

Security Assessment

Airscrubber

Company name: CTouch

Date: 10.08.2020

Colophon

|  |  |
| --- | --- |
| **Label** | **Gegevens** |
| Client | Ctouch |
| Autor | The group |
| Status | Review |
| Classification | Vertrouwelijk |

Version history

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Changes** |
| 1.0 | 10-8-2020 | Setup and fill with content |
| 2.0 | 10-14-2020 | Finalizing |
| 3.0 | 10-14-2020 | Review and edit |

Review

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Reviewed by** |
| 1.0 | 14-8-2020 | Rick Theeuwes |

## 

**Chapter 1**

**Management Summary**

Ctouch has asked students of FHICT to perform a time-limited penetration test of the Airscrubber application. This has been done in our own test environment.

Our impression of the security level of the application is not very positive. The creator didn’t pay a lot of attention to the implementation of this application. In other words, we found a lot of vulnerabilities. There are differences in the vulnerabilities between high- and low CVSS scores.

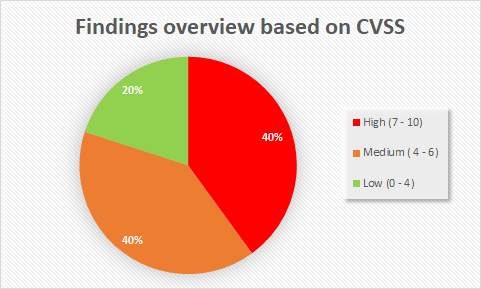
In the web application, there is a broken authentication vulnerability at which attackers - via a few actions - can give admin rights. Although in this application there is nothing to do with sensitive data, you can understand that it’s still a dangerous vulnerability.

Besides this, we want to mention the error handling that is (not) done in the application. This is a recurring problem that gives the attacker more attack vectors. For the completeness of the report, we want to mention it.

In this report you can find all the details of our research and findings, this also includes all technical advice.

1.1 Risks

The graph below shows the identified findings by each category of the CVSS:



1.2 Overview findings

The findings are sorted based on the CVSS classification. If more information is needed, it can be seen at [3.2 Analysis](#_vf3fa0h49e3n).

|  |  |
| --- | --- |
| **Finding** | **Risk** |
| Any session-valid user has all the possible rights like creating a user, location, and sensor. Also, there are no protected endpoints at all. |  |
| DOS is possible by sending a string to the Kafka server. |  |
| The passwords are not hashed when they are stored. |  |
| The Kafka server is not protected against outside requests. |  |
| Credentials in a .xml file & hardcoded credentials |  |

**Table of Contents**

[**Chapter 2**](#_9ebguro5h6st) **6**

[**2.1 Research question**](#_l5zynsppyp6s) **6**

[**2.2 Scope**](#_gm2gnfqavhfj) **6**

[2.2.1 Web applications](#_u2jmujnsg316) 6

[**3.1 Approach**](#_ubioixhhvus6) **7**

[3.1.1 General](#_m75qw8rytc2y) 7

[3.1.2 Web](#_kd6p7povjkss) 7

[**3.2 Analysis**](#_vf3fa0h49e3n) **7**

[**Chapter 4**](#_l9rj5ydqhqo4) **9**

[**4.1 Conclusion**](#_cfnbf0p1jr4l) **9**

[**4.2 Advice**](#_b282lo8radkt) **9**

[**Appendix A**](#_x9pd8qhgsqbk) **10**

[Broken authentication - roles](#_uahvwwt4ff7u) 10

[Bad error handling](#_17j0ms8pedv8) 11

[No hashes for password](#_lznh4b3z9n21) 11

[No Password Rules](#_s3joo2spizqk) 12

[Credentials in a .xml file & hardcoded credentials](#_nz92yau7vozr) 13

[Kafka server is not protected against outside requests](#_99w73j14bxcw) 13

[Input filtering](#_omu6uyziditq) 13

[DOS is possible by sending a string to the Kafka server](#_q1vd0ox9hmlr) 14

# Chapter 2

**Your request**

Farms and workshops need clean air to operate their businesses. So there can be a lot of stink or debris in the air, so there should be something to clean the air with. The Airscrubber fulfills this purpose: it takes the dirty air in, scrubs it, and clean air comes out.

The stakeholders of this project are Casper Schellekens, Mark Madsen, and Teade Punter; who represent FHICT and the Airscrubber project respectively. They are the contact to talk with if there are any questions and present our findings to them.

This project is used in a larger project, creating IoT guidelines. The findings from this pentest will be used to create guidelines to make safer and more secure IoT appliances.

# **2.1 Research question**

The application that is going to be tested is the Airscrubber software, which was made available via Gitlab. The goal is to try and find some general vulnerabilities and write down what the findings are. For example, there will be searched for *password storage*, *broken authentication*, *privilege escalation*, and other possible vulnerabilities.

# **2.2 Scope**

The scope of the project is the application on Gitlab and the sensors.

## **2.2.1 Web applications**

The web application that is running in the scope is the application itself and the database. Also the Kafka server and the sensors are in the scope of the project.

**Chapter 3**

Our Findings

# **3.1 Approach**

We put the application in docker and made it accessible, so we could pentest it. We followed the kill-chain:

1. Reconnaissance
2. Intrusion
3. Exploitation

There are more steps, but those aren’t relevant because we aren't going beyond exploitation since the application does not have any means to escalate. With these three steps, we were able to find vulnerabilities in the application.

## **3.1.1 General**

To test everything within the scope we did a form of scanning and exploitation after. Also we tried other variables with different inputs to “crash” the application (BurpSuite).

## **3.1.2 Web**

Testing was done on a server that was accessible to everyone in the group. The tools that were used are mostly the same tools we use in Kali Linux for other pentests. Among these are:

* Burp Suite
* Nmap
* Postman
* Security headers check

# **3.2 Analysis**

Below is a description of every problem found. Listed are the compromised systems, parameters of the exploits, cvss scores, and screenshots that prove the exploit. This list will be more technical than in the main document to provide a better understanding of what went wrong and what could be better.

1. **Broken authentication/authorization**

* If a user is logged in with a valid session, they can take that session and pretend they’re an admin. They can then create users, locations, and sensors. The endpoints aren’t protected, either. This means people can see what endpoints there are and what they return.

1. **Bad error handling**

* If there’s an error, the error messages will display too much information. Part of the code that regular users shouldn’t be able to see will be displayed. This is because the errors are handled badly. Hackers could take

1. **No hashes for password**

* Passwords are not hashed and are stored as plain text in the database. This is a vulnerability that makes it easy for a hacker to see the passwords if they gained access to the database.

1. **No Password rules**

* There aren’t any rules for the password when an account is created. This makes it possible to have a password with only one character or even no characters.

1. **Input filtering**

* There isn’t any input filtering and validation implemented in the application. This makes it vulnerable to SQL injection and XSS.

1. **Outdated software**

* There are outdated maven dependencies, which can lead to possible vulnerabilities.

1. **Weak JWT token**

* The JWT token is using a weak and guessable secret (e.g. “Very secret key”). This makes it possible to be brute-forced or guessed by the attacker

1. **Insecure Headers**

* The security headers are not configured properly, which makes them vulnerable.

1. **DOS attack**

* It is possible to attempt a Dos attack by sending a string to the Kafka server. This string will overload the server and cause services to stop.

1. **Outside requests**

* The Kafka server is not protected against outside requests. This means anyone can send requests

1. **Hardcoded Credentials**

* There are hardcoded credentials used in a .xml file. This means hackers could find and use these credentials for their own purposes.

## 

## **Chapter 4**

**Conclusion and advice**

The pentest concluded with several findings that show that the application was not entirely as secure as thought.

# **4.1 Conclusion**

The application has broken authentication; every user could access the admin page. Further, the server could not handle requests sent with a string. To solve this, there need to be more checks implemented that make sure the input sent to the server is legit. Furthermore, simple error handling could prevent unnecessary information from leaking to the user, which could lead to preventable attack vectors.

# **4.2 Advice**

The best advice that can be given to this project is to refactor it in its entirety. Errors need to be handled better, the input from the user needs to be filtered and the server needs to check if the requests it receives are legit coming from a user.

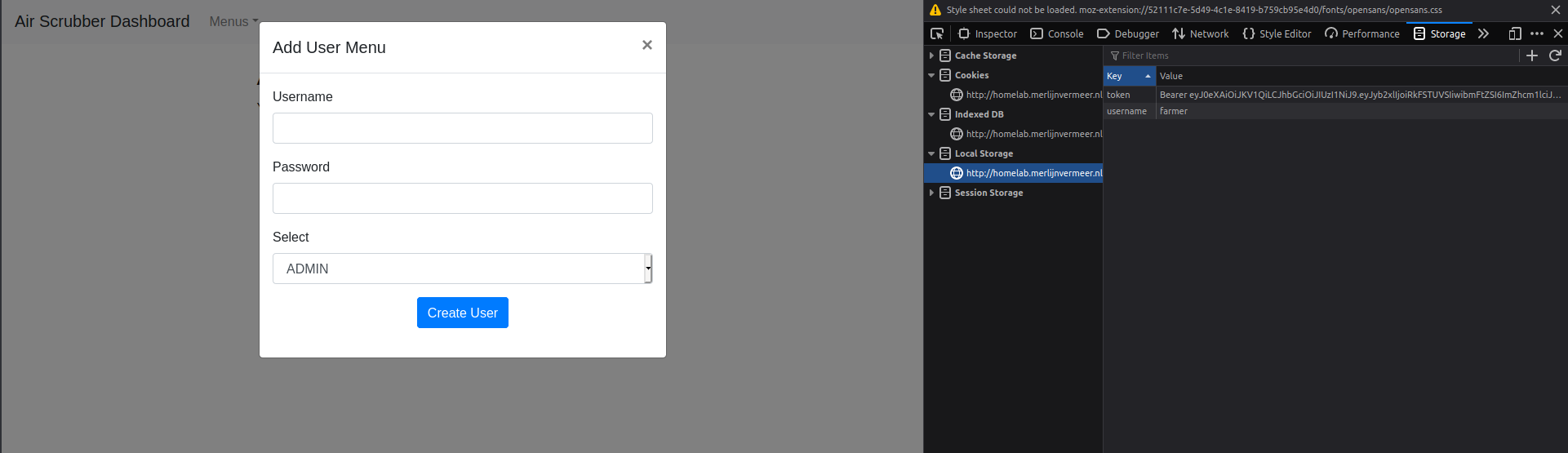
Regarding passwords, don't store it in a hardcoded XML file to be used in a production environment.

## **Appendix A**

**Technical Findings**

## Broken authentication - roles

**How to reproduce**

There is a vulnerability found with the authorization with the roles. The system cannot recognize if the specific user is admin or not. This leads to the problem that every user can access the admin dashboard. 

From the picture above it can be seen that if we log in as a normal user (In this case it is “farmer”) we can access the admin dashboard, simply by changing the path of the URL to {{site}}/admin. On the left side of the picture, it is shown that a normal user that is not an admin can create a new user and assign a role to it.

*NOTE*: Also there are no protected endpoints at all, so every user can access the endpoints! This is in combination with the vulnerability above high risk.

**Risks**

This exploit has a high risk of abuse. The normal user should not have access to the admin panel since he has no business there.

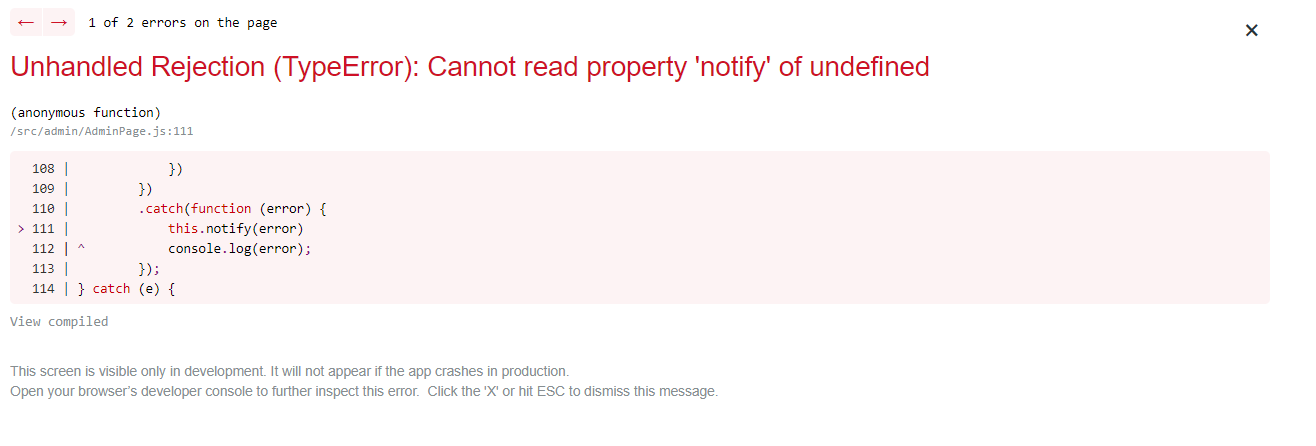
**Prevention**

Configure the JWT token to use policies and there should be a policy for each user role.

## Bad error handling

**How to reproduce**

Errors should be handled correctly. When an error happens, you don’t want the application to crash; instead, you want the program to handle it internally and continue in some way without showing too much detail to your user.



The picture above shows that there isn't proper error handling. This leads to showing information that shouldn’t be accessible by the user because an attacker can see what programming language is used and even can see fragments from the code.

**Prevention**

Make sure there are exception clauses, and exceptions should be as specific as possible. If an exception happens, make sure it describes the problem clearly in a log. Use a custom Error page on the frontend to show the user what went wrong.

**Risks**

Bad Error Handling to give attackers more information that could help them prepare an attack. The risks are usually low since the information leaked only adds to the overall picture for an attacker, but since it adds value that could be crucial to perform a successful attack, the risks CVSS score is low-high. This is also very inconvenient for the developers or users, since you cannot see what went wrong.

## No hashes for password

**How to reproduce**

The passwords should be hashed (e.g. with sha512). Otherwise, if they are stored in plain text, they can be easily used by a hacker to login and steal the account, if he gains access to the database or retrieves them from the browser.

**Risks**

If a hacker gets his hand on the password then he could easily login as any user, while the argument could be made that, since the attacker is already so far in the application that he could get his hands on the passwords, that it is already futile to hash them, it is still better to hash them so there is a possibility that the hash is not already known.

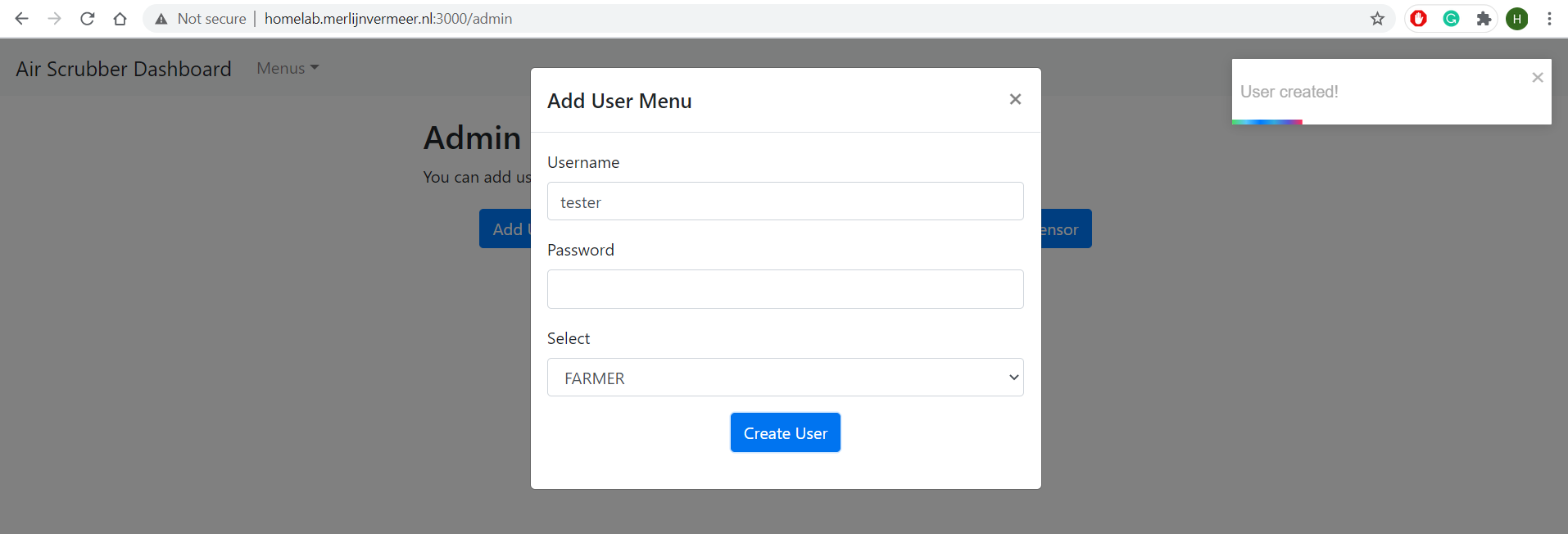
**Prevention**

The passwords should be hashed by creating two different entities of PasswordHash and PasswordSalt. Those two entities will be stored in the database and used from the server.

## No Password Rules

**How to reproduce**

All applications should demand their user to use a password that follows some basic rules to make it harder to be guessed or execute brute-forcing, or library attacks to their account.



The picture above shows the possibility of creating a user with a password containing no characters.

**Prevention**

Require the passwords to be at least 8 characters long including one capital letter, one normal letter, one digit, and one special character. This stops people from using default passwords such as admin or password because these are too easy to guess or hack.

## 

## Credentials in a .xml file & hardcoded credentials

**How to reproduce**

The credentials should be stored and used in a secure way without hardcoding them. The hardcoded passwords are stored in the code as plain text without any hashing or encryption. This makes them vulnerable and a hacker can easily access them. The picture below is an example:



**Risks**

If an attacker accesses the source code, he can easily find any hardcoded passwords and use them for further exploitation of the application and the system. He can gain access to and control of the system.

**Prevention**

Store the credentials as local environment parameters and if it is possible encrypt or hash them to obtain extra security.

## Kafka server is not protected against outside requests

**How to reproduce**

This can be reproduced if you are on the same network as the Kafka server. Then you can reach the server and send data to it. Not all data, as you can see in here: [DOS is possible by sending a string to the Kafka server](#_q1vd0ox9hmlr).

**Risks**

The risk of this finding is that every user can send data to the Kafka server. Also, attackers are able to send malicious data to it. This will increase the attack vectors which the attackers have.

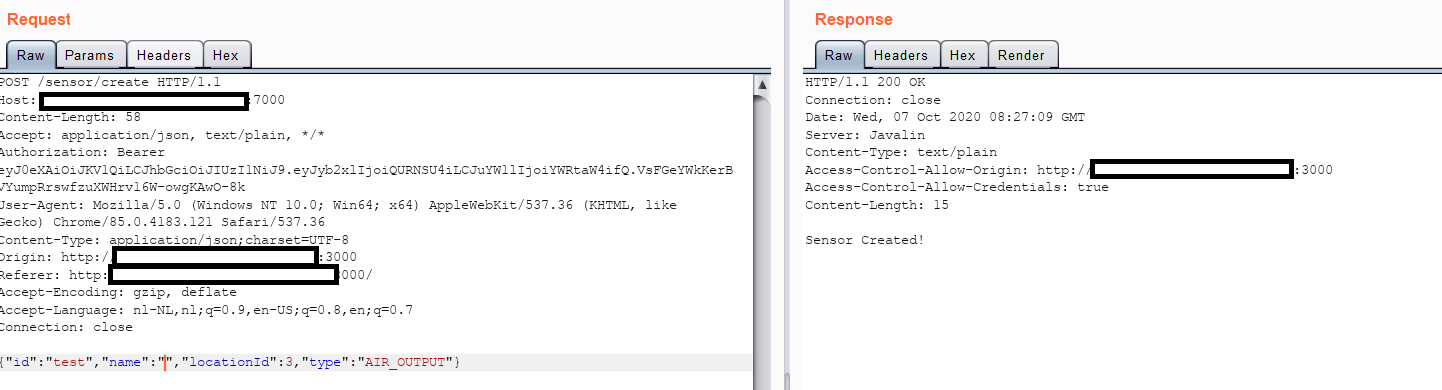
**Prevention**

To make sure not everyone can send (malicious) data to the Kafka server, there needs to be a form of authentication. So the server needs to know if you have the right to send data to it, otherwise, an attacker can abuse the Kafka server.

## Input filtering

**How to reproduce**

1. Go and create a sensor
2. Intercept the POST request and throw it in the repeater
3. Remove data of an attribute, in this example form the “name”.
4. Forward this request and you see the picture below

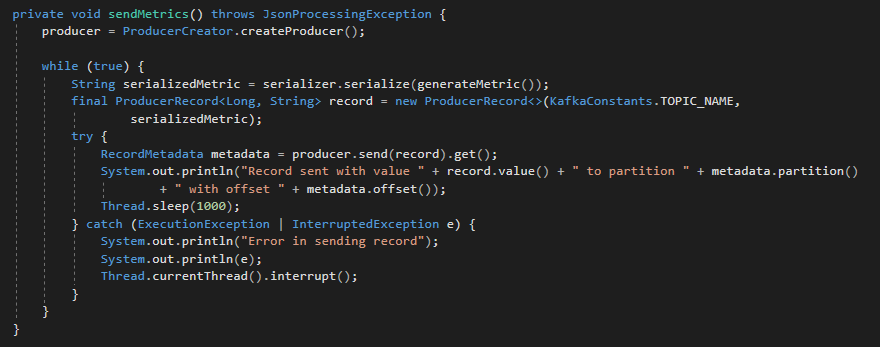
**Risks**

If there is no input filtering you can spam literally everything to the backend, like users without name or sensors without name.   
  
**Prevention**  
All user input should be filtered either by implementing an object relationship manager, for example, hibernate or by using regular expressions.

## DOS is possible by sending a string to the Kafka server

**How to reproduce**

If the Kafka server is open to the internet, everybody can send a packet to it and crash it. The server only accepts floats as data format, so if a string is sent instead of a float the whole server crashes.



**Risks**

The risks are very high since the denial of service directly impacts the ability of a company to work. If the services are down then the company is down.

**Prevention**

Check the incoming request on legit parameters before processing them. More authorization and authentication is needed to make sure only the sensors are the ones sending data to the server.