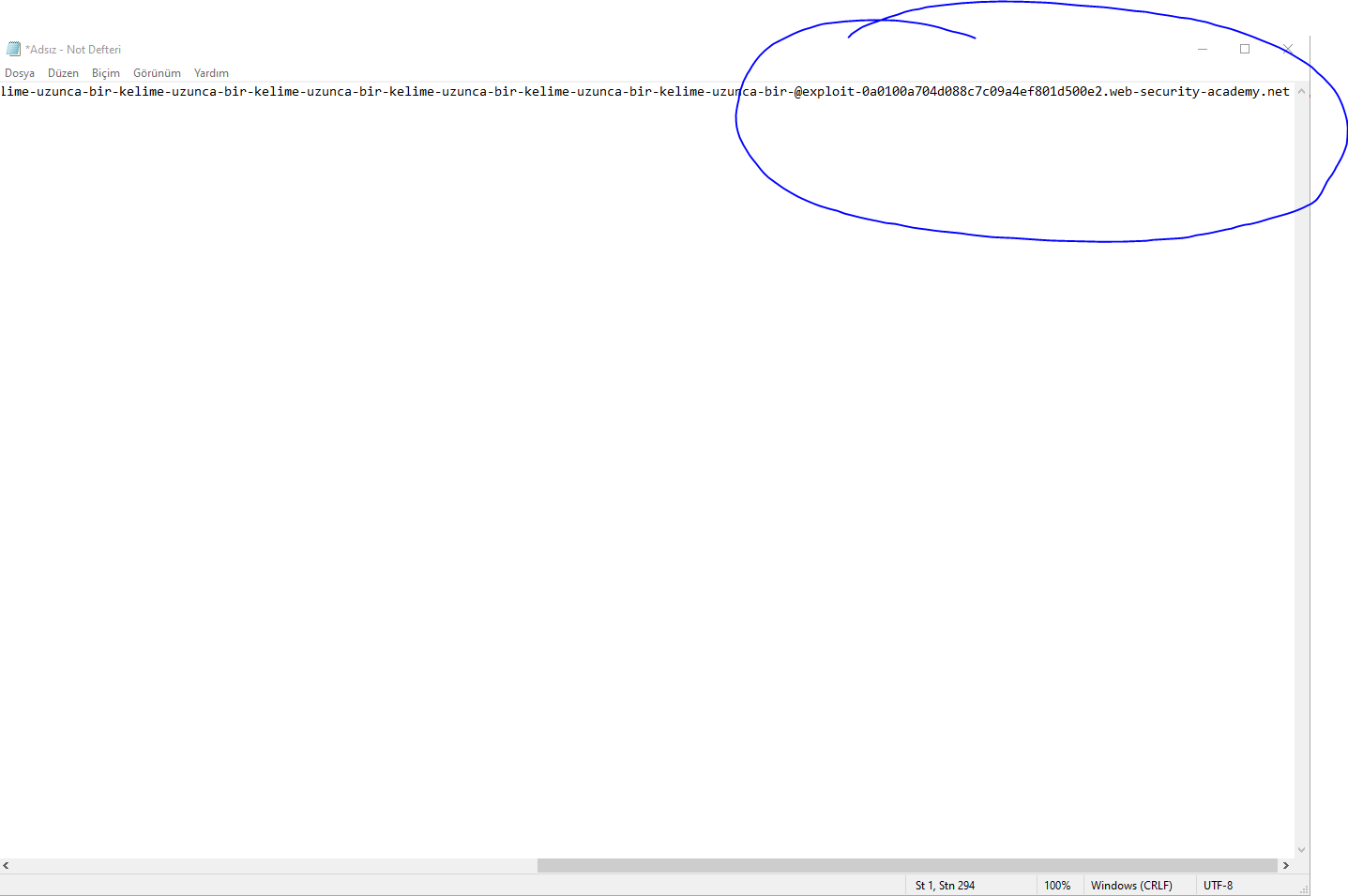
1-) First we are searching for admin subdomain to delete user Carlos. (we can use dirb or we can easily write /admin or /administrator on url)



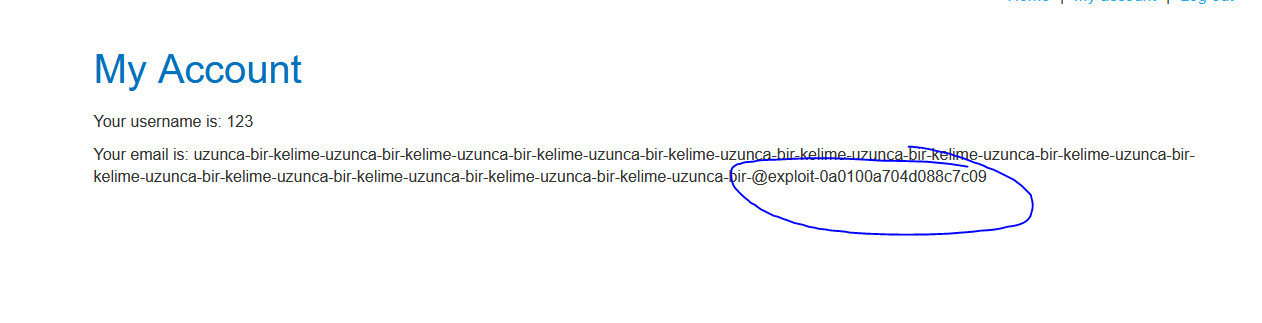
2-) As we can see admin interface only available to DontWannaCry user. So we need to become that user. So on register page, I try the string limits of the every input fields to see what happens. (Invalid username format error comes from application)



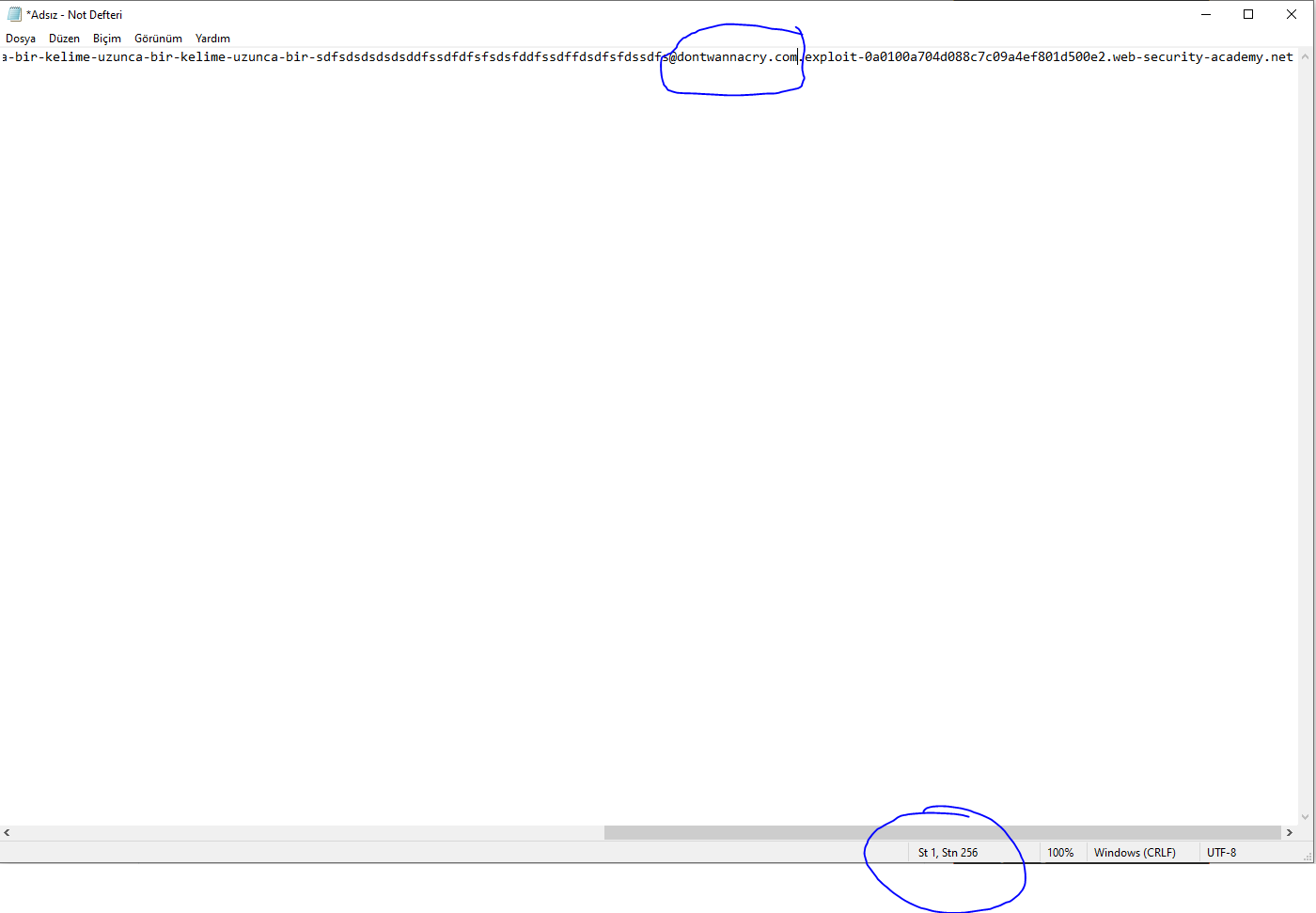
3-) Then try it on the mail part



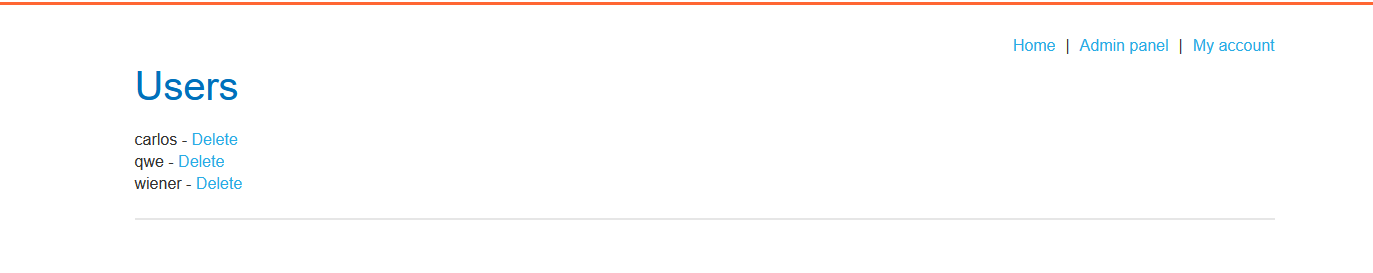
4-) As we can see in the mail part after 256 char it cuts the string

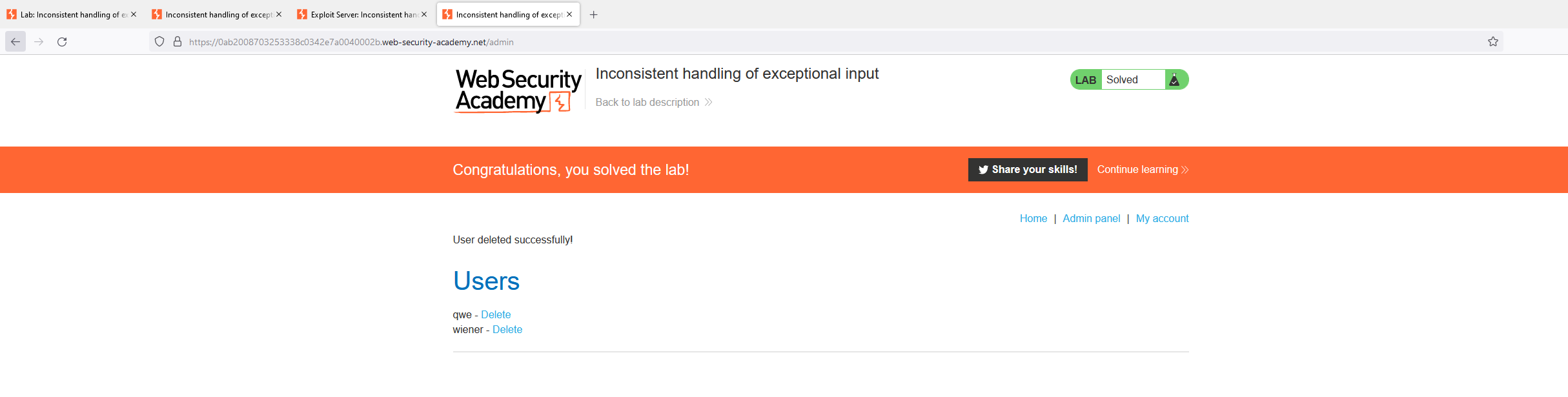


5-) Then I will try to create a user with @dontwannacry.com (256. char is m of dontwannacry.com After this part all chars will be deleted. We need to place “.” after the .com part of dontwannacry to be able to work email mechanism)

6-) After account verification on email client we can see our email :







**3-) Making flawed assumptions about user behavior**

One of the most common reason of logic vulnerabilities is flawed assumptions about user behaviours. This can lead wide range of issues where developers have not considered potential dangerous scenarios that violate these assumptions. Here is a few example of these dangerous assumptions.

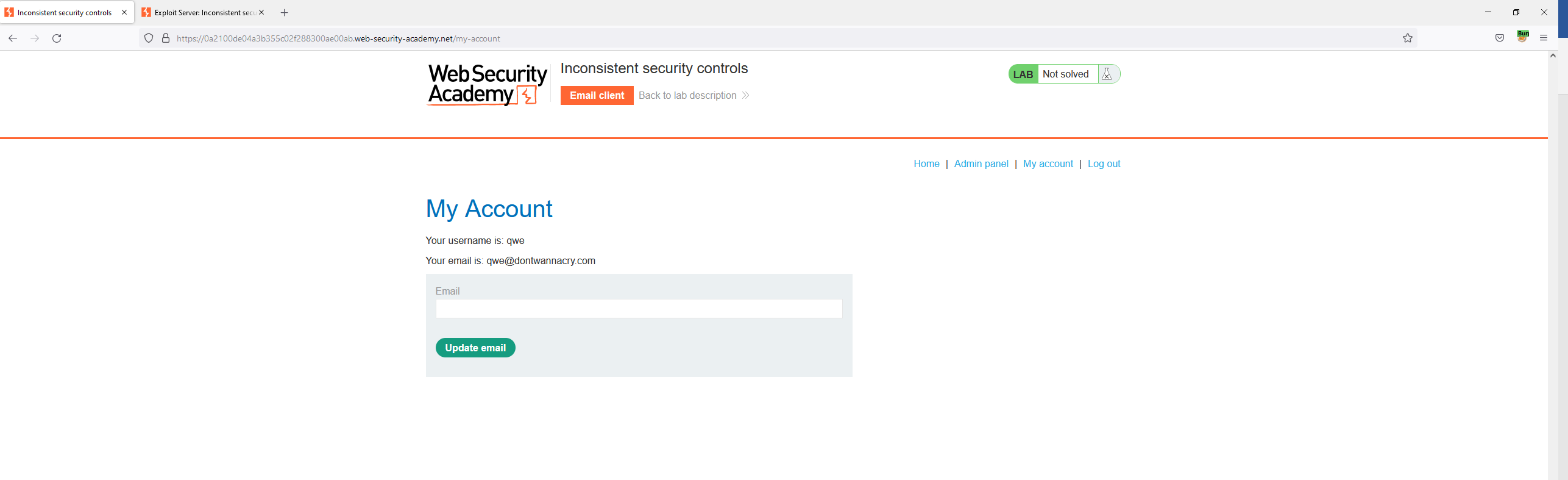
**Trusted users won't always remain trustworthy:**

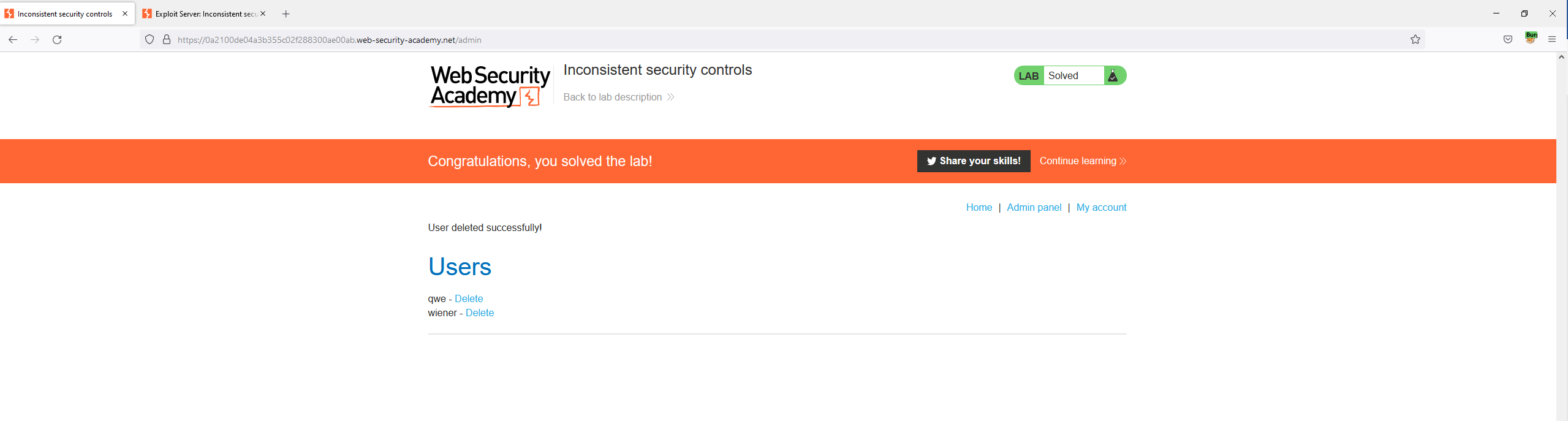
Sometimes developers initally use robust measures to show the application like a secure app, but after the initial part, they did not check the input indefinitely. This can result lax enforcement of the same controls from that point on. If business rules and security measures are not applied consistently throughout the app, this can lead to potentially dangerous loopholes that may be exploited by an attacker.

**LAB 6: Inconsistent security controls:**

This lab's flawed logic allows arbitrary users to access administrative functionality that should only be available to company employees. To solve the lab, access the admin panel and delete Carlos.

1-) Admin panel only available to dontwannacry mailed users. So I updated my E-mail as DontWannaCry user and admin panel was available





**Users won't always supply mandatory input:**

One misconception is that users will always supply values for mandatory input fields. Browser may prevent users from submittin a form without required input but attacker can delete that input field when using Proxy tools like Burp. The also can remove parameters entirely.

This is a specific issue in cases where multiple functions are implemented within the same server-side script. In this case absence or presence of spesific parameter may determine which code is executed. Removing parameter values can allow attacker to Access code paths that are supposed to be out of reach.

When we trying to find logic flaws, we should try removing each parameter in turn and observing what effect this has on the response. Follow these steps to find logic flaws:

* Only remove 1 parameter at a time to see every parameter’s function when we delete it
* **\*\*\*Try deleting the name of the parameter as well as the value. The server will typically handle both cases differently(CSRF cookie parameter value ile birlikte name’i de silme mantığı gibi)**
* Follow multi-stage processes through to completion. Sometimes tampering with a parameter in one step will have effect on another step.

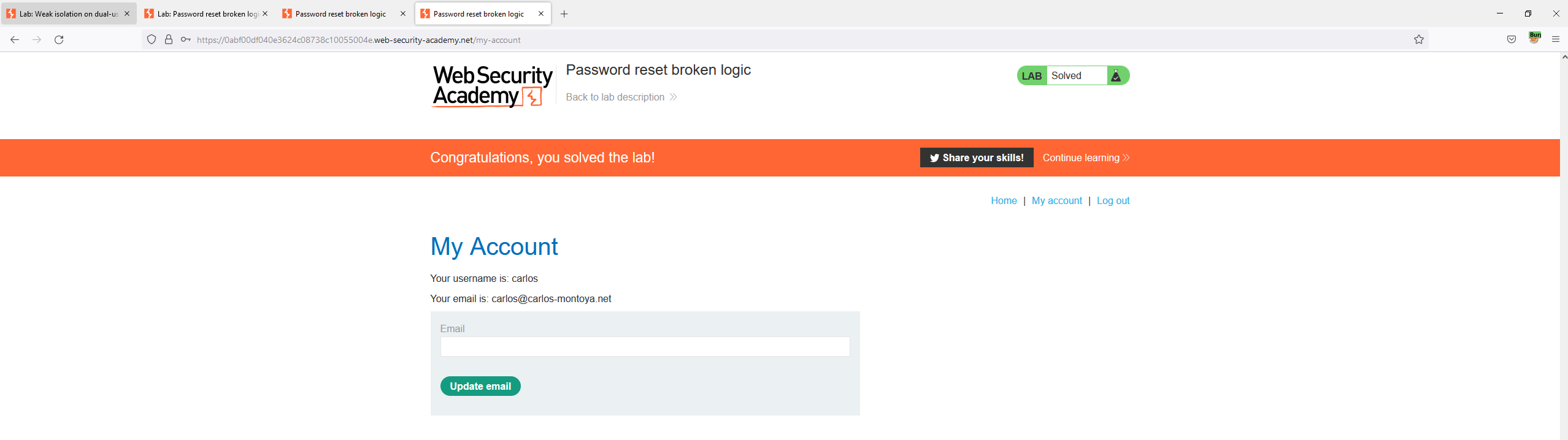
This applies to both **URL** and **POST** parameters, but also check cookies everytime. This simple process can reveal some strange application behaviour that may be exploitable.

**LAB 7: Password reset broken logic:**

This lab's password reset functionality is vulnerable. To solve the lab, reset Carlos's password then log in and access his "My account" page.

1-) I tried so much to Access Carlos user’s my account page from forgot password page but that was not possible. So I tried to reset wiener user’s password and affirmed it on email client so page redirected me to a page which ask us what is the new password and confirmation of the password. When I captured that change request I saw a parameter named as “username=wiener”. When I changed it as “carlos” I have changed password of carlos user’s password. And then I could reached carlos’s my account page.

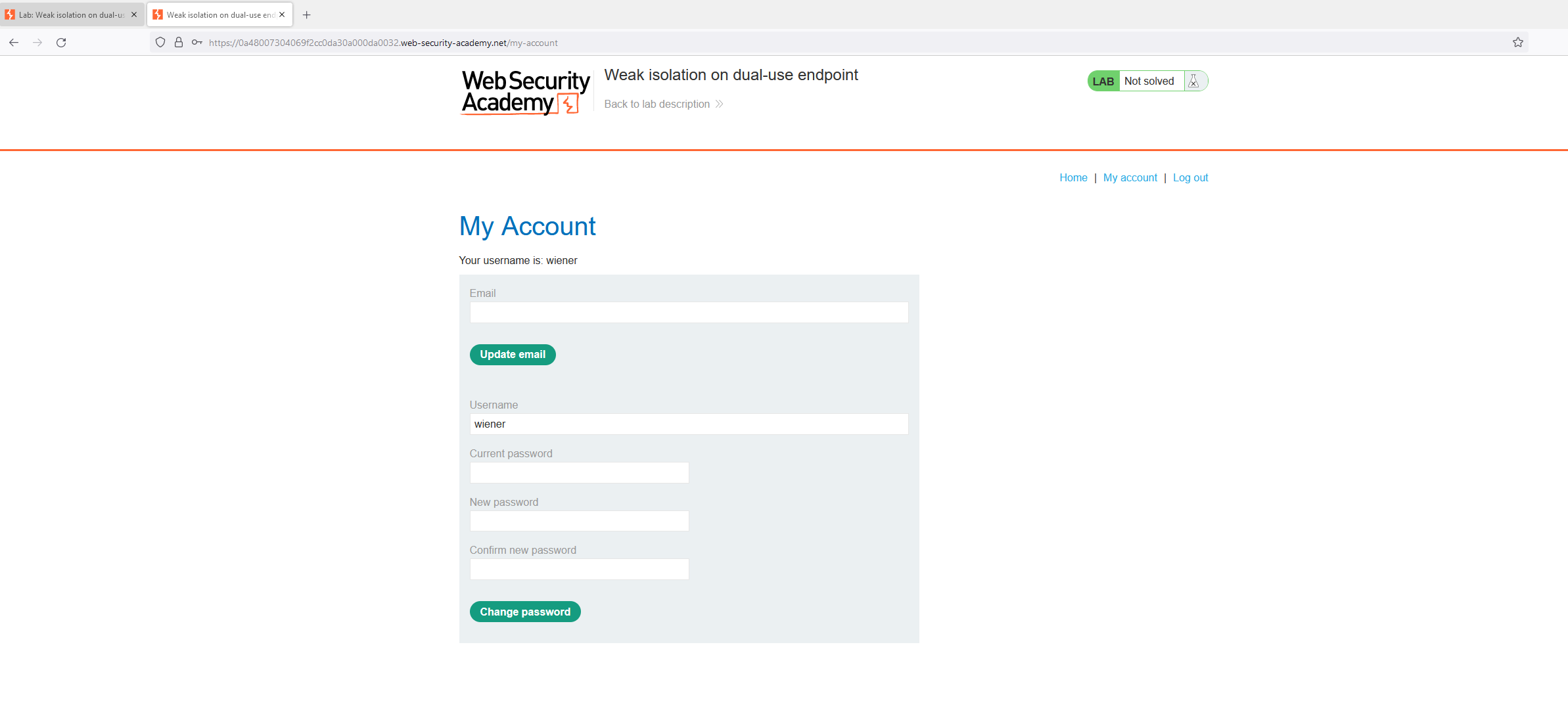
2-) And the lab is solved



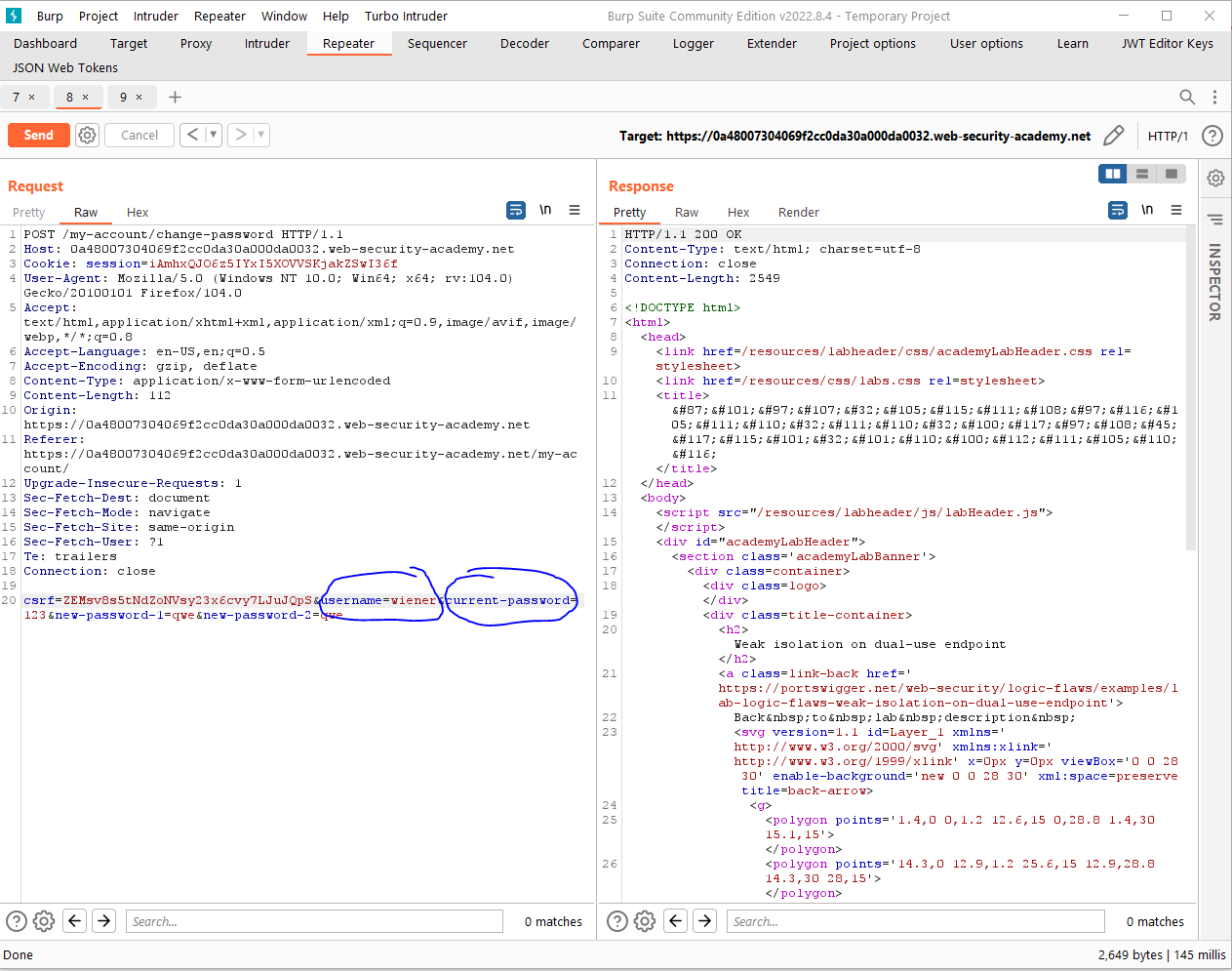
**LAB 8: Weak isolation on dual-use endpoint:**

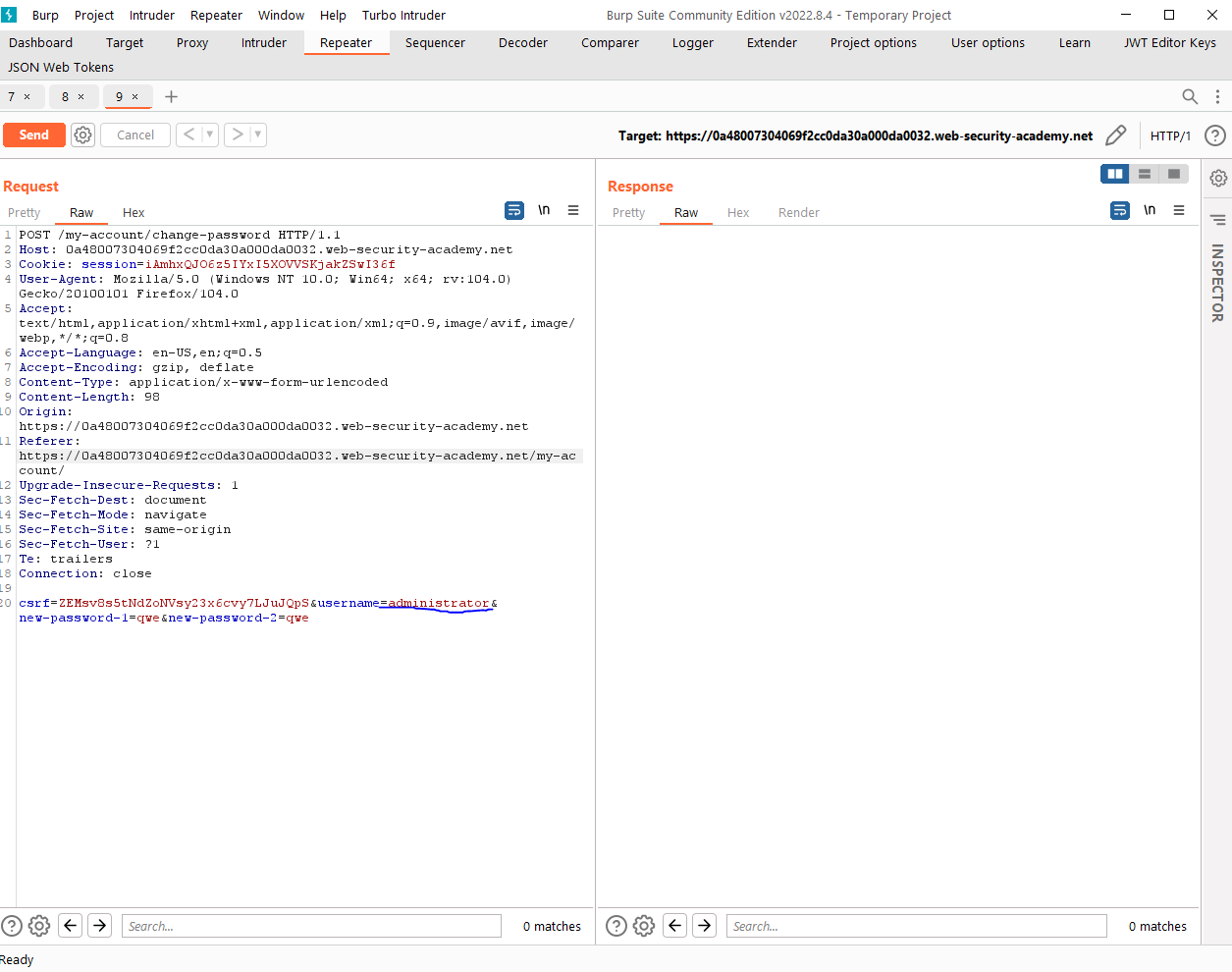
This lab makes a flawed assumption about the user's privilege level based on their input. As a result, you can exploit the logic of its account management features to gain access to arbitrary users' accounts. To solve the lab, access the administrator account and delete Carlos.

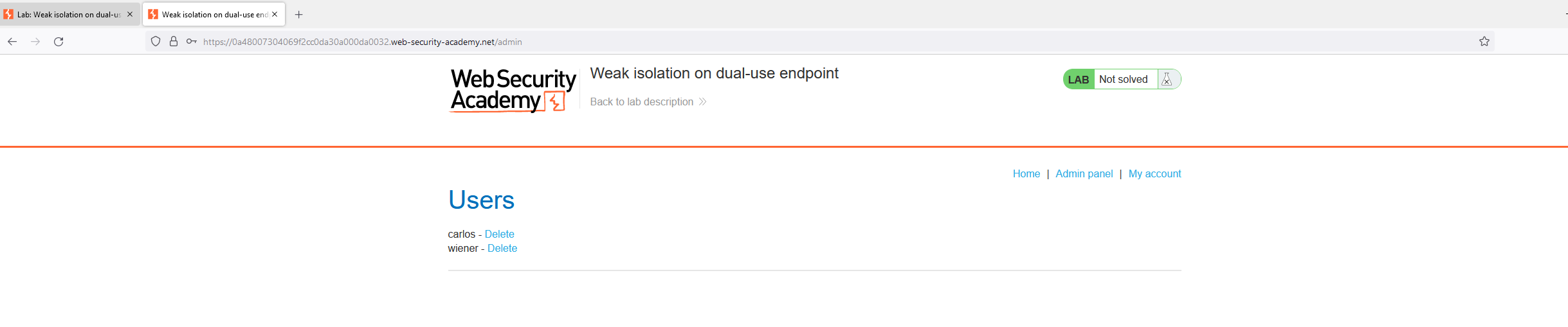
1-) I logged in as wiener user’s account and saw password changing page

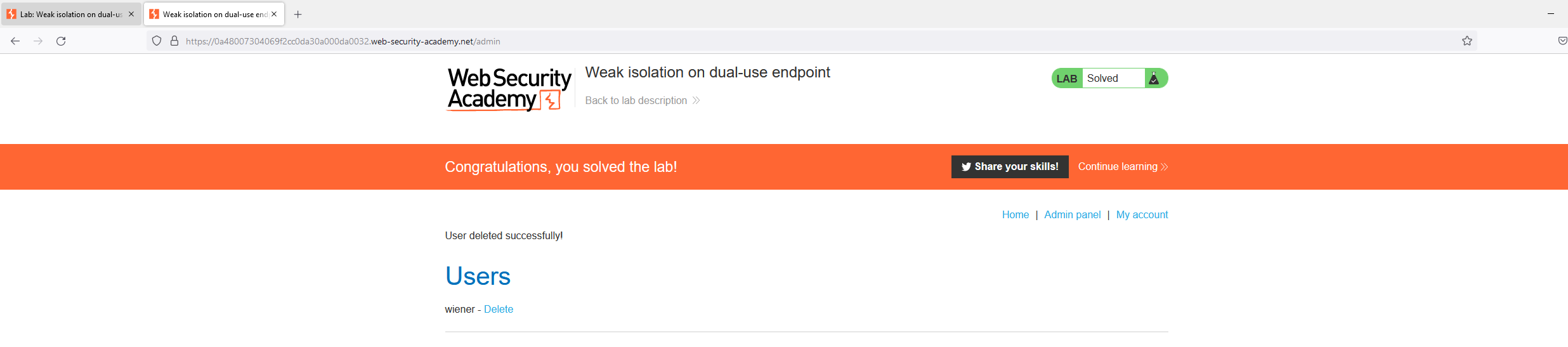


2-) I captured the password reset request and tried to change the username=”admin” but the app is giving error because current password was not corret. So at first, I tried to delete only the value, nothing happened. But after that when I delete the whole parameter, password was changed for wiener even without current password field. So I tried it with “admin” username but app says there were no user at that name. After that, when I changed username=”administrator” the request was successful. and I deleted user carlos









**Users won't always follow the intended sequence:**

So many requests rely on predefined workflows consisting of a sequence of steps. Web interface guide users through this process. Taking them to the next step of the workflow eachtime when they complete the current one. But, attackers may not adhere to this intended sequence. Not preventing this situation can lead dangerous flaws that may be exploitable.

For example, many websites that implements 2-FA requires user login on one page before redirect user to another page to enter 2-FA code. Assuming that users will always follow this porcess through to completion and, as a result, not verfying what they do, may allow attackers to bypass the 2FA entirely.

Assumption making about sequence of events can lead wide range of problems even in the same workflow or functionality. By using Burp Proxy and Repeater attacker can capture and replay the request whenever they want and this allows them to complete different actions while the app is an unexpected state.

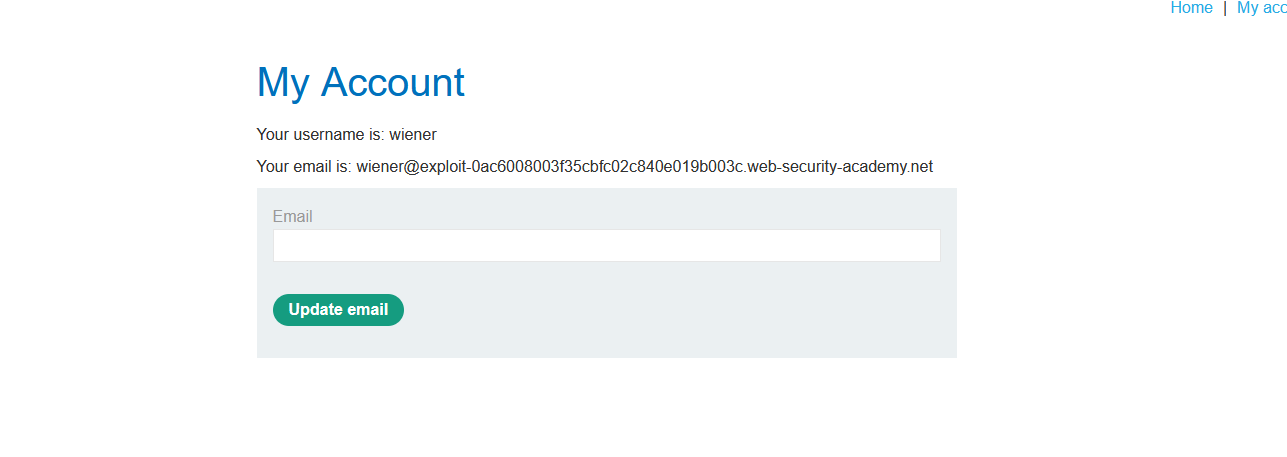
To identify these kind of flaws, we should use forced browsing to submit request in an unintended sequence. For example, we can skip some steps, Access a single step more than once, return earlier steps and so on. We need to take notes of how different steps are accessed. Although we often just submit a **GET** or **POST** request to a specific URL, sometimes we can Access steps by submitting different sets of parameters to the same URL. As we do like in any logic flaw, we need to find what assumptions the developers have made and where the attack surface lies. After that we can look for a way to exploit them.

\***Important Note:** This kind of testing will often cause exceptions because expected variables have null or uninitialized values. In this case, pay close attention to any error messages or debug information because they can be a really valuable source of information disclosure, which can help us to fine-tune our attack and understand key details about the back-end behaviour

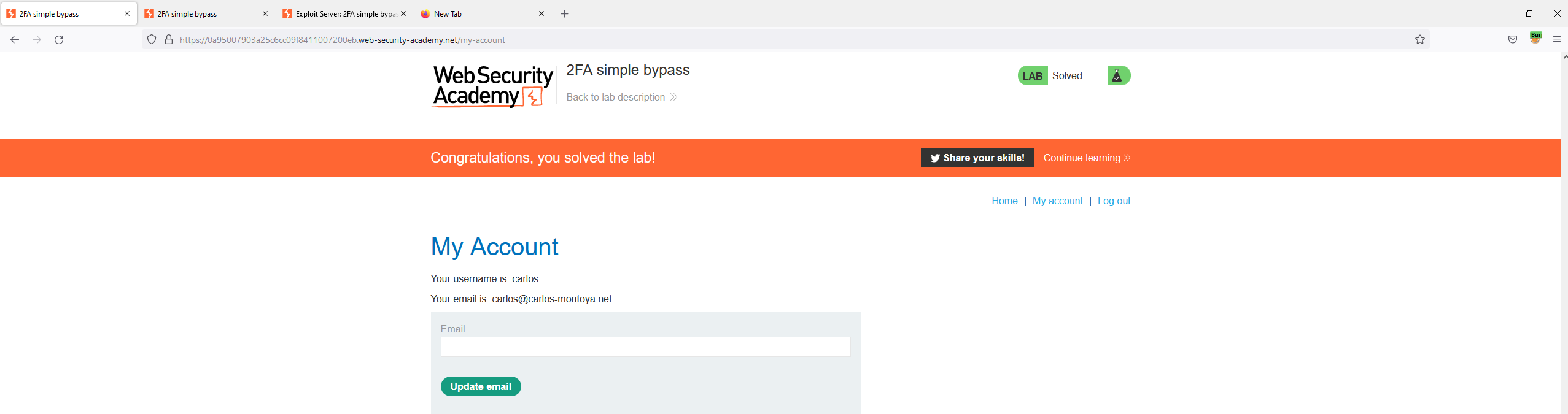
**LAB 9: Weak isolation on dual-use endpoint:**

This lab's two-factor authentication can be bypassed. You have already obtained a valid username and password, but do not have access to the user's 2FA verification code. To solve the lab, access Carlos's account page.

1-) I entered the wiener user’s account and app wanted 2-FA code from me. When I entered the code app redirect me to the wiener’s my account page



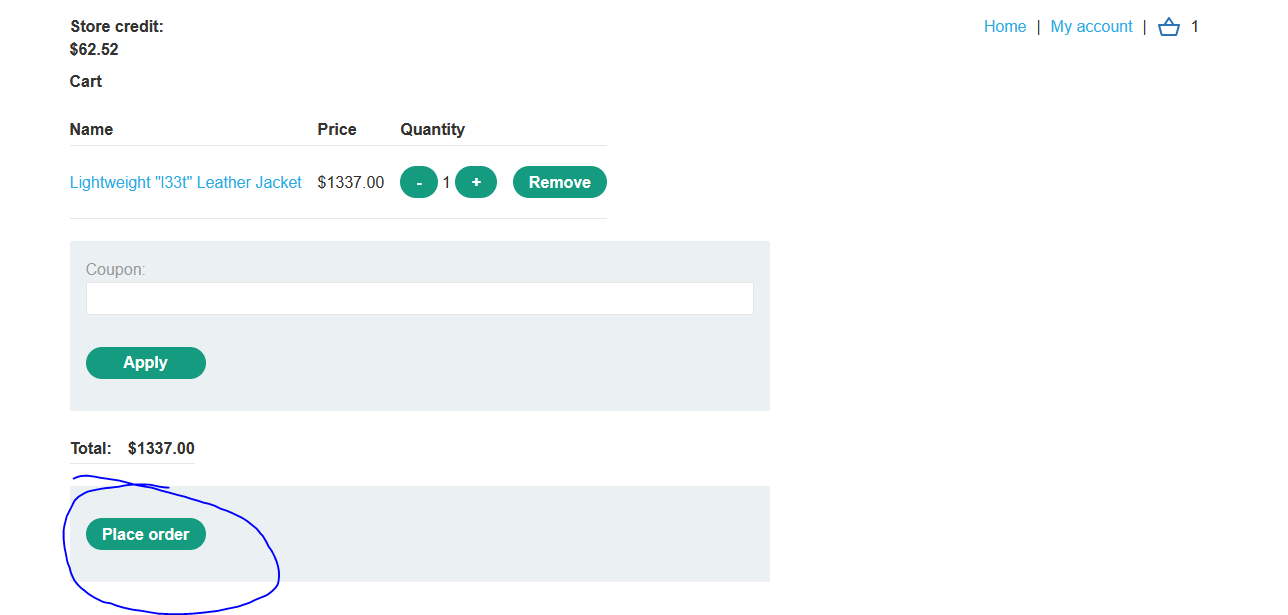
2-) When I try to enter carlos’s account app wanted 2-FA code from me but I copied the my account page link from wiener’s my account page url. Server did not properly control the 2-FA code mechanism. When I copy the my account page link after logged in as Carlos, App redirect me to the Carlos’s account page even without 2-FA code has entered(When I paste my account page url, app looked cookies on my request and I had Carlos’s account’s verified cookies. Then didnt check for 2-FA completion process and redirected me to the acount page.)

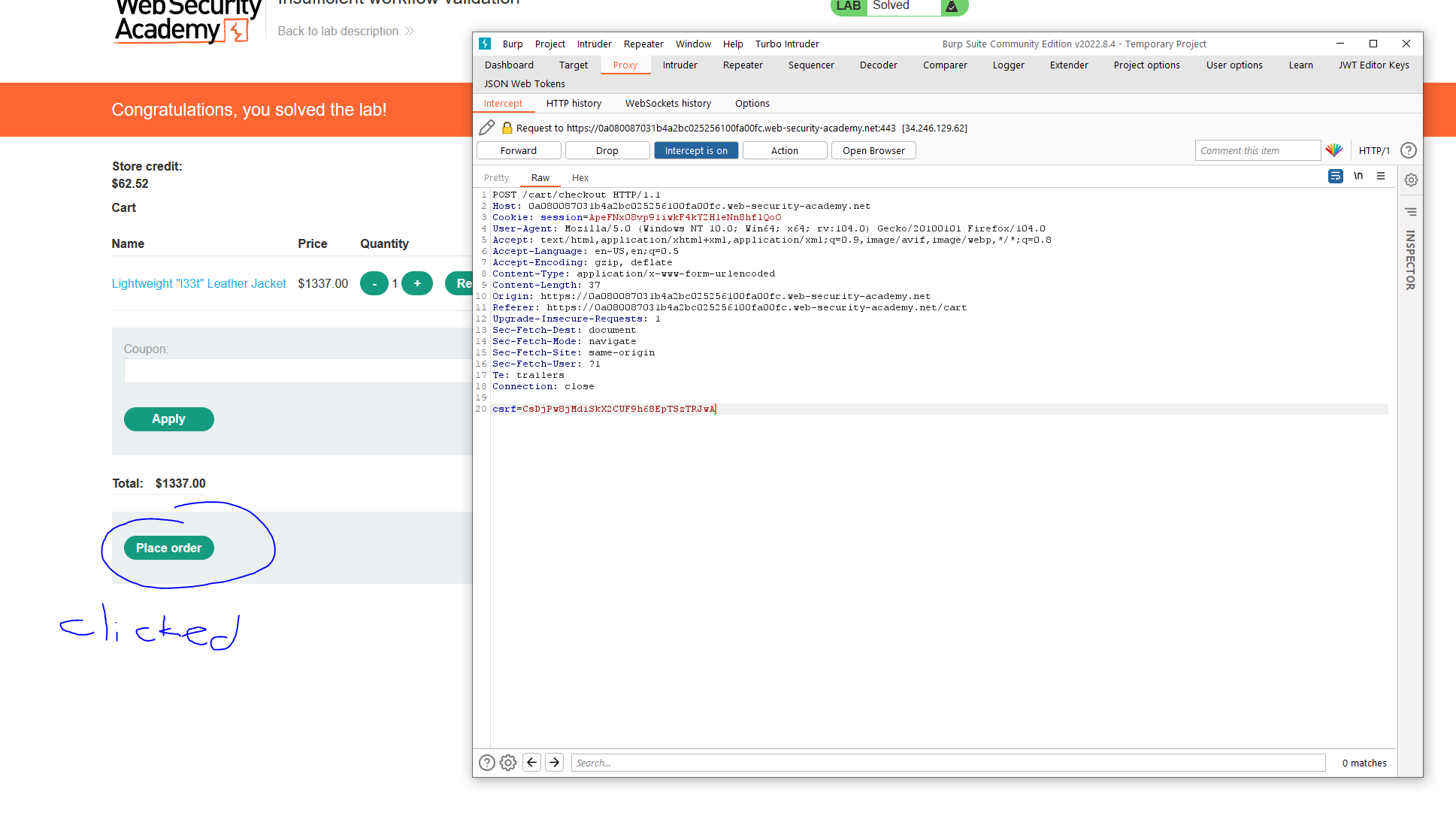


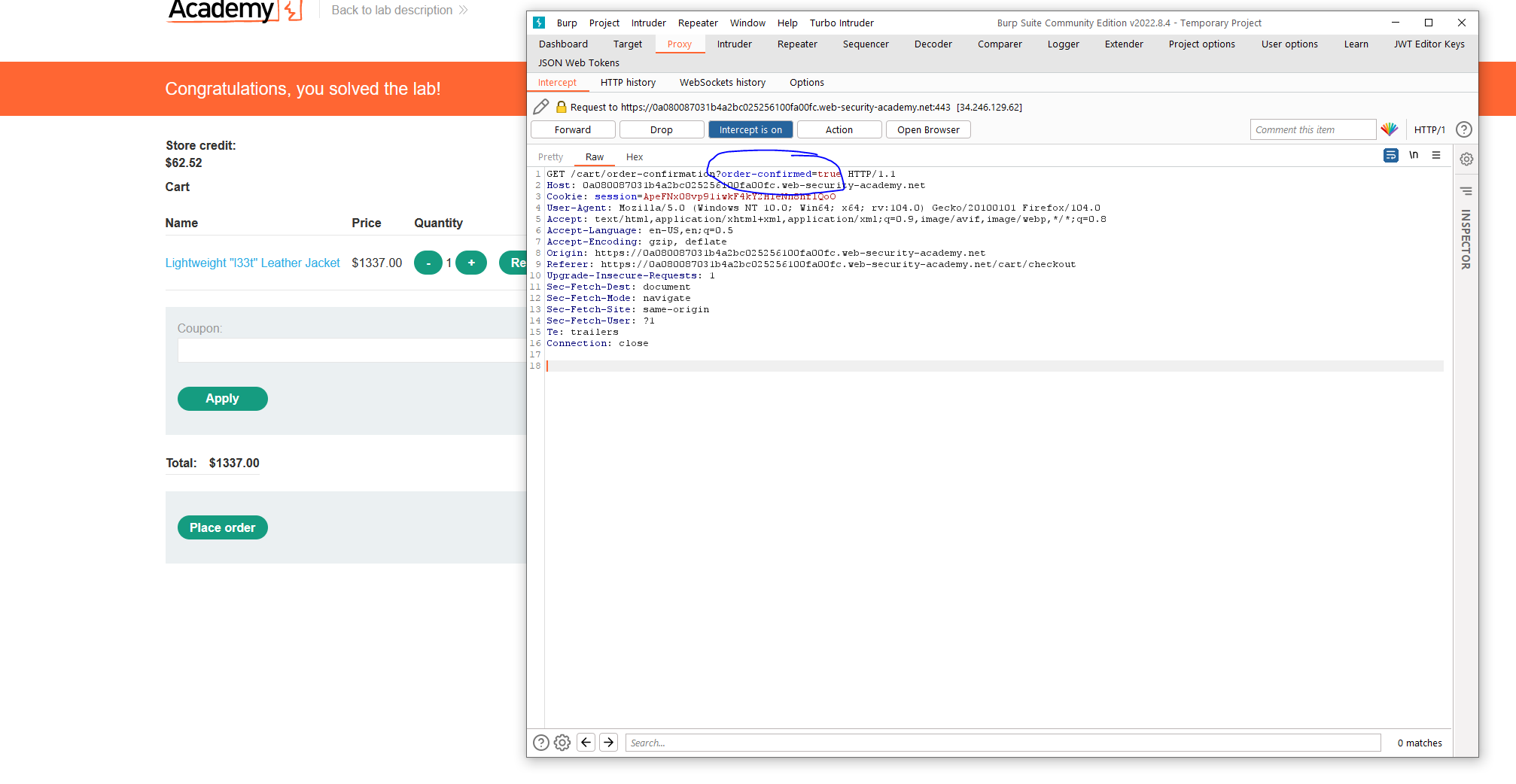
**LAB 10: Insufficient workflow validation:**

This lab makes flawed assumptions about the sequence of events in the purchasing workflow. To solve the lab, exploit this flaw to buy a "Lightweight l33t leather jacket".

1. First I bought a 9$ pillow from the market and captured the order accepted request with Proxy.
2. Then I added jacket to the cart and repeated order confirmation request on that page and Jacket has been bought from me

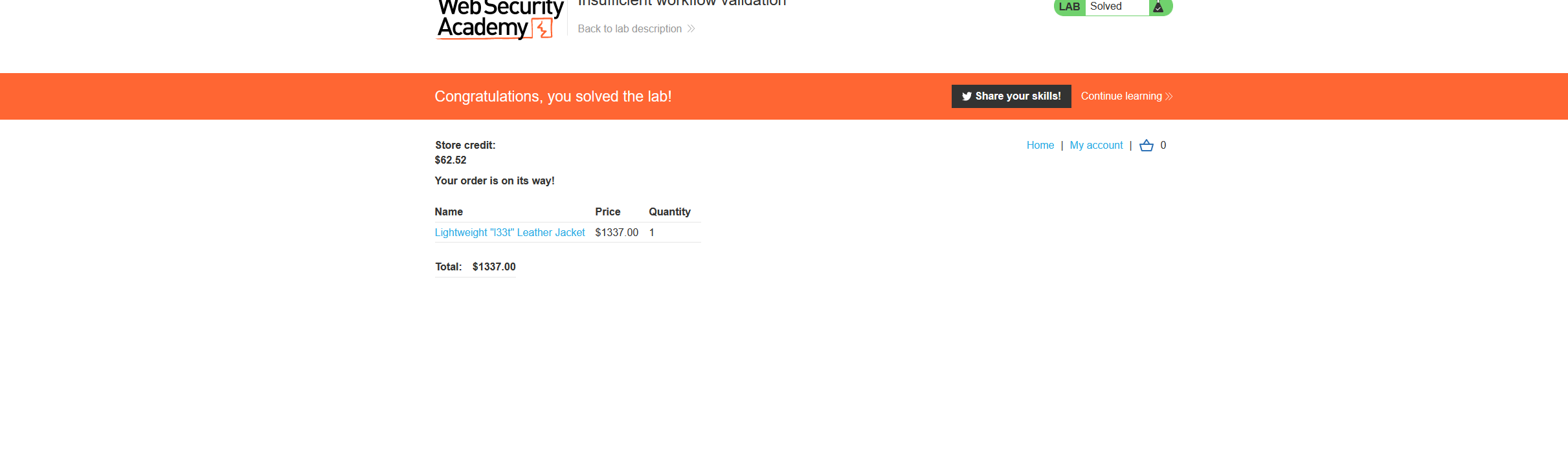






(Changed request with the order confirmation request that I have captured when buying 9$ pillow)

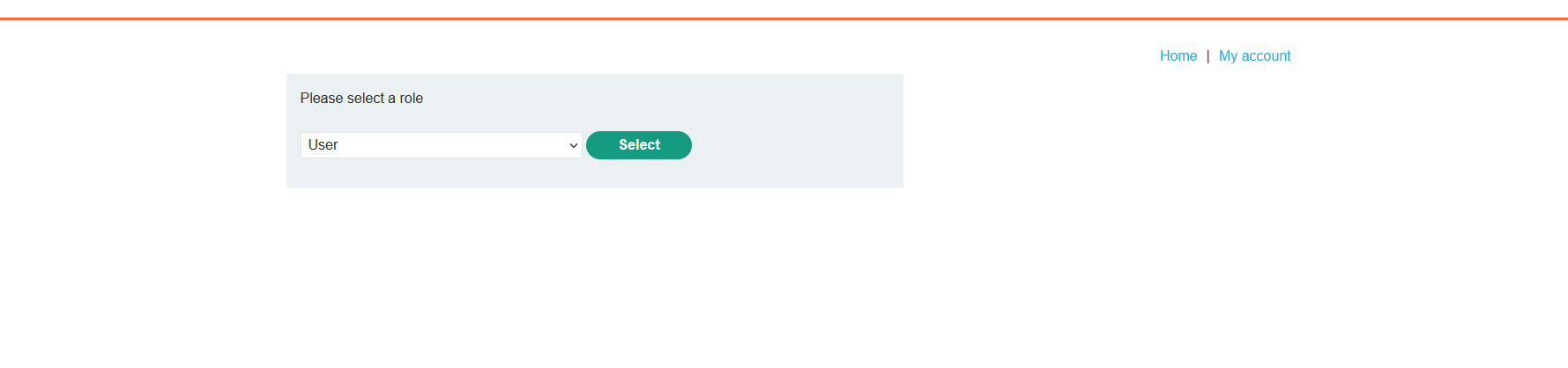
3-) Lab is done



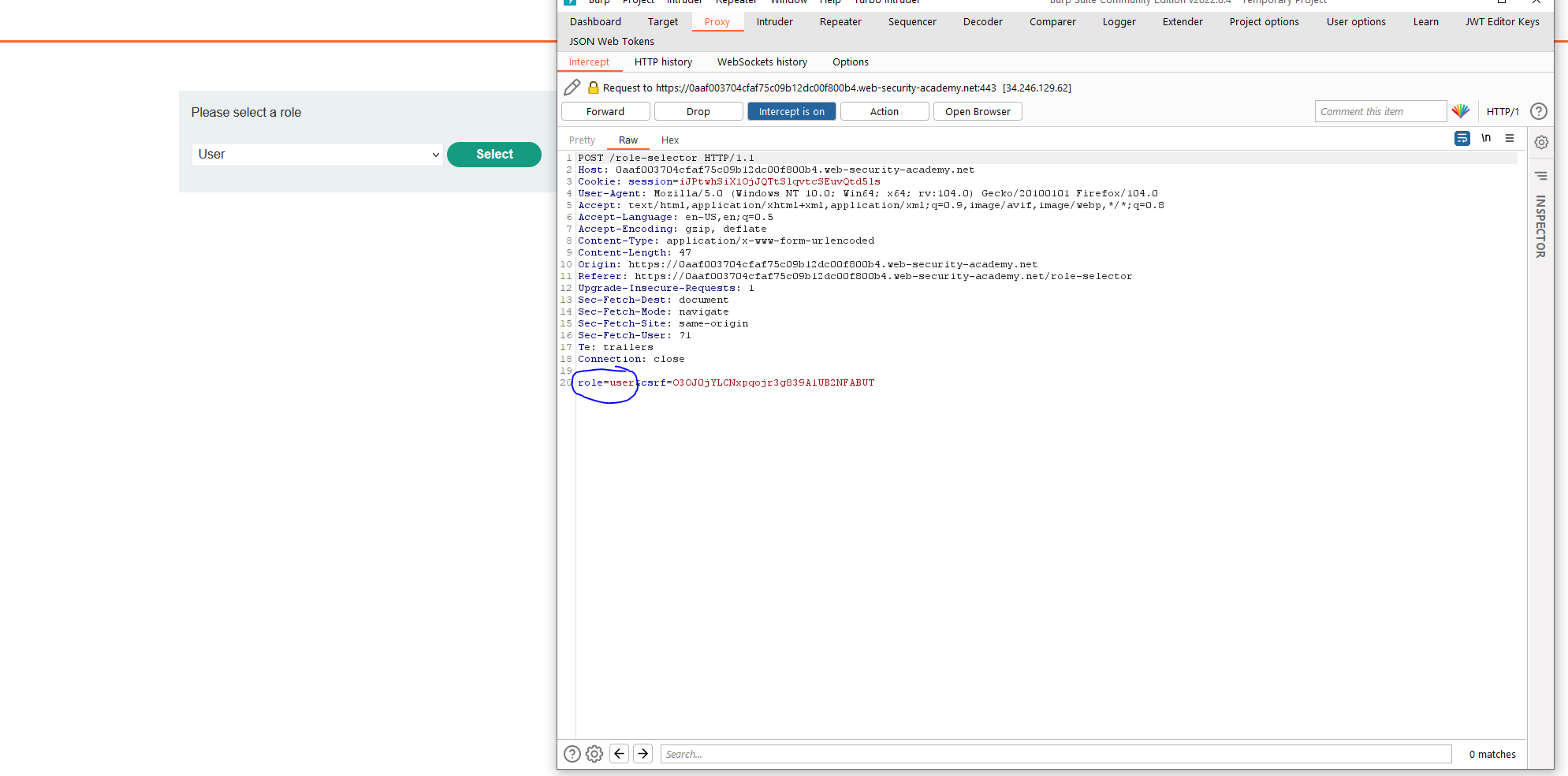
**LAB 11: Authentication bypass via flawed state machine:**

This lab makes flawed assumptions about the sequence of events in the login process. To solve the lab, exploit this flaw to bypass the lab's authentication, access the admin interface, and delete Carlos.

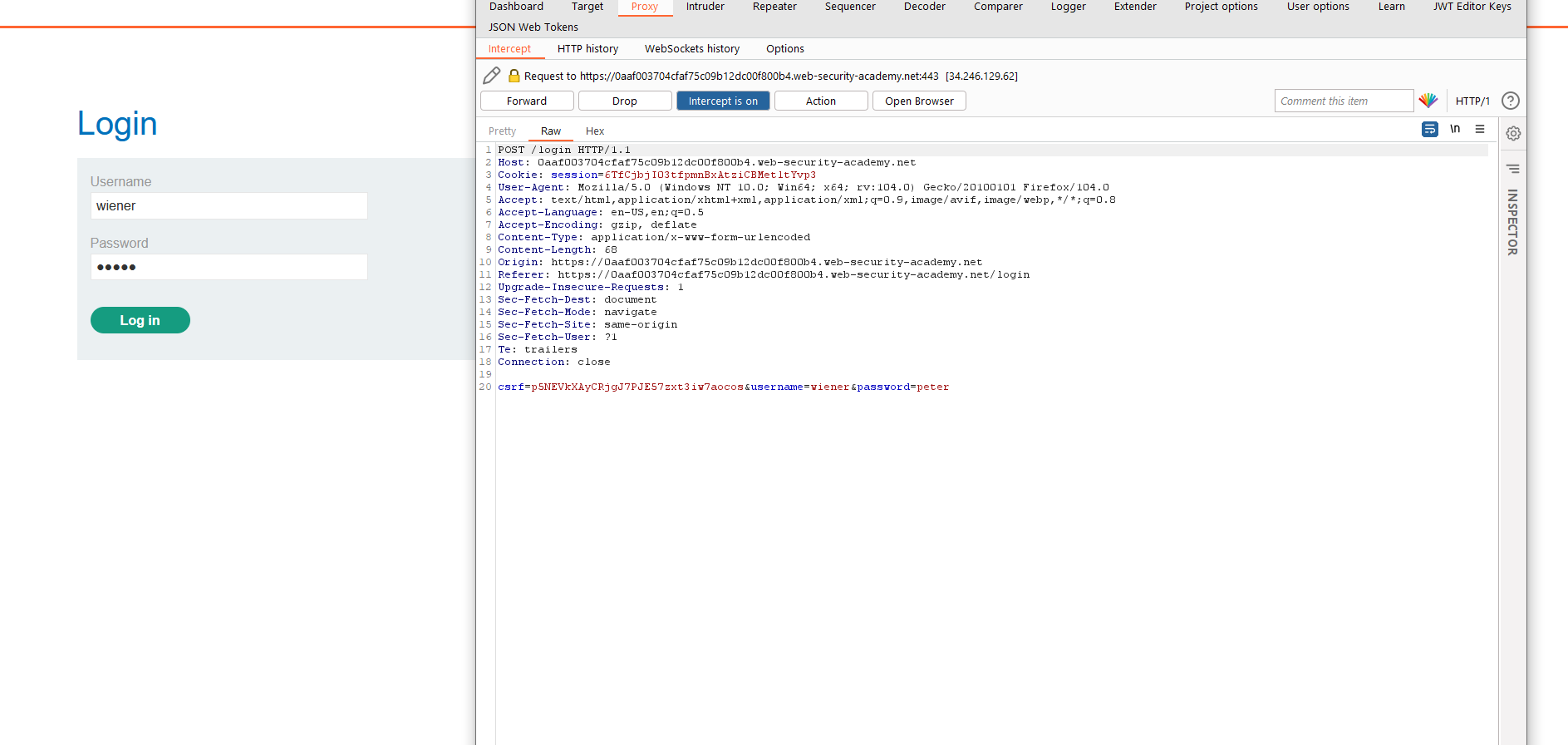
1-) First the app wants us to login the application. After the login screen app wants us to select the role.



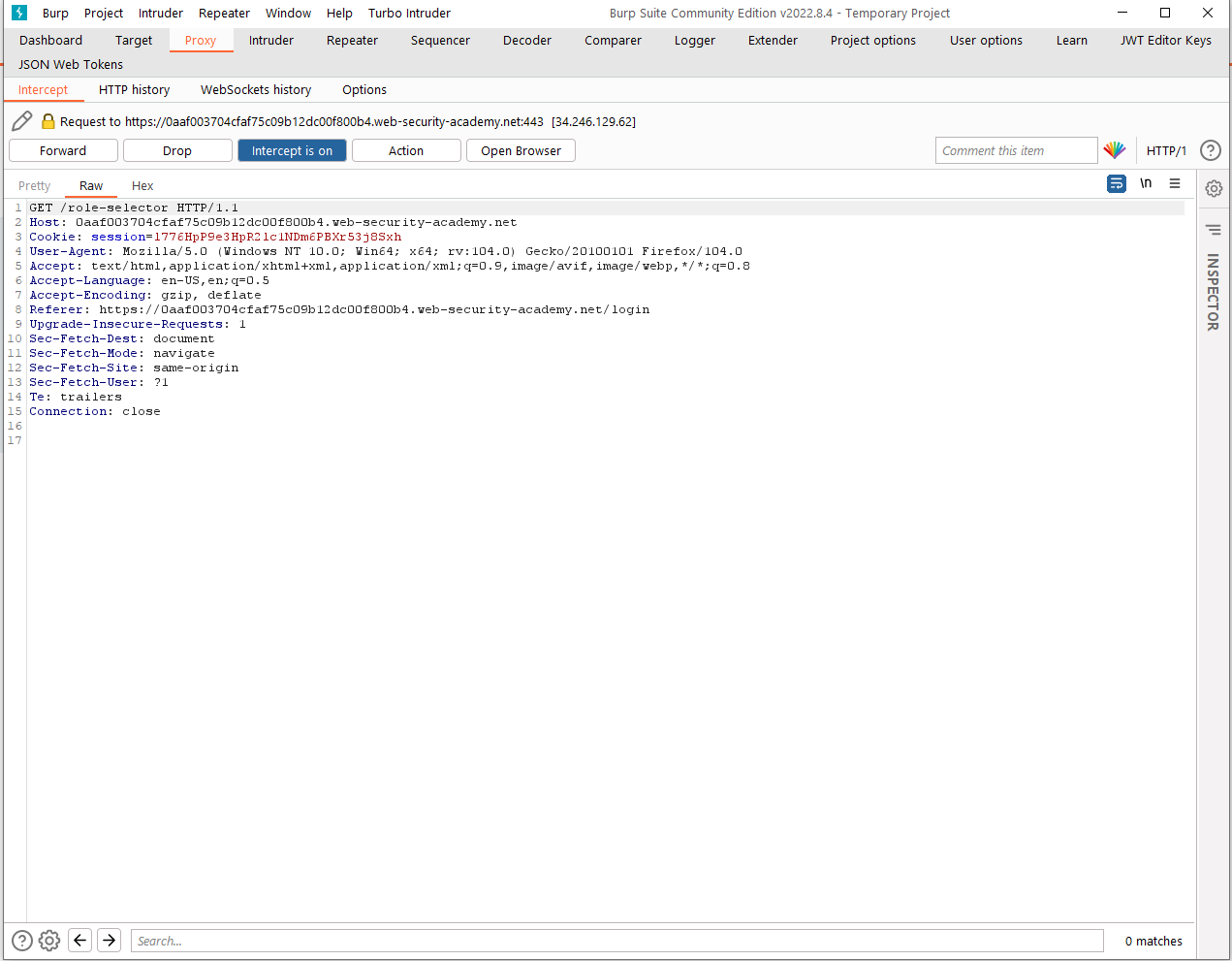
2-) I captured the role select request but no matter what I tried, couldn’t make myself as admin role in this request.(Changing role as admin doesn’t work)



3-) So I tried to login the app and drop the role selector page’s get request. See if I can drop the request then maybe app gives me the user permission of No:1 id in the database and that most frequently is admin user.

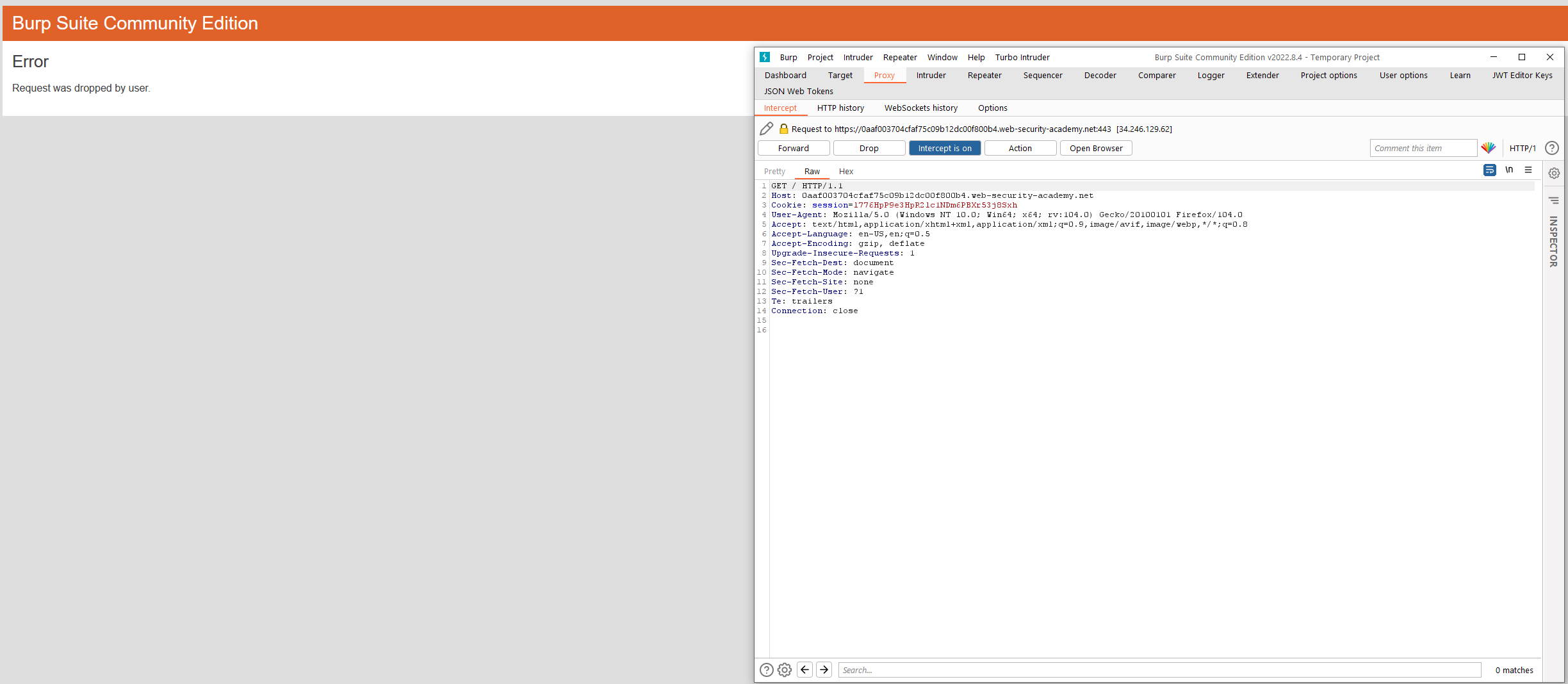


(Login request forwarded)



(Role selector page’s request dropped)

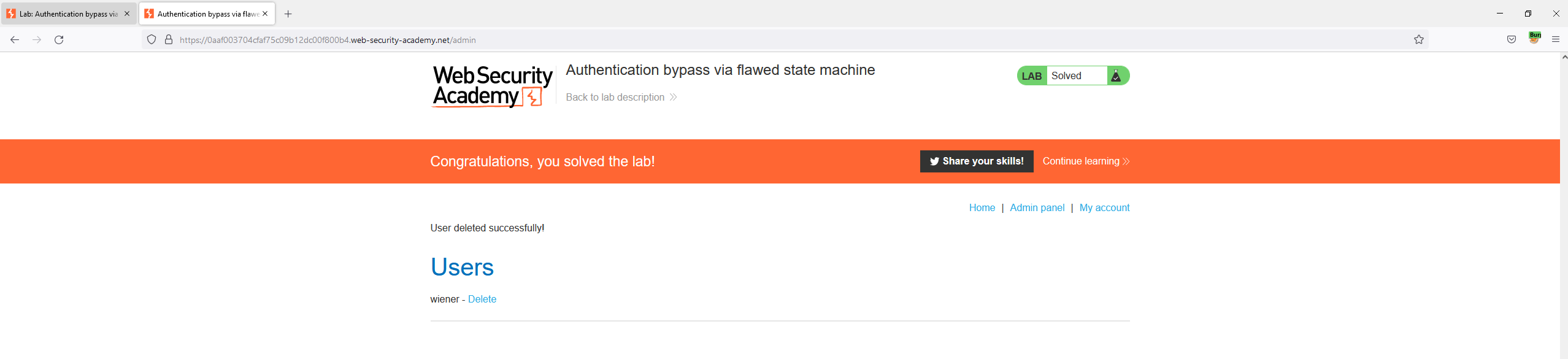
4-) Then tried to go to home page after request was dropped by copying the home link to the url





(Admin panel has reached)

5-) Lab is completed



**4-) Domain-specific flaws:**

In many cases, we will encounter logic flaws that spesific to a business domain or purpose of the site.

The discounting functionality of online shops is a classic attack surface when hunting for logic flaws. This oftenly occured flow can be a gold mine for attacker with all kinds of basic logic flaws occuring in the way that discounts are applied.

For Example, an app gives users who order over 1000$, %10 discount. This can be vulnerable because if business logic fails to check whether the order was changed after the discount is applied. In this case, attackers can add products to his/her card until $1000 to gain the discount, after that, remove the items that they dont want to buy before placing order. They would recieve the discount even order doesn’t satisfies the intended criteria.

We should pay close attention to any situation where prices or any sensitive values are determined or adjusted by user actions. Try to understand what algorithm the application uses to make these adjustments and at what point that adjustments are made. This often involves manipulating the app so app will be in a state where applied adjustments do not fit the original criteria that intended by developers.

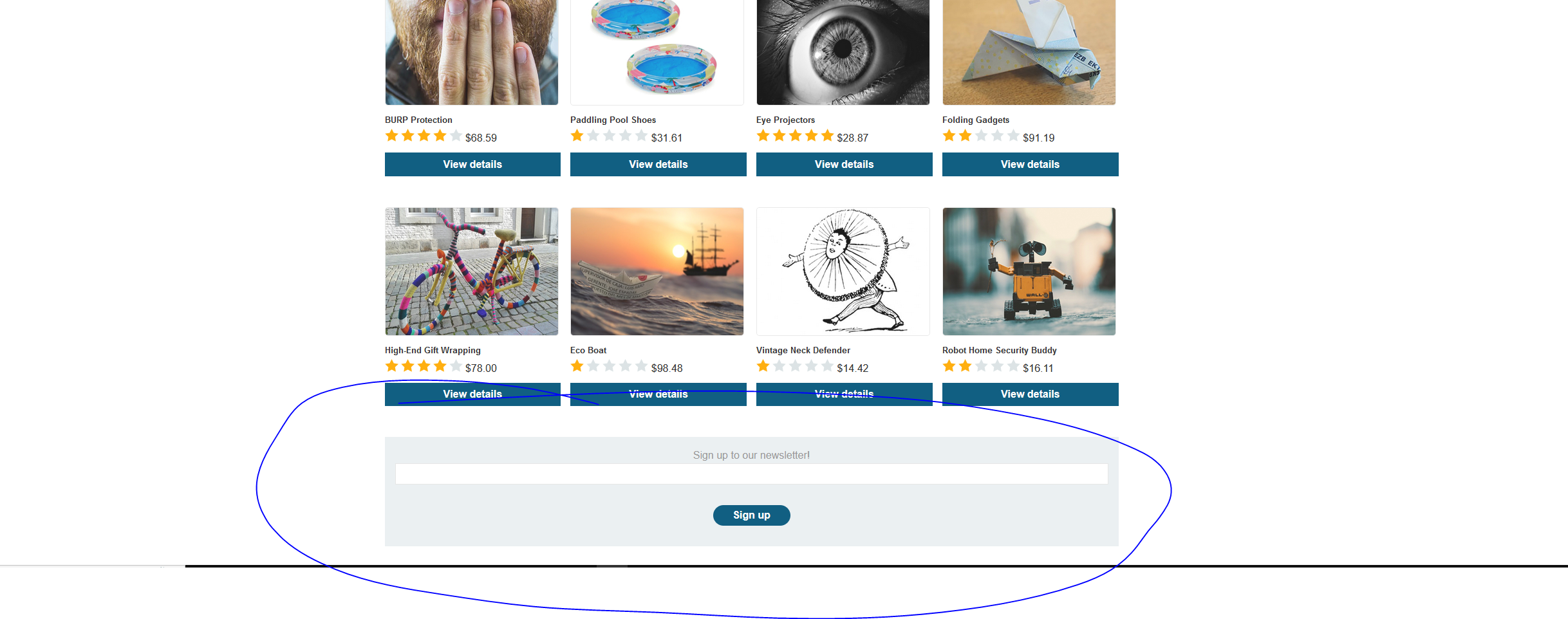
To identify these vulnerabilities, we need to think carefully about what objectives that attacker might try to find different ways of achieving this using the provided functionality. This may require a certain level of domain-specific knowledge in order to understand what might be adventageous in a given context. For instance, we need to understand social media to be able to know how to gain large number of followers.

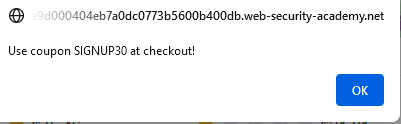
Without this knowledge we can dismiss dangerous behaviours because simply we are not aware of it’s potential effects. Likewise, we may struggle to join the dots and notice how two functions can be combined in a harmful way. As a Security Engineer, we will encounter apps from less familiar domains to us. In this case, we must read as much documentation as possible and also talk with subject-matter experts if possible to gain knowledge about that domain. **More obscure domain means, other testers will have missed plenty of bugs.**

**LAB 12: Flawed enforcement of business rules:**

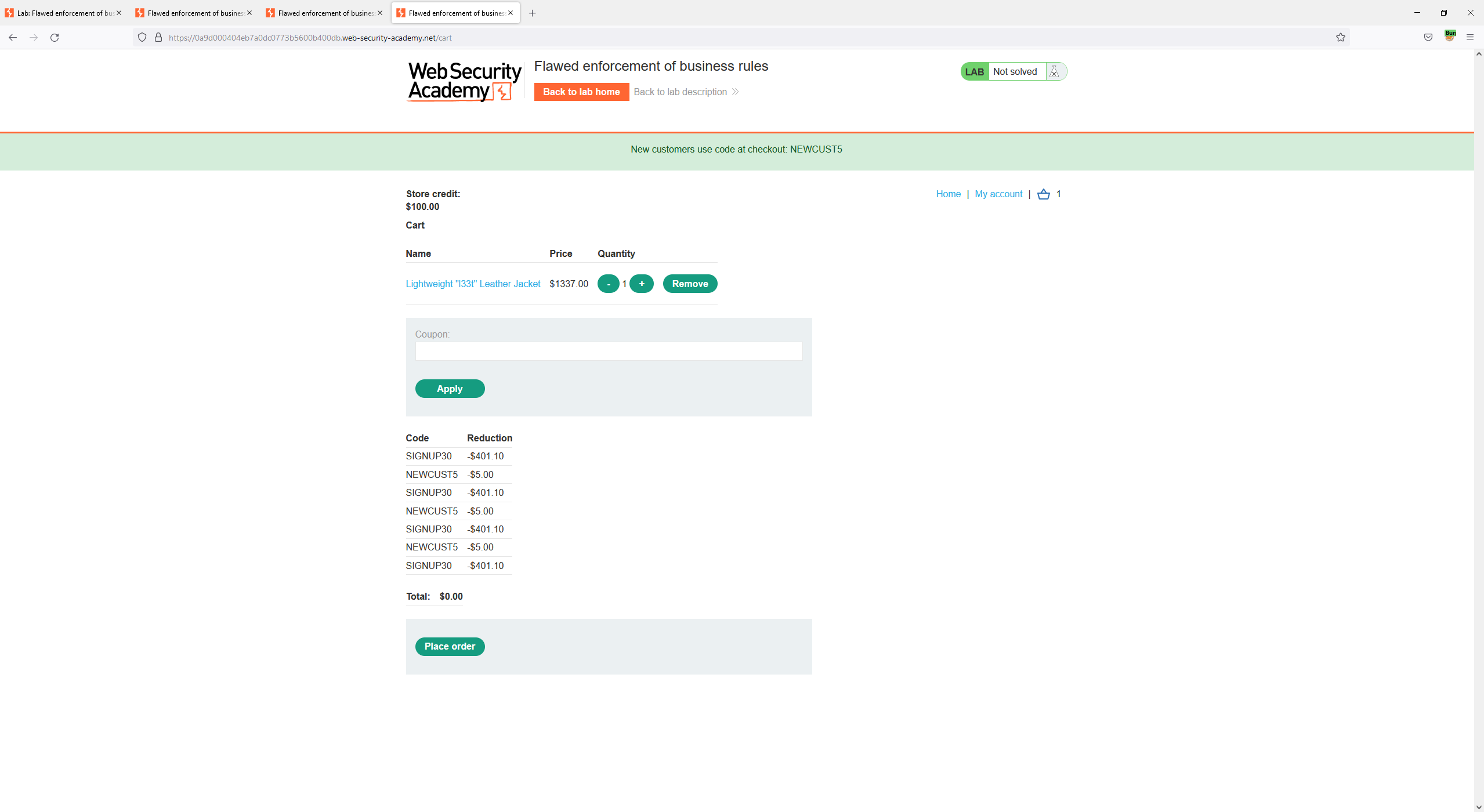
This lab has a logic flaw in its purchasing workflow. To solve the lab, exploit this flaw to buy a "Lightweight l33t leather jacket".

1. App give new customers a discount which is 5$. Also bottom of the page if we sign up to their newsletter page also give 400$ discount too.

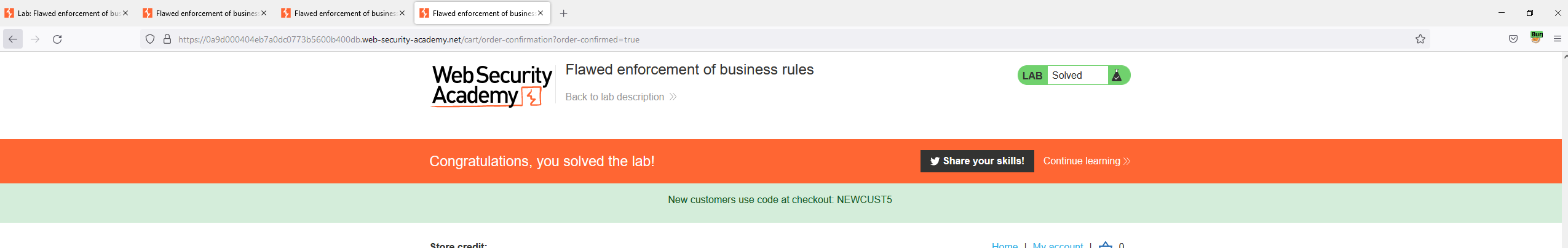




2-) If we try to apply same coupon twice app gives error. But if we apply coupons one after another app gives discounts for every apply transaction.



3-) Lab is done.



**LAB 13: Infinite money logic flaw:**

This lab has a logic flaw in its purchasing workflow. To solve the lab, exploit this flaw to buy a "Lightweight l33t leather jacket".

1. When we register with an email to get discount the below of the page it gives us %30 discount. Application also sell gift cards for 10$(also when you enter gift card code to your my account page it will give 10$ back). If we buy gift cards with %30 discount code everytime. So we are gaining 3$ everytime.
2. We need to macro all these request sequence in Burp to gain 3$ fast. Go to Project options(Sessions tab) -> Session Handling Rules-Add (Scope) -> URL Scope (Include all URLs) -> Go to details tab -> Rule Actions Add (run a macro).
3. In the opened page Macro Reader -> Select these 5 request And then click OK

POST -> /cart

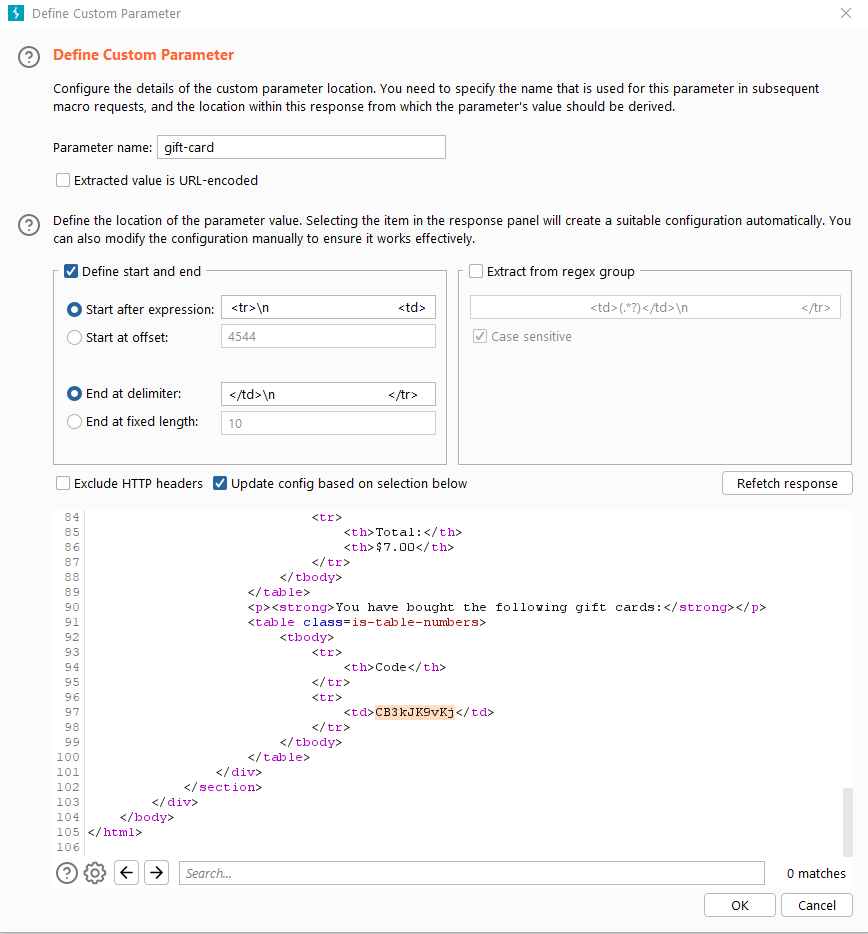
POST -> /cart/coupon

POST -> /cart/checkout

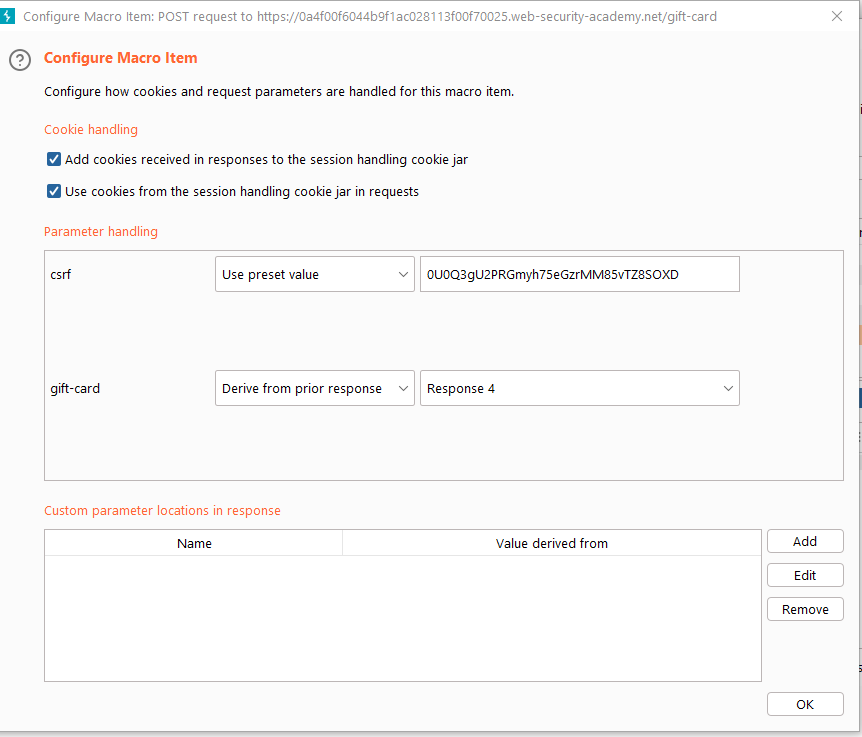
GET -> /cart/order-confirmation?order….

POST -> /gift-card

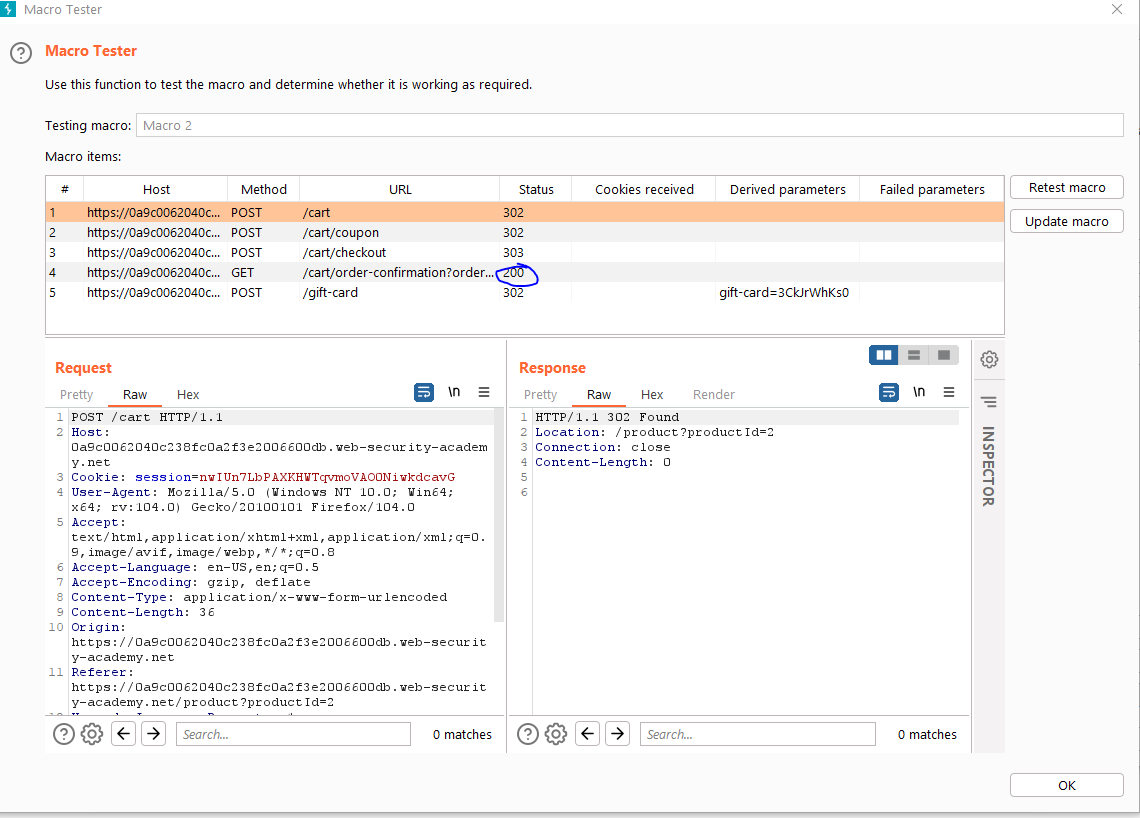
1. Click Get request on macro editor and Configure item button on right. At Custom parameter locations in response -> Add -> Parameter name: gift-card
2. At the bottom of the page in the response part find gift card code and cross out with Mouse. And after that click OK twice to close the opened tabs.



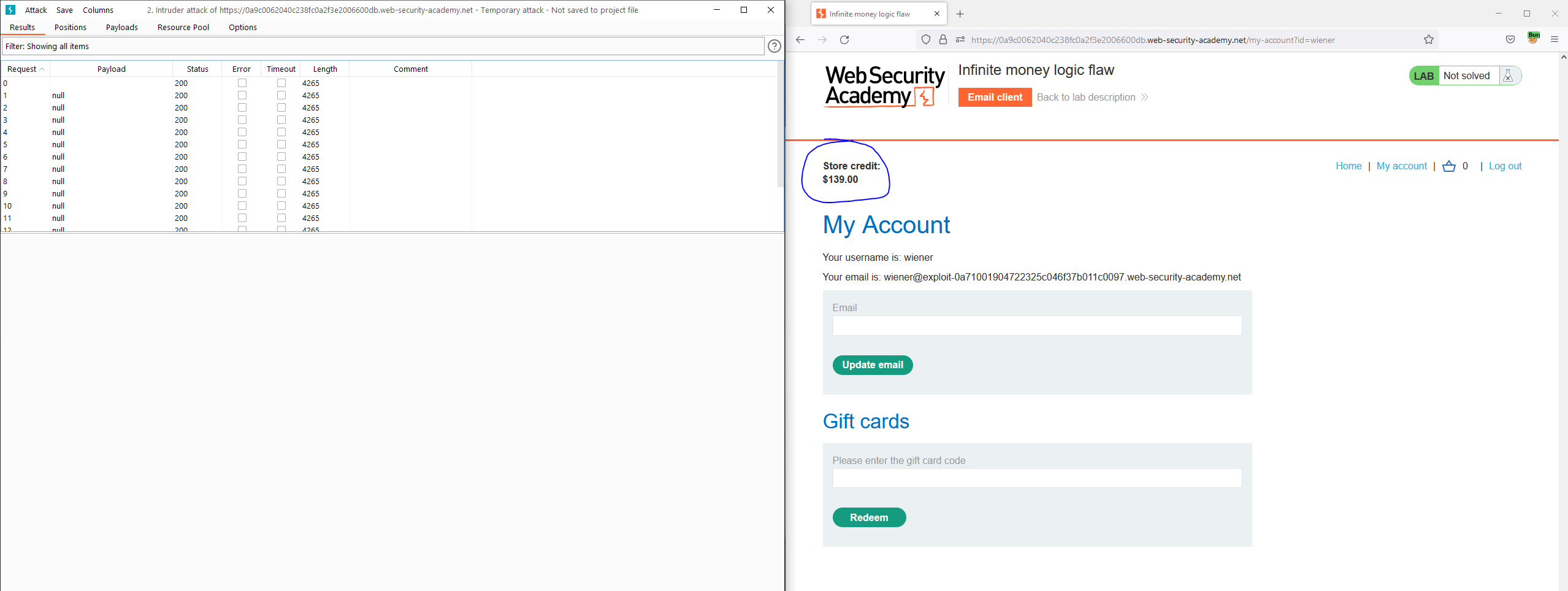
1. Select /gift-card POST request -> Configure item -> At Parameter handling frame -> gift-card (Derive from prior response)-(response4) and click OK



1. Test macro at the Macro Editor Page and we will see that we are getting Http 200 response on get request and also 302 on POST /gift-card we can see that gift card code is new code that we got from earlier. Click OK 4 times



1. Go to Http History -> Find GET /my account request -> Send it to Burp Intruder -> Click Clear button at right -> Attack Type : Sniper
2. Payloads -> Payload Type : null payloads -> Payload Options (Generate 412 payloads) -> And Start Attack. After that we can see that our Money is growing at a time.



1. The lab is completed (After 1.15 hour)



**4-) Providing an encryption oracle:**

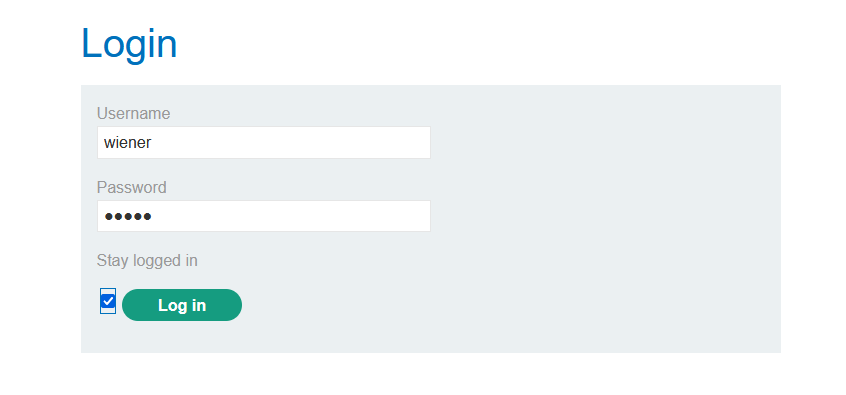
Dangerous scenarios can happen when user-controllable input is encrypted then resulting ciphertext is showed back to the user in some places. This kind of input sometimes known as “encryption oracle”. An attacker can use this input to encrypt arbitrary data using the correct algorithm and asymmetric key.

This becomes dangerous when there are other user-controllable inputs in the application that expect data encrypted with the same algorithm. In this case, attacker can user encryption oracle to create valid encrypted input and then pass it into other sensitive functions.

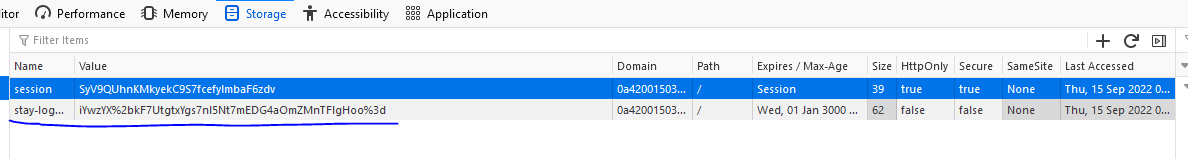
This issue can be combined if there is another user-controllable input on the site that provides the reverse function (decrypting mechanism). This helps attacker to decrypt other data to identify expected structure. This helps attacker to save some work when creating malicious data but it is not necessarliy required to craft a successful exploit. The severity of an encryption oracle depends on what functionality also uses the same algorithm as the oracle.

**LAB 14: Authentication bypass via encryption oracle:**

1-) At first, we are seeing that developers have added stay logged in check in login page

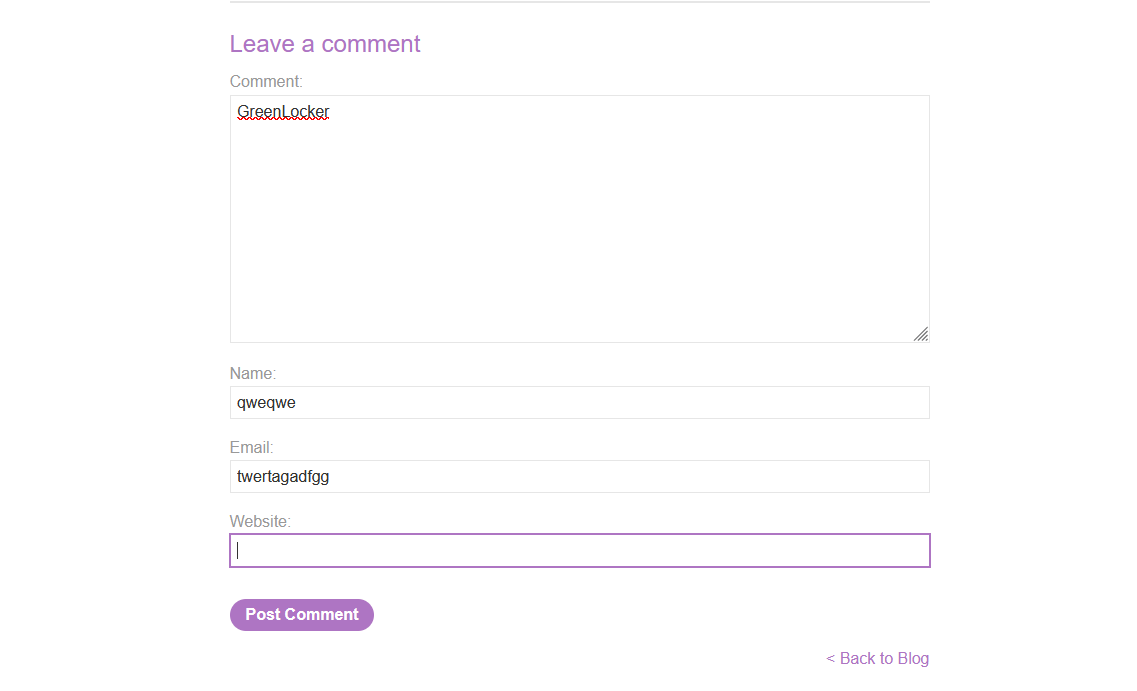


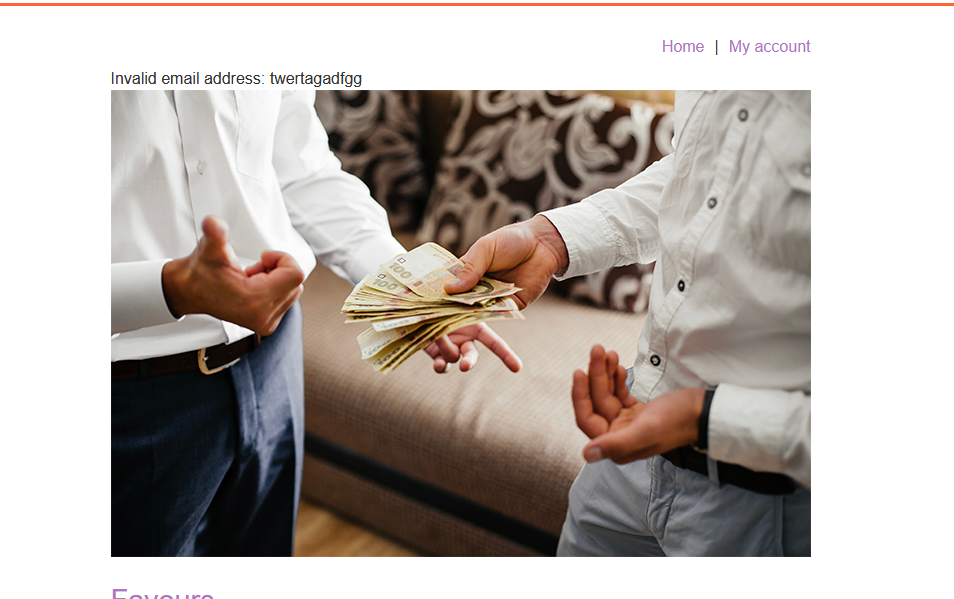
2-) After we logged in, in cookies there is new, ENCRYPTED value is created :

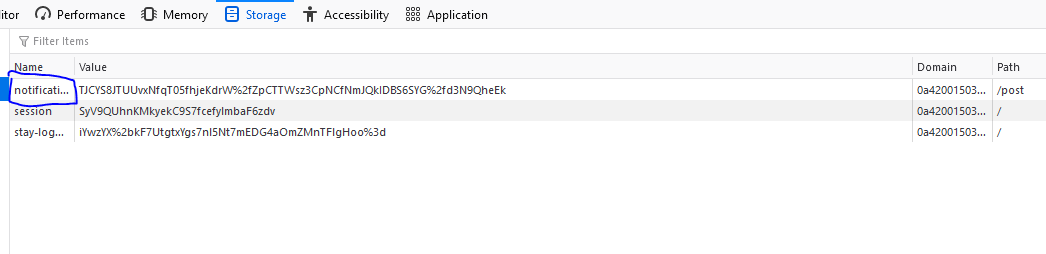


3-) Now I try functionalities of the website so maybe I can get something useful.

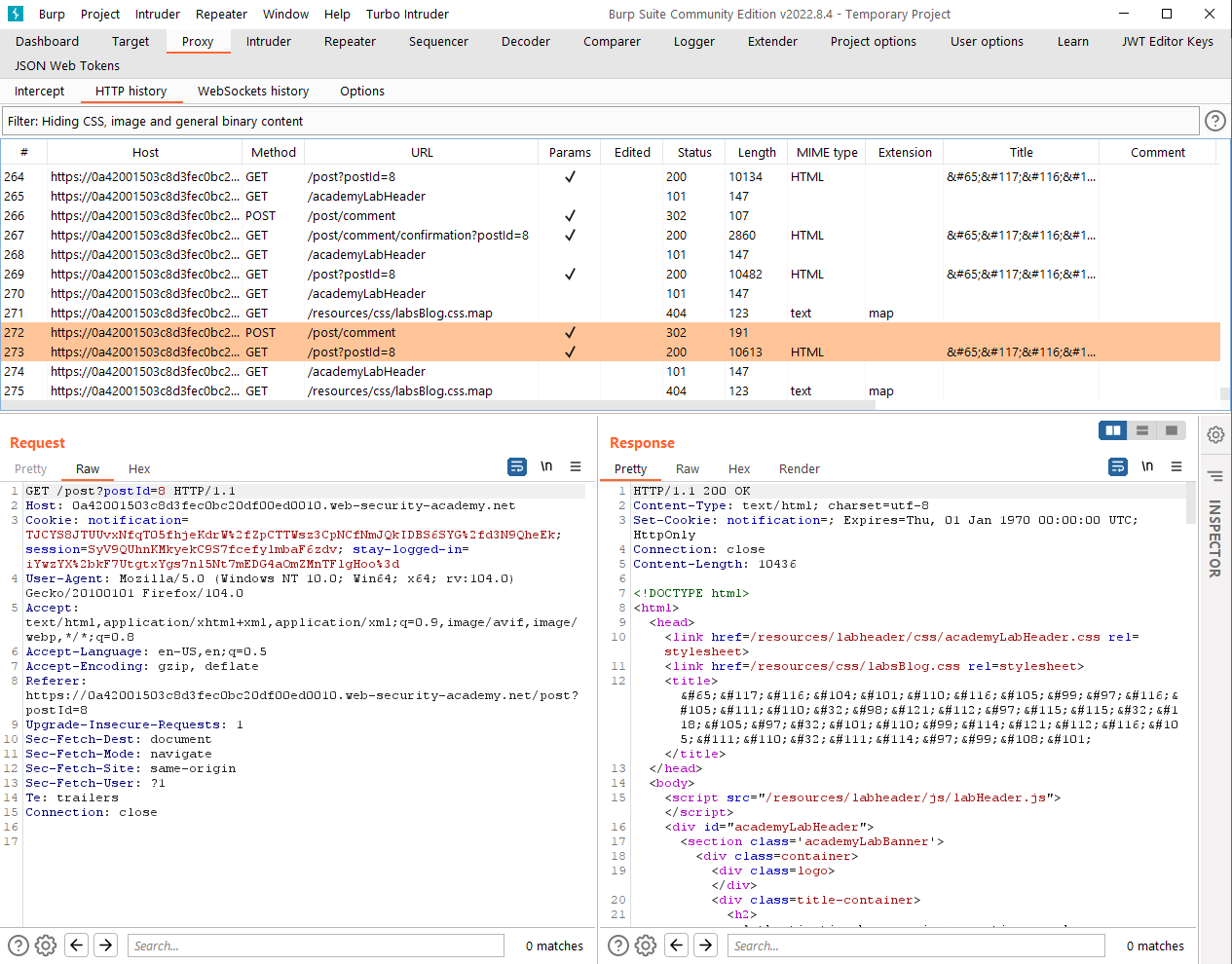
4-) After a few tries when I posted a comment on blogs and entered required parameters correctly everything was OK. But when I entered mail with different criteria application gives notification at top of the page. And also most importantly there is a newly created cookie which named NOTIFICATION which shows us some encrypted value in cookies.



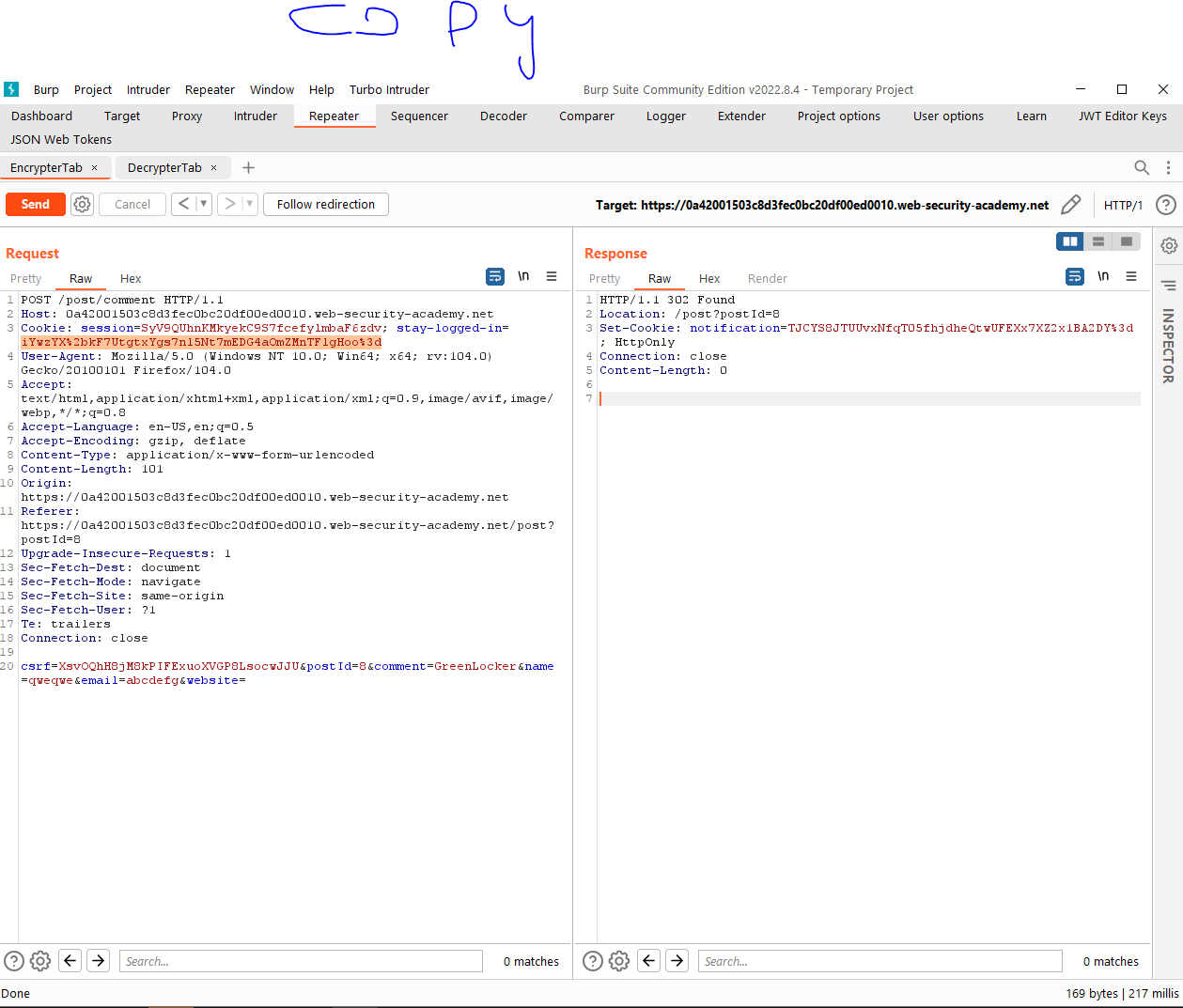


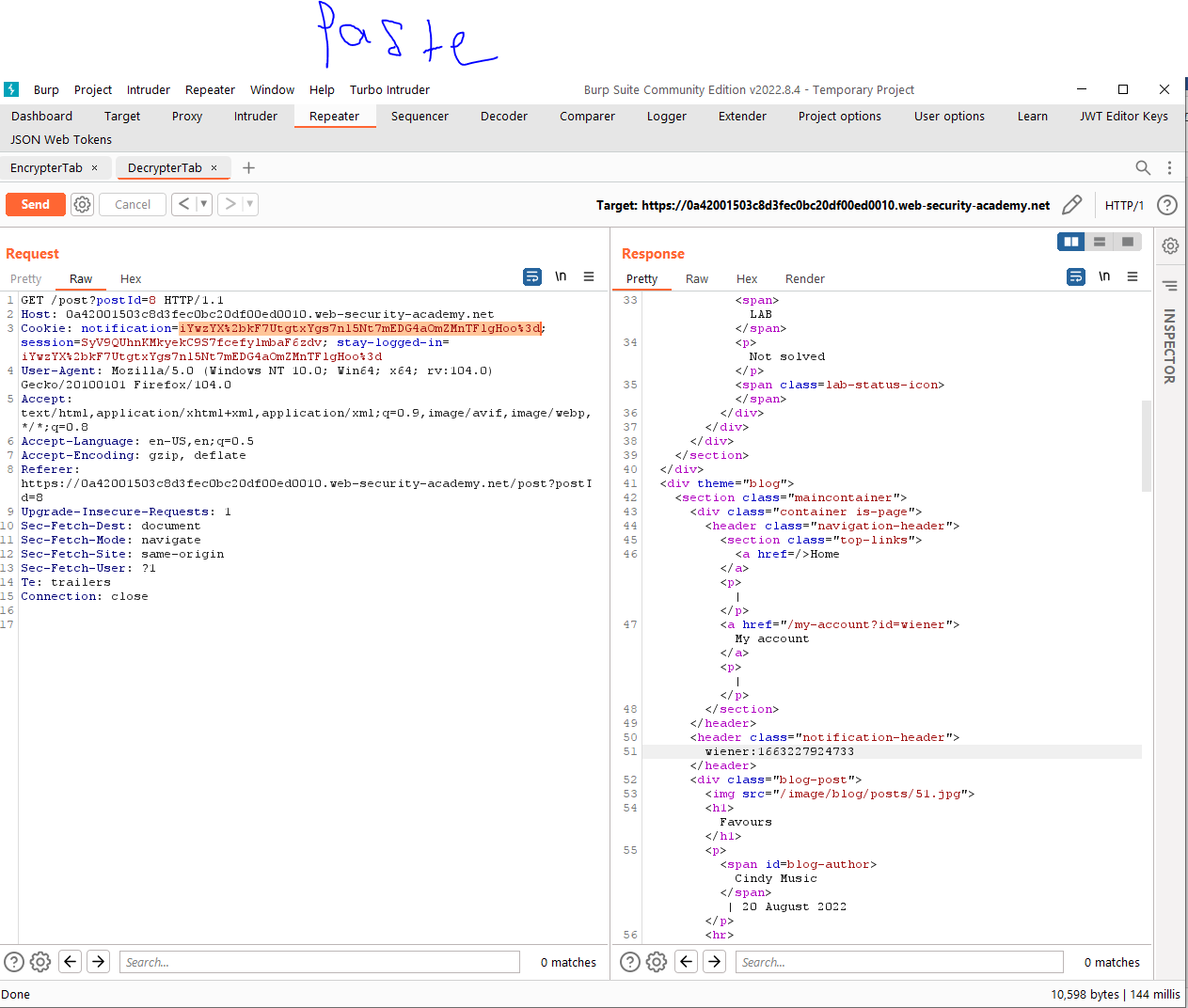


5-) After the encryption oracles has shown to us, I send both POST /comment request and GET /post?postId=8(id can varies) to the repeater to create an admin mailed account.

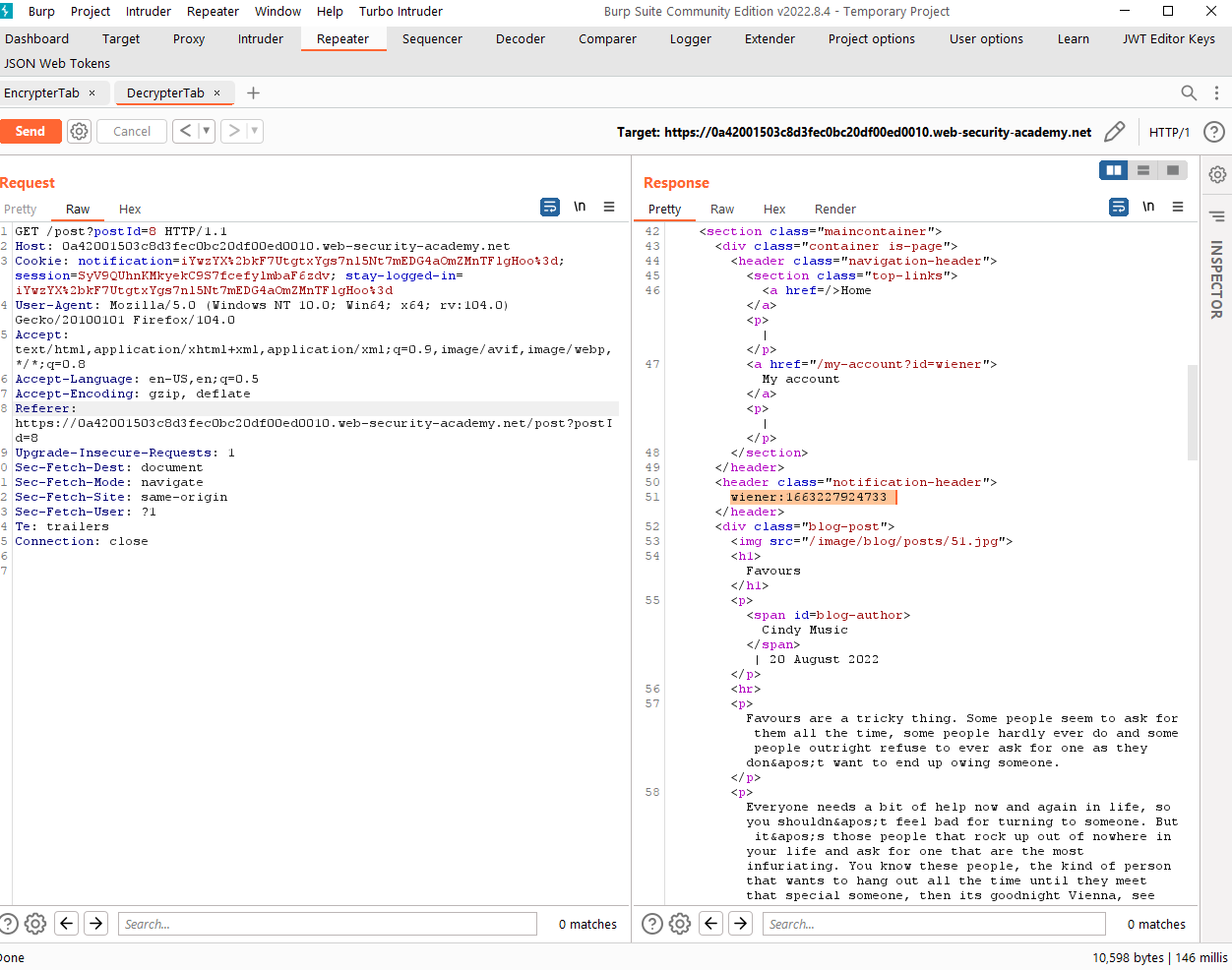


6-) Now, we have 2 information. Stay logged in cookies is our encryption oracle and notification cookie is it’s decrypter. When I enter values to stay logged in cookies notification gives us it’s decrypted values. So I want to be sure that this assuming is true I tried to decrypt the value in the stay logged in parameter at the notification parameter.

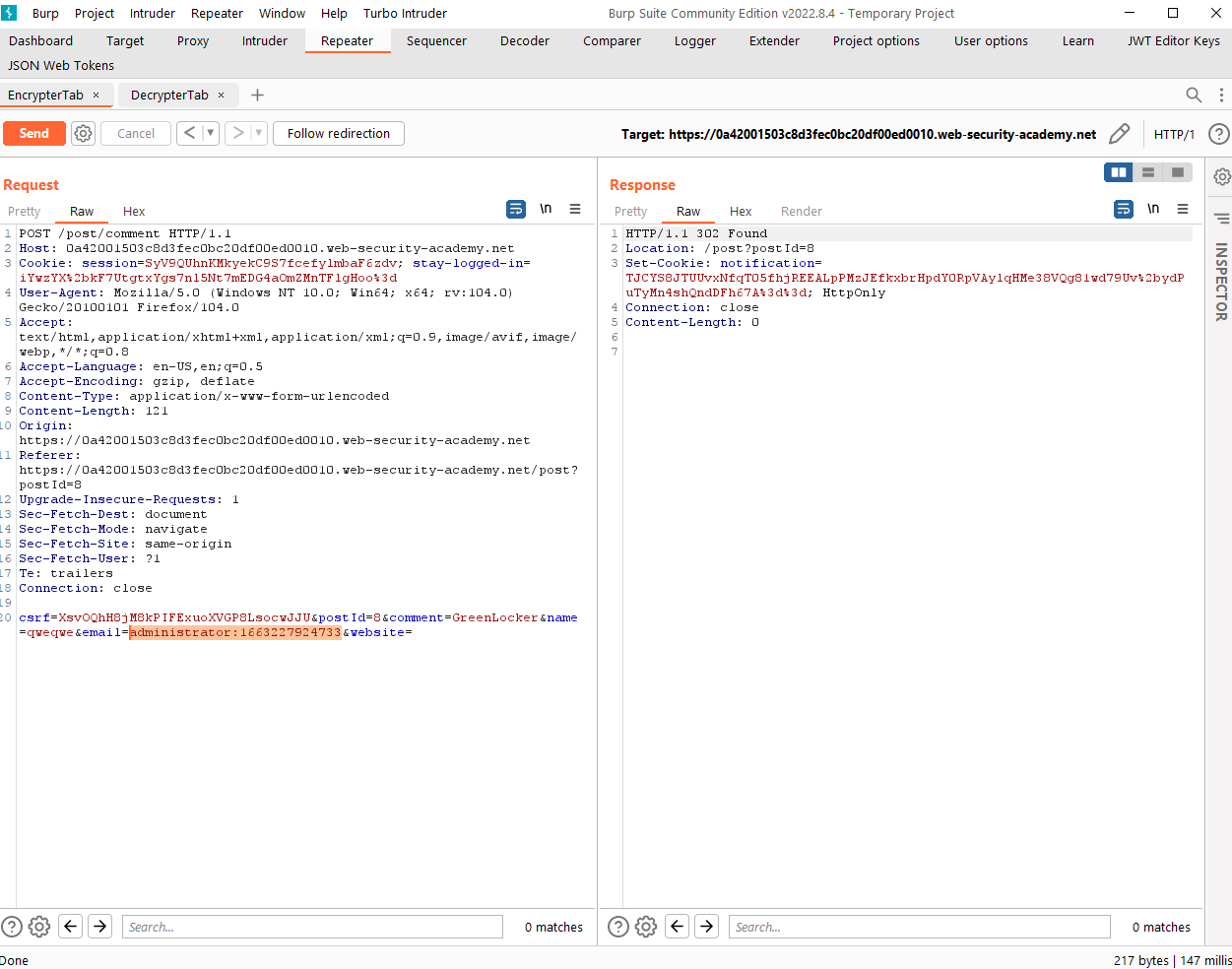




Then send the request and decrypted value is in front of us:



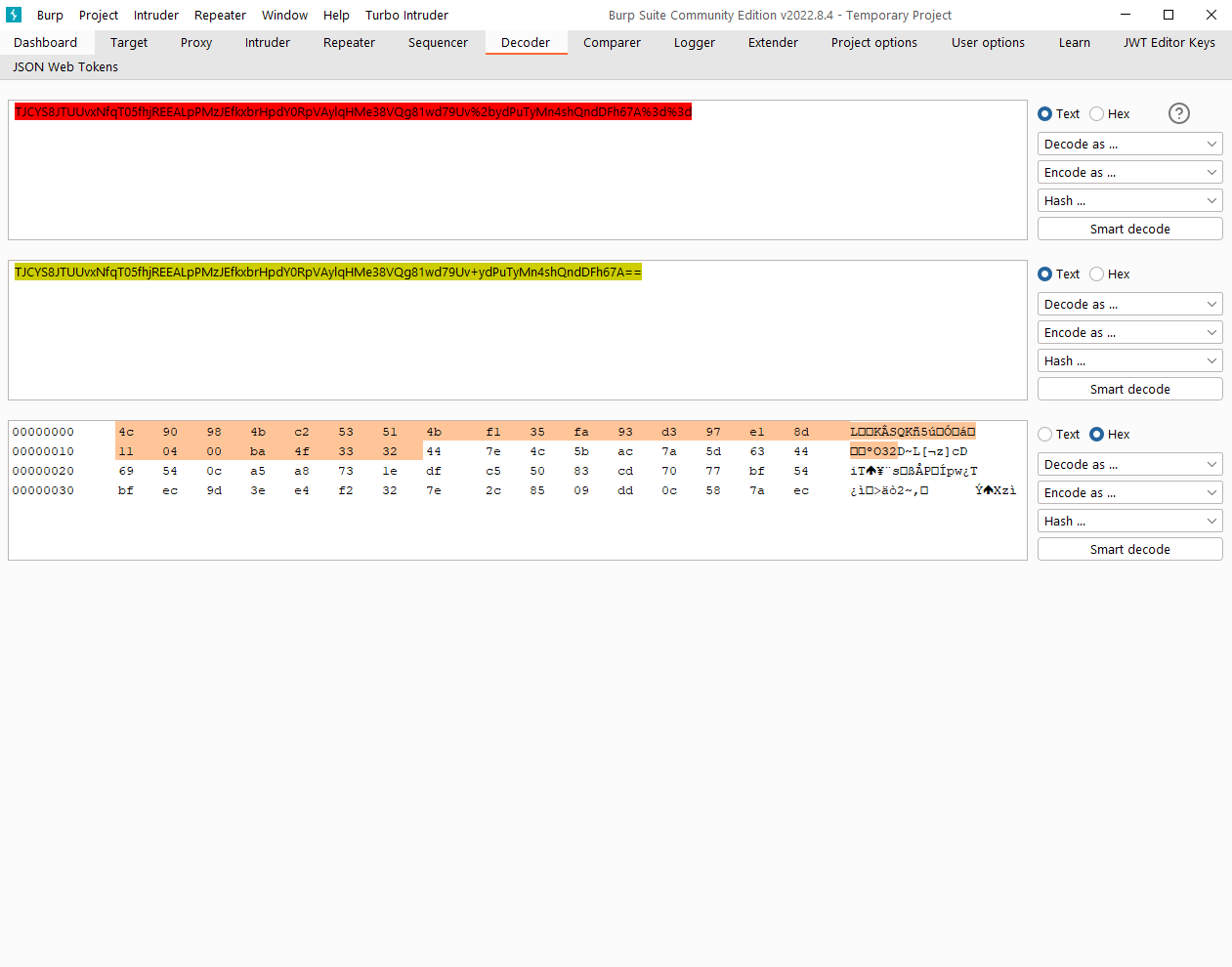
7-) After that we are copying the time stamp after wiener: “timestamp”. Because we want to make our account as admin not anybody else so we gonna use its timestamp. And lets write our mail as admin with our account created timestamp and after that copy the notification value in the response.

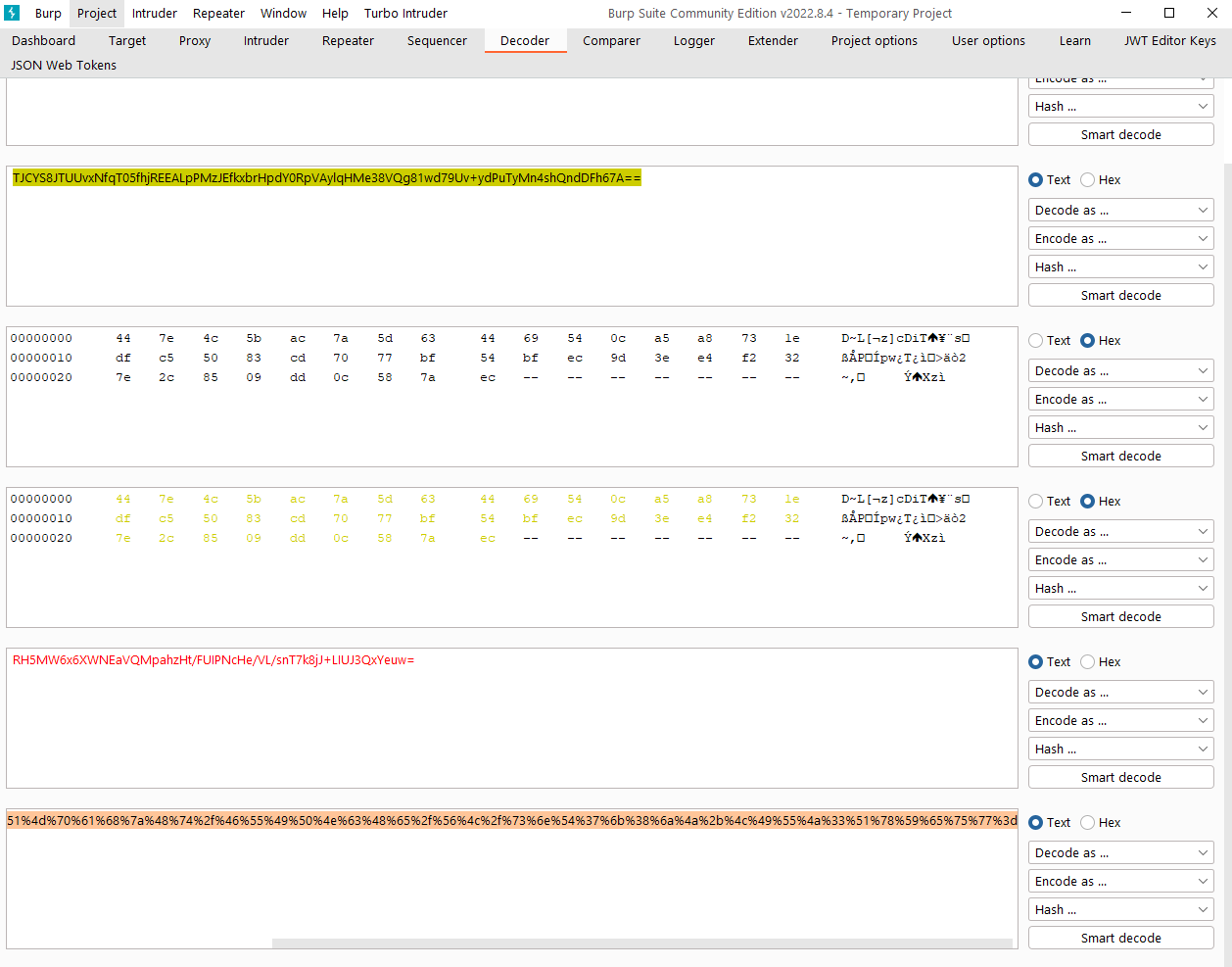


8-) Paste the notification value that we copied to the Notification cookie in the decrypter tab to see whats happening.

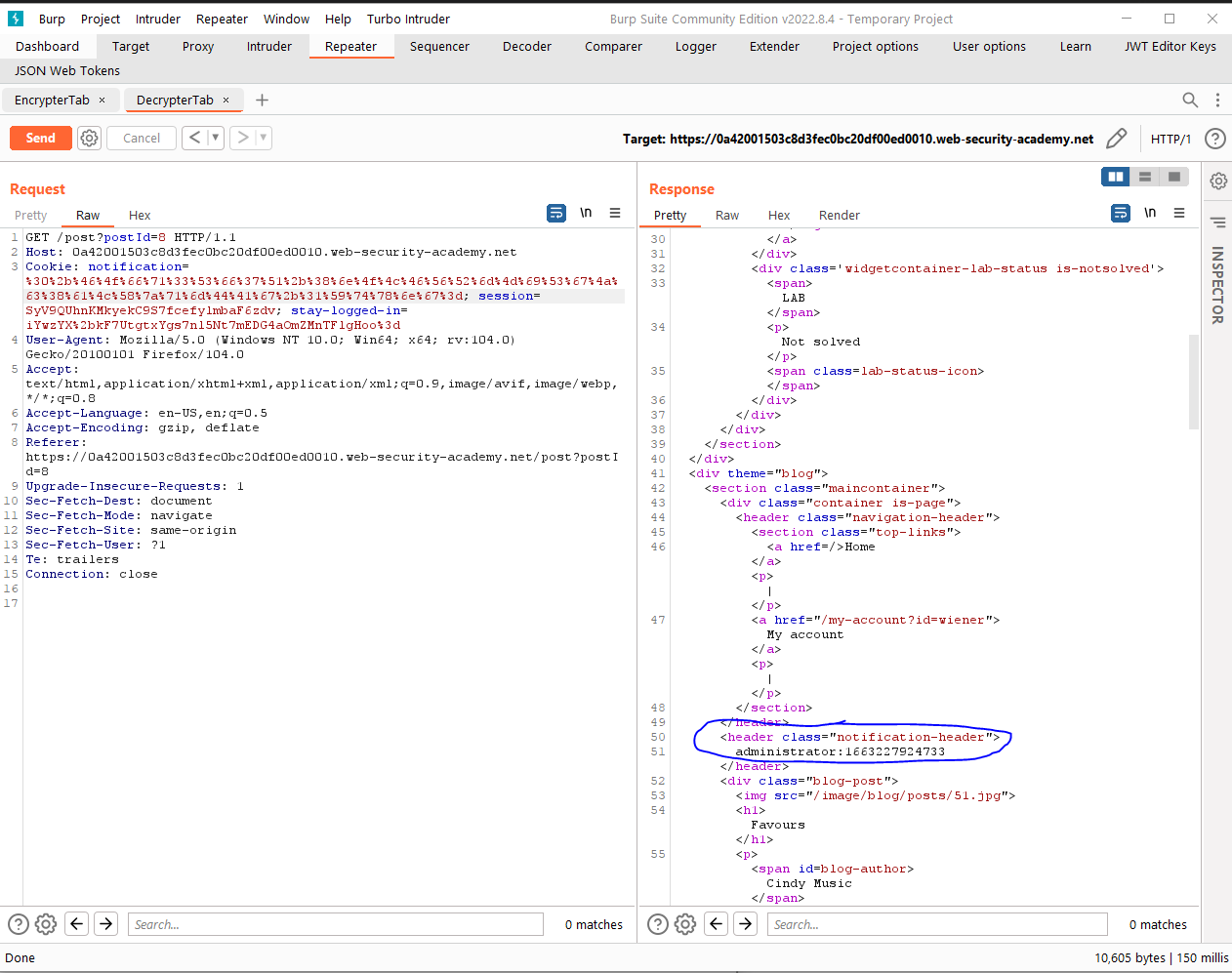


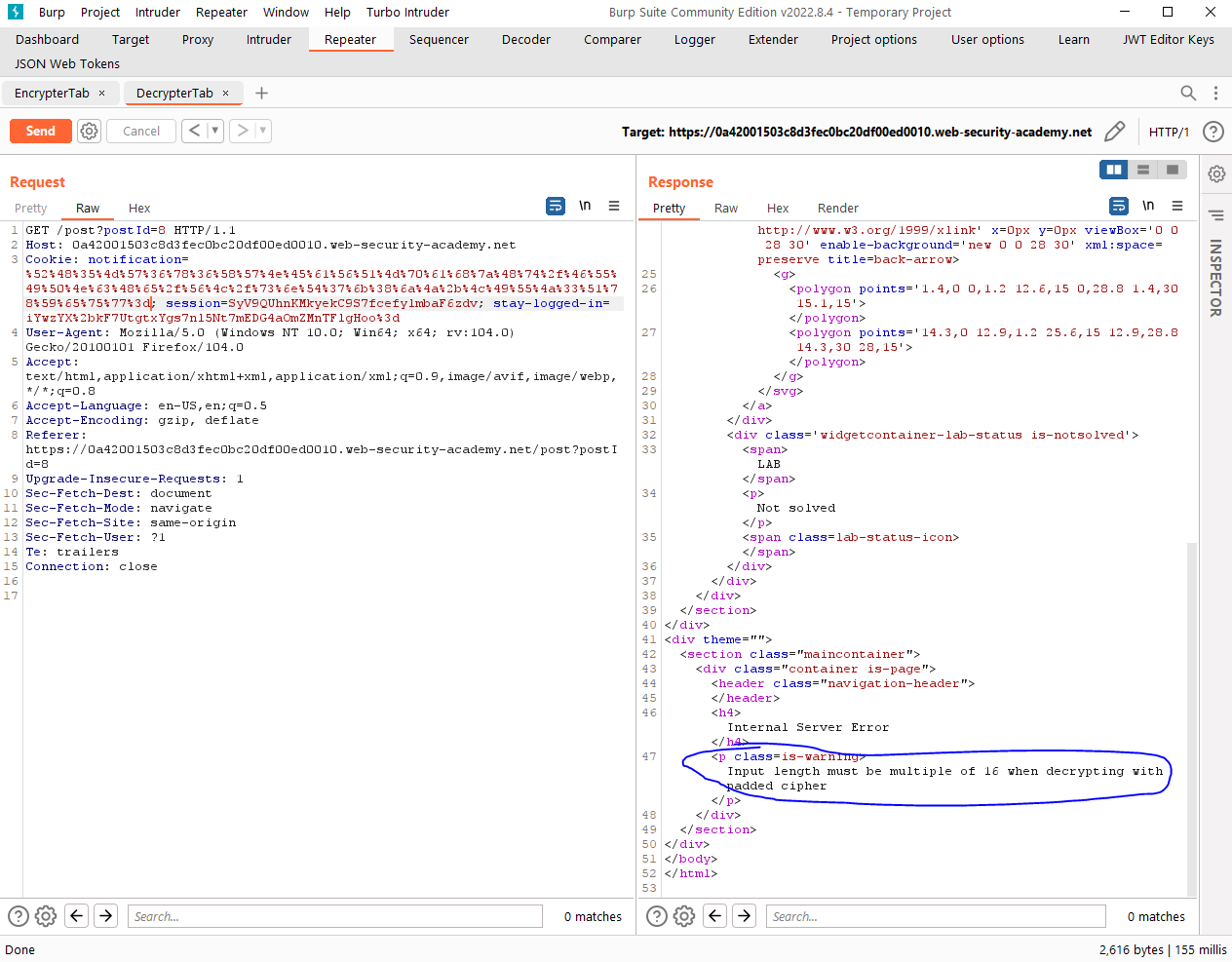
9-) App says Invalid mail address. So maybe we can try to delete the “Invalid email address: ” part and force the app to accept our mail as administrator:”timestamp”. This part is 23 bytes and we want to delete 23 bytes from notification value. Then I copied the notification value in encrypter and send it to the Burp Decoder. If we really want to find true decrypting algorithm in this step we will try so hard to find them. Really really so hard. After a few times I understand that they are using 2 step encryption **No:1 Base64, No2: Url-encoding.** I decoded our notification value reverse of this (no1: url no2: base64) ant try to delete first 23 bytes of the resulting value to delete “Invalid email address: ” part. I selected 23 bytes from decoded value and reencrypt it again.





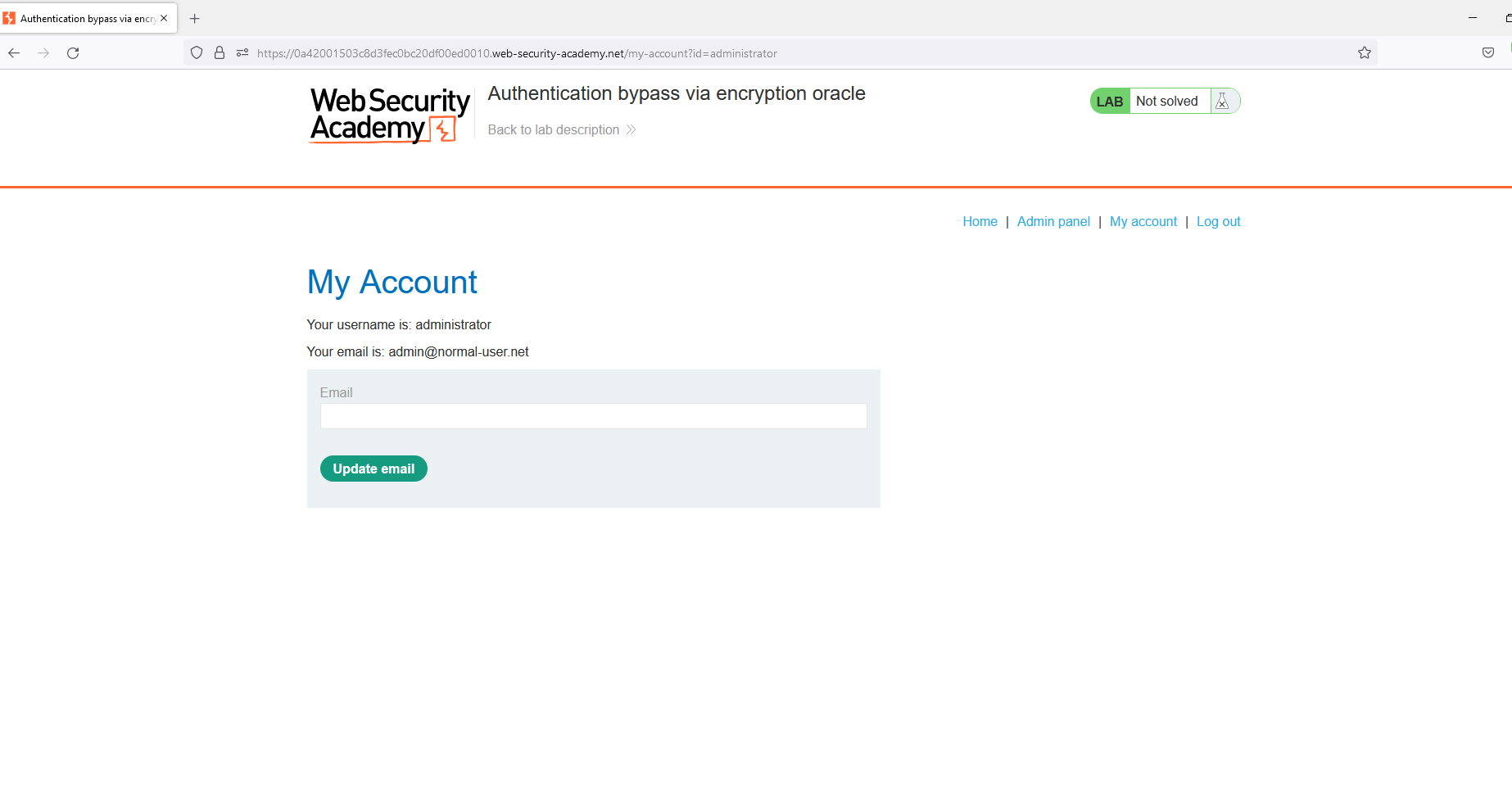
10-) When I try to enter the value that I get from de decrypter to the Decrpytertab Notification value on repeater app gives HTTP 500 Server Error. “Input length must be multiple of 16 when decrypting with padded cipher”. So I add how many xxx that is needed to the administrtor:”timestamp” to reach 32 char.

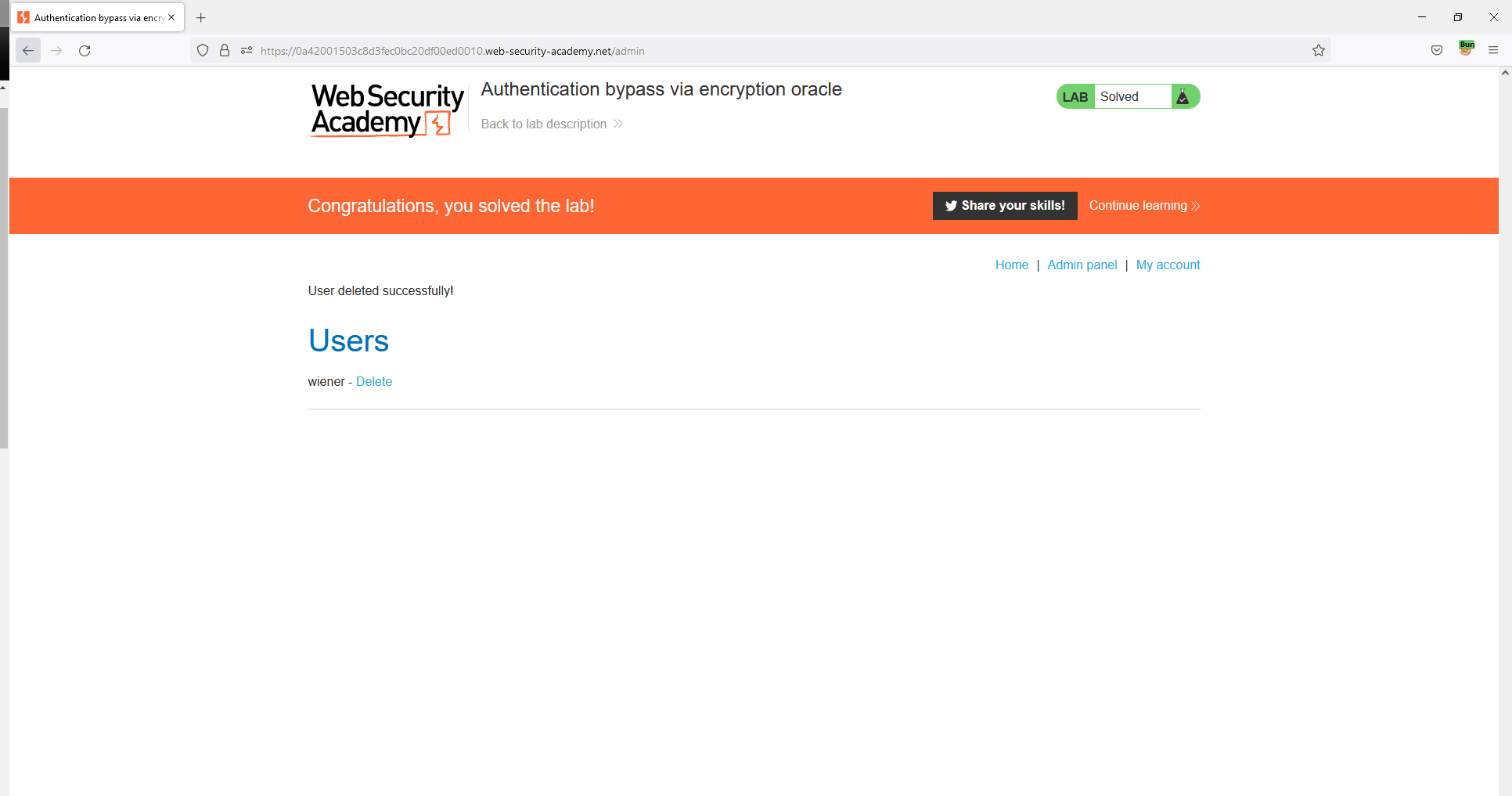




11-) After the process is done, capture my account request and delete session parameter and change stay logged in parameter as the value that we got from decrypter: %30%2b%46%4f%66%71%33%53%66%37%51%2b%38%6e%4f%4c%46%56%52%6d%4d%69%53%67%4a%63%38%61%4c%58%7a%71%6d%44%41%67%2b%31%59%74%78%6e%67%3d

Admin panel is reachable and delete the Carlos.





**How to prevent business logic vulnerabilities**

The keys to preventing business logic vulnerabilities are to:

* Make sure developers and testers understand the domain that the application serves
* Avoid making implicit assumptions about user behavior or the behavior of other parts of the application

You should identify what assumptions you have made about the server-side state and implement the necessary logic to verify that these assumptions are met. This includes making sure that the value of any input is sensible before proceeding.

It is also important to make sure that both developers and testers are able to fully understand these assumptions and how the application is supposed to react in different scenarios. This can help the team to spot logic flaws as early as possible. To facilitate this, the development team should adhere to the following best practices wherever possible:

* Maintain clear design documents and data flows for all transactions and workflows, noting any assumptions that are made at each stage.
* Write code as clearly as possible. If it's difficult to understand what is supposed to happen, it will be difficult to spot any logic flaws. Ideally, well-written code shouldn't need documentation to understand it. In unavoidably complex cases, producing clear documentation is crucial to ensure that other developers and testers know what assumptions are being made and exactly what the expected behavior is.
* Note any references to other code that uses each component. Think about any side-effects of these dependencies if a malicious party were to manipulate them in an unusual way.

Due to the relatively unique nature of many logic flaws, it is easy to brush them off as a one-time mistake due to human error and move on. However, as we've demonstrated, these flaws are often the result of bad practices in the initial phases of building the application. Analyzing why a logic flaw existed in the first place, and how it was missed by the team, can help you to spot weaknesses in your processes. By making minor adjustments, you can increase the likelihood that similar flaws will be cut off at the source or caught earlier in the development process.

REFERENCES

<https://portswigger.net/web-security/logic-flaws#:~:text=What%20are%20business%20logic%20vulnerabilities,to%20achieve%20a%20malicious%20goal>.

<https://portswigger.net/web-security/logic-flaws/examples>

<https://portswigger.net/web-security/all-labs#business-logic-vulnerabilities>