Cybersecurity project

Adam Genell
agenell@kth.se

Ellen Sigurðardóttir
ellensig@kth.se

Julia Wolak
juliawo@kth.se

18 October 2023

1 Introduction

We have implemented three different security countermeasures in NodeBB.

- **Database leakage and corruption** (mandatory): Ensuring that the database cannot be accessed remotely without going through NodeBB
- **Network eavesdropping**: Using a reverse proxy to encrypt traffic between the user and the server, making unencrypted traffic exist only locally
- **Unauthorized access**: Limiting login attempts to prevent brute-forcing of passwords

2 Problems

2.1 Mandatory – database leakage and corruption

Having remote access to your database can be beneficial, it offers increased accessibility, flexibility, and cost-effectiveness. Remote workers can access it from wherever there is an internet connection, which is great for distributed teams. This flexibility can help with maintenance, scaling, and updates on the database. For the same reason, you can even save money on setting up and maintaining the database infrastructure, you don't have to think about the hardware. You can have a backup or a replica of your database, if something corrupts the data or the server goes down, you can quickly switch to the backup.

But those benefits and many more can come with a significant cost. Allowing remote access to a database can come with a significant security and operational risk. It increases the risk of unauthorised individuals gaining access to your data. It could lead to data breaches; hackers can manipulate or even delete your data. The performance of the database can also be slower due to network latency and a significant amount of bandwidth consumption.

When we were looking for ways to prevent access to the database unless it was through NodeBB, we ended up in the installation notes for MongoDB, the

database. There we found:

> By default, MongoDB launches with `bindIp` set to `127.0.0.1`, which binds to the localhost network interface. This means that the `mongod` can only accept connections from clients that are running on the same machine. Remote clients will not be able to connect to the `mongod`, and the `mongod` will not be able to initialise a replica set unless this value is set to a valid network interface.

To double check this was default, we checked the configuration file for MongoDB, /usr/local/etc/mongod.conf on MacOS, and saw that `bindIp: 127.0.0.1` was already set. As the installation notes said, that means that we can now only access the database from our local machine, no remote access is allowed. We added `port: 27017` to the file.

To test this, we used two computers. Computer A had the MongoDB database, and computer B had MongoDB Compass, a GUI for MongoDB. There, you can query and analyse your data visually. First, we changed the `bindIp` in the mongod configuration file to `0.0.0.0`, so everyone would be able to access it remotely, and made sure there wasn't a firewall set up, and restarted the service to make sure the changes were applied. Then we tried to connect to the database via MongoDB Compass using the IP address of computer A and the port connected to mongod, 27017, but that didn't work. Then we tried to use the admin username and password that we set up when we set up the database, but that didn't work either. After some trial and error and googling, we decided to ditch MongoDB Compass and get MongoSH, the shell for MongoDB. We downloaded it andn we made sure to add mongosh.exe to path. Then we opened the Command Prompt and typed `mongosh "mongodb://username:passw@192.168.1.6:27017"` and we got back:

```
C:\Users\Ellen>mongosh "mongodb://username:passw@192.168.1.6:27017"
Current Mongosh Log ID: 652ea5ae83223559b772d8dd
Connecting to:  mongodb://<credentials>@192.168.1.6:27017/?directConne ⌋
↪  ction=true&appName=mongosh+2.0.2
Using MongoDB:  7.0.0
Using Mongosh:  2.0.2
```

which means that we were able to connect to the database.

To test that binding the IP address to 127.0.0.1, or localhost, would refuse the remote connection, we change it back in the mongod configuration file. We restarted the service and tried again. That time we got back:

```
C:\Users\Ellen>mongosh "mongodb://username:passw@192.168.1.6:27017"
Current Mongosh Log ID: 652ea5ea14dd8595960fd7f8
Connecting to:  mongodb://<credentials>@192.168.1.6:27017/?directConne ⌋
↪  ction=true&appName=mongosh+2.0.2
MongoNetworkError: connect ECONNREFUSED 192.168.1.6:27017
```

which means that the connection was refused, we were not able to remotly access the database.

In conclusion, ensure that both `bindIp: 127.0.0.1` and `port: 27017` are in the mongod configuration file under `net`.

2.2 Network eavesdropping

By default, NodeBB listens on port 4567 and expects unencrypted HTTP traffic. There are two problems with this. The first is a minor inconvenience. Port 80 is the agreed on default port for HTTP traffic, so a user wanting to access a service listening on a different port must excplicitly specify the port. The second is a security problem. Unencrypted traffic is vulnerable to man-in-the-middle (MITM) attacks, meaning user accounts and admin accounts may be compromised.

Using nginx, we set up a HTTPS server to act as a reverse proxy for NodeBB, solving both the inconvenience problem and the security problem at the same time. The reverse proxy listens on ports 80 and 443, which are the default ports for HTTP traffic and HTTPS traffic respectively. Any requests received on port 80 are redirected to the HTTPS server, which redirects traffic back and forth to NodeBB using the `proxy_pass` configuration option.

Traditionally, to acquire SSL/TLS certificates, one has to pay a company *(certificate authority)* to issue one. These certificates are valid for a set period of time, after which one has to pay the company to issue a renewed certificate. Another option is to create a self-signed certificate. This is secure enough for most purposes, but there is no chain of trust established. As such most web browsers will display a warning when trying to access a website with a self-signed certificate.

A newer way of obtaining certificates is using a free service to issue them. We used certbot with the certificate authority Let's Encrypt. By running the certbot program, it asks Let's Encrypt to issue a certificate. Let's Encrypt then sends us the information required for certbot to create some certificate files on the filesystem.

Until recently, Let's Encrypt would generate 2048-bit RSA-type certificates by default. In version 2.0.0 of certbot, the default certificate type changed to ECDSA. ECDSA is a more secure encryption algorithm in general, but in case an RSA-type certificate is required, this can be specified by running certbot with the `--key-type rsa` flag.

Once the certificate files had been created, we combined nginx configuration options from the NodeBB reverse proxy documentation and Mozilla's SSL Configuration Generator to create the configuration files. They can be found in `network_eavesdropping/` in this repository. To install this on a new machine or using a different domain name, the `server_name` option and the options referencing certificate files should be changed.

The configuration options significant for TLS are those beginning with `ssl_`. `ssl_certificate` is the "full chain" for the certificate, with `ssl_certificate_key` being the private key. For extra security, we only allow TLS v1.3 by setting the option `ssl_protocols`. There are a few different versions of SSL/TLS, with TLS v1.3, officially released in 2018, being the newest and most secure.

In nginx, there is a configuration option `ssl_prefer_server_ciphers` which restricts the permitted TLS cipher suites. In general, it's more secure to turn this option on, because some ciphers may be less secure than others. However, in TLS v1.3, the selection of ciphers is already quite limited, so in this case we choose to allow the client to use whichever cipher they prefer.

The certificate generated by Let's Encrypt is valid for 3 months. To renew it, running `certbot renew` will place new certificates in the same place as before so there will be no need to update any configuration files. To automatically renew, a utility such as Cron can be used to schedule `certbot renew` to run at a certain interval.

At the end of the nginx configuration file is the configuration for the reverse proxy. The primary option to accomplish this is `proxy_pass`, but there are also a number of HTTP headers as specified in the NodeBB documentation. These headers allow NodeBB to work as intended even when running behind the reverse proxy.

2.3 Unauthorized access

Unauthorized access refers to unauthorized individuals or automated systems attempting to gain access to a system, application, or service without proper authentication. This problem is often aggravated by brute force attacks, where attackers systematically try different combinations of usernames and passwords, often taken from a variety of lists of common passwords that are available online. These attackers typically use automated scripts or even botnets, which are a group of devices connected to the Internet. They relentlessly attempt to gain access, posing a severe security threat. Those kinds of attacks may have various impacts. There are three main ones that come to mind:

1. **Data compromise**: Successful unauthorized access can lead to a data breach where confidential information is compromised. Attackers may gain access to users' personal data, financial records, confidential business data or restricted information.
2. **Violation of privacy**: Users' privacy is significantly compromised when their accounts are accessed without consent. Personal information, contact history and other sensitive data can be exposed and misused.
3. **DoS**: Brute force attacks can consume significant system resources. These resource-intensive attacks may lead to a denial of service, rendering the

system temporarily or permanently inaccessible to legitimate users. System performance can also be significantly affected.

To mitigate this security threat, it is crucial to implement measures to limit failed login attempts. By controlling the number of login attempts, the system can effectively stop the efforts of attackers, protecting user data and the overall security and availability of the system.

As a countermeasure to this problem, two solutions were chosen:

1. Built-in feature in NodeBB
2. Rate limiter in nginx

The combination of using a rate limiter in nginx and implementing the built-in feature in NodeBB to block users after a certain number of failed login attempts is an effective approach to mitigate unauthorized access due to brute force attacks. While nginx rate limiting serves as an efficient, network-level defense mechanism, NodeBB's built-in account lockout is a user-centric, application-level protection feature. These two solutions work in synergy, with nginx offering broad protection against mass attacks and NodeBB focusing on securing individual accounts within the application. By combining these measures, we can create a comprehensive security strategy that defends against brute force attacks while preserving system availability and user data integrity.

2.3.1 Built-in feature in NodeBB

After installing NodeBB and all the required components according to the documentation, we create the admin account and log in. From the main page, in the upper menu we click on the *Admin* icon, that will take us to the admin panel. Now we click on *Settings*, and from the drop-down menu we choose *Users*. We go to the *Account Protection* area, and here we will see two variables: *Login attempts per hour* and *Account Lockout Duration (minutes)*. We set them both to five. Why? For a legitimate user five attempts is enough to login considering any misspellings, it's also enough attempts to know that it's better to click *Forgotten password?* then try to guess more. But in case a legitimate user locks an account, it's only five minutes, which doesn't compromise user availability at a high level (taking into consideration it's a forum). This feature also allows us to choose after how many attempts and for how long users can be eventually locked.

2.3.2 Rate limiter in nginx

Similar to the first solution we install NodeBB and all the required components according to the documentation. In this solution however we will work with the configuration file of nginx. We go to the folder where we have nginx installed, and then in the `conf` folder, there will be the configuration file `nginx.conf`. We have to edit this file. In the `http{}` block, we write the line

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=5r/s;
```

`limit_req_zone` defines a shared memory zone named `mylimit` that will store rate limiting information.

`$binary_remote_addr` is the variable that captures the binary representation of the client's IP address.

`zone=mylimit:10m` specifies that the `mylimit` zone should have a size of 10 megabytes. This space is used to store data related to rate limiting.

`rate=5r/s` defines the maximum allowed request rate for this zone. In this case, it is set at 5 requests per second (r/s). Requests exceeding this rate will trigger rate limiting.

Then we scroll to find the `server{}` block (within the `http{}` block), and for the `location{}` block we write

```
location /login {
    limit_req zone=mylimit burst=10;
}
```

`location /login { ... }` defines a location that applies the rate limiting rules only to the requests made to the /login path.

`limit_req` is used within the `/login` location block to enable rate limiting based on the `mylimit` zone previously defined. It limits the rate of incoming requests for this specific location.

`zone=mylimit` specifies which zone to use for rate limiting. In this case, it's the mylimit zone defined earlier.

`burst=10` allows for a certain number of requests to be accepted at short intervals before a rate limit is enforced. In this configuration, it allows a burst of 10 requests. After the burst limit is reached, any further requests will be forced into rate limiting.

After saving the configuration file, we need to restart nginx, so that it can be applied. This is done by running `nginx -s reload`.

3 Contributions

3.1 Adam Genell

The first thing I did was to fork the NodeBB repository and add the other group members as collaborators. However as we read more about the project we realized that we would not need to modify the actual source code, so I created a new empty branch called `countermeasures`.

I have installed NodeBB and its dependencies on an Oracle Cloud free-tier server (running Ubuntu). Due to it being free-tier, it has rather limited resources and initially I was not able to install all of the dependencies due to the installer running out of memory. I solved the issue by creating a swap file and decreasing the memory limit for Node.js.

After successfully installing NodeBB, I started doing the necessary work to implement the *network eavesdropping* countermeasure. I already had an overall idea on how to do this, and started by installing nginx and certbot from the package manager. I looked into setting up the reverse proxy, and found the documentation for doing so on the NodeBB documentation website.

After that it was a straightforward process of using certbot to generate certificates and adding the appropriate nginx configuration options. When everything was finished I added the configuration files to the Git repository.

3.2 Ellen Sigurðardóttir

I started out by cloning the repository that we had made from the NodeBB official repository. I had to add my existing SSH key so I would be able to clone it. When I had cloned it, I followed the installation documentation for NodeBB. Since my computer is new, I had to begin by installing NodeJS and Homebrew. I followed the instructions from their respective websites.

In the installation documentation for NodeBB they recommend using Redis as the database, so I did that first. Then I realised that we had decided to use MongoDB, I stopped going down that route and installed MongoDB using Homebrew in the terminal. I followed both the macOS and Ubuntu instructions when I was setting it up. I then started MongoDB as a macOS service and continued setting up NodeBB. When that was all set up, I went to localhost:4567 (NodeBB's default port) and I started exploring NodeBB.

I worked on the mandatory part, *database leakage and corruption*, where I wanted to prevent any remote access to the database we were working with. This was my first time working with MongoDB, so I spent some time going over how it works.

When I was starting out, I wasn't sure what the best way was implement the prevention. After some research I quickly realised that the best way to this is to bind the IP address to localhost or 127.0.0.1.

3.3 Julia Wolak

First thing I did was to install NodeBB and all the required components, following documentation provided on NodeBB website, on how to run NodeBB on Windows. Then I familiarize myself with the system and what NodeBB is.

Then I started researching possible solutions for the Unauthorized access problem. I did find some plugins, but some of them were very outdated, or were providing solutions that could not be very user-friendly, for example reCaptcha plugin. I decided that we shouldn't use it, since I think everyone can agree that current reCaptcha mechanisms are very annoying to the actual user. Furthermore, nowadays there are tools that are quite successful in bypassing reCaptcha, which have an even higher chance in resulting in nothing else than compromising availability for the user.

After that, I found a built-in feature in NodeBB and I decided that I will combine it with nginx Rate Limiter, which could result in a defense-in-depth approach. I created accounts and changed the settings. My next step was to change a configuration file of nginx and restart everything.

As it was my first time working with MongoDB, nginx and Node.js I've spent more time getting to know all the components.