Cybersecurity Overview - Project

DD2391 HT23/ 18 October 2023 19:00

Group 12
Alperen Guncan
Ka Ho Kao
Pawat Songkhopanit
Melissa Julsing

Link to git: https://github.com/To5BG/NodeBB

The imposed requirements do not allow for a thorough report with all configurations and explanations needed. Therefore, the repository contains **all** the information needed to replicate the setups. The repository is structured as follows:

- The master is a direct fork of the origin repository https://github.com/NodeBB/NodeBB
- The master branch's README.md contains all the information to navigate in the repository:
  - Contains group number and authors
  - Contains report (what you are reading right now)
  - Each issue is resolved in a separate branch, along with example configuration files, and a **separate thorough writeup** for the specific problem, named **writeup.md**. It is also set to be the README.md for the specific branch
  - Both the branch name and the folder name that contains all these things are specified as markdown blocks below each problem
  - Summary of how each problem was addressed

Table of Contents

Database leakage and corruption (Mandatory)

The default configuration of the database and NodeBB may be insecure (see the installation notes). We would like to prevent any remote access to the database that is not mediated by NodeBB software.
A more thorough writeup that 2 pages cannot account for, can be found in the repo!!

Threats

By default, if no "bind" configuration directive is specified, Redis listens for connections from all available network interfaces on the host machine. This means that Redis is configured to accept connections from any IP address and any network interface on the system. While this configuration provides flexibility for clients to connect to Redis from different sources, it can pose security risks such as unauthorized access, data exposure, and injection attacks.

In addition, by default, the Redis server does not have an authentication set up, which may lead to every comment being executed without any verification. Therefore, it does not allow you to manage a system with roles and track events. An impactful threat is, for example, that an evil user could modify or delete the database with one comment. With no authentication, it is also not possible to take track of who is doing what to the server.

Another threat is that NodeBB itself does not have a built-in firewall. A firewall is a network security device or software that monitors and controls incoming and outgoing network packages based on predetermined security rules. It acts as an agent between two networks. Without having a firewall, the server is vulnerable for example unauthorized access and DDoS attacks.

Chosen countermeasure

Solution 1: Redis restrictive Configuration

Bind_address
In Redis, it is possible to specify the network interface or IP address on which Redis will listen for the incoming access connection. By setting the `bind_address` configuration option, we can control which network interfaces or IP addresses are allowed to connect to our Redis server. This configuration restricts the remote access of the network and only allows local access. This also helps isolate Redis from other services or applications running on the same server. By setting `bind_address` to `127.0.0.1 ::1`, the Redis server will allow access from the local machine only.

redis.conf:
```
bind 127.0.0.1 ::1
```

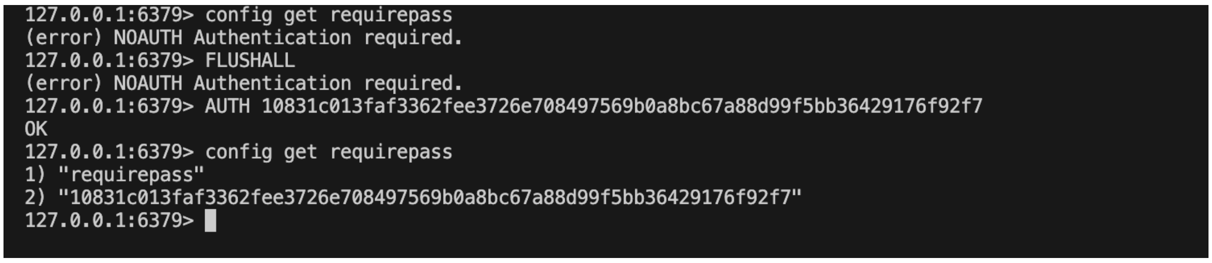
Authentication on the Redis server
When the `requirepass` setting or password setting is enabled on Redis, redis will refuse any query by unauthenticated clients. Redis provides an Access Control List (ACL) system that enables control over privileges and permission for the redis-server and redis-cli. With the `ACL genpass` command, we can generate a password that is strong and reliable since the password is generated using SHA256 and HMAC-SHA256 to ensure the randomness of the password. The 256 bits (64 hex characters) ensure the randomness of the password which will be hard for attackers to guess or solve for the password. To configure the `requirepass` parameter, we use the command `config set requirepass [randomstring]` to secure Redis behind a 64-byte-long password.

redis.conf file:

```
requirepass 10831c013faf3362fee3726e708497569b0a8bc67a88d99f5bb36429176f92f7
```

To test the configuration, we use the command `FLUSHALL` to see whether the Redis would allow us to flush all the data without the authentication first.

Redis-cli settings:



```
127.0.0.1:6379> config get requirepass
(error) NOAUTH Authentication required.
127.0.0.1:6379> FLUSHALL
(error) NOAUTH Authentication required.
127.0.0.1:6379> AUTH 10831c013faf3362fee3726e708497569b0a8bc67a88d99f5bb36429176f92f7
OK
127.0.0.1:6379> config get requirepass
1) "requirepass"
2) "10831c013faf3362fee3726e708497569b0a8bc67a88d99f5bb36429176f92f7"
127.0.0.1:6379> █
```

Solution 2: Firewall for NodeBB

By default, NodeBB runs on port 4567. We want to configure our firewall to ensure that only port 4567 is listened to. We configured a software firewall at the location where the server is hosted, to control traffic to and from the NodeBB. Since the NodeBB was running on MacOS we used "pfctl" which is a command-line interface for managing and interacting with the Packet Filter (PF) firewall. First, we created the redis.pf file and added the configuration line.
Redis.pf file;

```
rdr pass inet proto tcp from any to any port -> 127.0.0.1 port 4567
```

With this rule, only matching incoming TCP (Transmission Control Protocol) packets on the internal interface are allowed to be passed (pass) through the firewall. When going to any other port besides http://localhost:4567/ the website won't be accessible. When we used a different port such as port 80, we noticed that unlike before, the website was now also not visible in the browser.

Difficulties

The authentication service for the Redis server is not secure enough. It currently relies on a single password for all users, which is not ideal. To enhance the security, we need to implement an authentication service that has multiple users and bind the actions of those users to the logging services or give users different privileges for optimal security instead.

A further improvement to consider in the future is configuring the `bind address` or firewall settings in alignment with our actual application setup. For example, we need to set the bind address to the real address we use to host our server. We also need to take into account the specific firewall configuration for the device on which we host the server.

Unauthorized access

Users may try to brute force passwords, possibly using a botnet. We would like to limit failed login attempts without compromising availability.
A more thorough writeup that 2 pages cannot account for, can be found in the repo!!

Threats

The default implementation of NodeBB does not limit the amount of login requests that a user can do. The only way that it addresses this issue is by implementing a temporary account lockdown after a certain number of failed login attempts. However, there are a few issues in the way it is implemented, including that there are no guarantees for proper async behavior, non-configurable variables, and the quick reset of the login attempts.

Not addressing login attempts and/or rates is clearly a big security issue, since a user can simply try a large amount of passwords, whether with a dictionary, combinator, or pure brute-force attack, to gain access to accounts he is not supposed to have access to. Such an attack has the additional complexity that it can congest the network of the server, slowing down all other processes, including countermeasures to the said issue! This is why one of our solutions makes use of a third-party service whose operation is decoupled from the server's.

Chosen countermeasure

The application has a lockdown functionality built-in, which is a good start for addressing this issue. However, we should ensure that there are no workarounds, specifically the ones described above. We make use of Nginx to ensure the amount of requests is manageable. As an extra security measure, we also introduce a Captcha solution. It is encouraged that both are used in conjunction for maximal security.

Solution 1: Nginx rate and connection limiting

Nginx's main purpose as a third-party service is the one of a load balancer. The robustness of its implementation and the long list of developers backing this product are other good reasons why one would want to go with Nginx over implementing rate limiting in the server from scratch. It provides a lot of flexibility in how developers can limit both the number of requests and connections - some filter options include subpath, location, requester data, header information, and cache/memory location. In our case, we want to limit the amount of requests and connections to the location /login on an IP basis. Below is an example configuration:

```
limit_req_zone $binary_remote_addr zone=one:10m rate=5r/m;
limit_conn_zone $binary_remote_addr zone=addr:10m;

server: {
    listen              80;
    server_name         localhost;
```

```
  location / {
    proxy_set_header    X-Real-IP $remote_addr;
    proxy_set_header    X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header    X-Forwarded-Proto $scheme;
    proxy_set_header    Host $http_host;
    proxy_set_header    X-NginX-Proxy true;
    proxy_pass          http://127.0.0.1:4567;
    proxy_redirect      off;
    # Socket.IO Support
    proxy_http_version  1.1;
    proxy_set_header    Upgrade $http_upgrade;
    proxy_set_header    Connection "upgrade";
    location  /login {
      limit_req zone=one  burst=5 nodelay;
      limit_conn          addr  2;
      proxy_pass          http://127.0.0.1:4567;
    }
  }
}
```

Basically, we copy the basic configuration for a reverse proxy, and then we make use of nested location directives to rate-limit /login specifically. Here we limit the endpoint to 5 requests per minute, grouped by IP, and to two connections per IP (justified as 1 connection per method, the methods being GET and POST). The problem states the issue of logging in credentials specifically, so the config is left at that. The developers are free to add more directives, with separate caches and/or rate specifications as needed.

Solution 2: Use of captchas

Restricting traffic to the server, when accompanied by the account lockdown functionality, is often enough to handle most flavors of bruteforce attacks. However, in a situation where the botnets are quite sophisticated in taking great care in their attempts, or if the overall capacity of the server is small compared to what the botnets can hoard, rate limiting may not be a sufficient solution. As such, the team also took their time to implement a solution that makes use of **Captchas**. A separate PHP server is hosted, whose endpoints are routed by Nginx, sharing same domain as NodeBB server. Full configuration, design process, and configurations *are in the repo*.

Difficulties and considerations

One future consideration for developers extending the pool of locations that should be under rate limits - it is preferable if the location list is extended with a regex or an OR directive than it is to duplicate the location directive already present. This is due to both brevity and performance reasons.

Network eavesdropping

HTTP traffic is not encrypted. We would like to use SSL to protect the confidentiality of posts that are submitted or accessed by the users.
A more thorough writeup that 2 pages cannot account for, can be found in the repo!!

Threats

The default implementation of NodeBB uses regular HTTP for establishing communication between the web client and server. This protocol is not acceptable for mainstream usage across an insecure channel like the internet. The main reason for this is that HTTP does not encrypt its payloads, which leads to *personal information leakages*. In other words, a man-in-the-middle attacker can simply sniff all the traffic, and all of a sudden they have a clear look on what the client is doing. This also includes any passwords and sensitive/private information. As such, if we plan to deploy this application to the Internet with real users, *a secure protocol like TLS needs to be used*.

Chosen countermeasure

There are two main groups of solutions that can address this issue - one is to upgrade the internal connections to use TLS, and the other is to pipe all the connections to a secure service, which is another way of describing a third-party reverse proxy. It would use regular HTTP to connect to the server but connect to the clients with a secure protocol. The group was given one solution per category and weighted the merit of each. For the descriptions below, and for the demonstration, *self-signed certificates* are used. This is addressed below.

Solution 1: Built-in upgrade

NodeBB uses Express.js for middleware and the bundled *http* and *https* Node.js modules to start the server. As such, given that the source code is properly engineered, all it takes is to ensure an *https* server is initiated. After analyzing the source code, one can see that the application attempts to read the key-value pair *'ssl'* in *config.json*, and if it finds it, it will boot with an *https* server. Therefore, the solution only requires that the user adds in the config

```
"ssl": {
   "key": "<KEY-PATH>",
   "cert": "<CERTIFICATE-PATH>"
}
```

This solution is much cleaner and requires no extra infrastructure, and therefore less attack surface. Yet, the developers of NodeBB do not recommend this solution, and opt for client developers to use a proxy instead. The reason for this is unclear - not justified in the project's README either. The group's guess is that the application does not have built-in support for beneficial features, which a proxy like Nginx can provide.

Solution 2: Reverse proxy

A simple, ubiquitous service that is being used for reverse-proxying is Nginx. We run the application with its default configuration (*http,* port 4567), and then set Nginx to reverse-proxy from the server to port 443. Below is an example configuration:

```
server: {
   listen                   443 ssl;
   server_name          localhost;
   ssl_certificate        <CERTIFICATE>.pem;
   ssl_certificate_key  <KEY>.pem;
   ssl_protocols          TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
   ssl_ciphers            HIGH:!aNULL:!MD5;
   location / {
      proxy_set_header     X-Real-IP $remote_addr;
      proxy_set_header     X-Forwarded-For $proxy_add_x_forwarded_for;
      proxy_set_header     X-Forwarded-Proto $scheme;
      proxy_set_header     Host $http_host;
      proxy_set_header     X-NginX-Proxy true;
      proxy_pass             http://127.0.0.1:4567;
      proxy_redirect         off;
      # Socket.IO Support
      proxy_http_version  1.1;
      proxy_set_header     Upgrade $http_upgrade;
      proxy_set_header     Connection “upgrade”;
   }
}
```

Furthermore, to maximize security, one has to configure his/her own *ipconfig* table to block all non-local traffic toward the NodeBB server. This makes sure that clients can access the web server only through the proxy, by extension only through HTTPS.

This solution requires much more configuration and requires that the developers also have Nginx running. Conversely, the presence of more infrastructure could introduce more vulnerabilities to hackers, especially sniffers on the proxy service. However, this solution moves all the load balancing and traffic encryption away from the application, and into a third-party trusted service, which also comes with additional neat features.

Difficulties and considerations

Configuration is quite straightforward. An important consideration, however, is that the certificates used here were *self-signed*. This is problematic - if a certificate has no authority or in other words, if it is not signed by trusted third parties, it is almost not any more secure than not using HTTPS at all. This is because a MITM attacker can forge two pairs of certificates of his own, send them to the client and server, and act as an intermediary, allowing him to still decrypt and read all the traffic. For this demonstration, however, self-signed signatures were deemed sufficient to show the idea. For a real application, the certificates would need to be signed by a trusted party - one free option is *Certbot*.

Contribution

Alperen Guncan

- Set up Git repository
  - Write master's README.md
  - Decide on the problem submission structure
  - Decide on branch structure
  - Prepare branches for each problem
    - Standardize problem solution structure
    - symlink for README.md writeups
- Completed 'Network eavesdropping' **in its entirety**, including:
  - Nginx configuration solution
    - Set up reverse-proxy
    - Upgrade connection with TLS
    - Optimize config
  - SSL config.json solution
    - Look through the source code
    - Expand on config.json
  - Add reference/example configurations under folder *ssl*
  - Create the writeup.md, which contains the justifications, threats, solutions, considerations, and configuration steps *thoroughly*
  - Add problem entry to report (what you are reading right now)
- Completed 'Unauthorized access' **in its entirety**, including:
  - Nginx configuration solution
    - Set up reverse-proxy
    - Limit request and connection rate with caches
    - Modify source code to account for 503 errors
  - Captcha solution
    - Research good captcha options
    - Integrate the assets into the repo
    - Modify the existing code to accommodate for the captchas
  - Add reference/example configurations under the folder *login-limiter*
  - Create the writeup.md, which contains the justifications, threats, solutions, considerations, and configuration steps *thoroughly*
  - Add problem entry to report (what you are reading right now)
- Proofread the report, all writeups separately, and test all solutions

Ka Ho Kao

- discuss what problems we are going to tackle and provide some suggestions to each problem
- setup NodeBB on my own computer
- For problem “unauthorized access”
  - work on modifying the “login.js” script
  - work to disable the username and password field after three attempts together with Melissa
  - work to disable the login button after three attempts together with Melissa
  - try to set a timer to re-enable the users to login after a certain of time

  However, the solutions have not be adopted since it is not sufficient against BotNet

- work on the nginx implementation and configuration together with Melissa

- For problem “database leakage and corruption”
  - discuss the solutions of solving the problem
  - find ways and work together for password generation
  - re-implement, test and refine the solutions of redis restrictive configuration and firewall

  > Report:
  - Write partly about the chosen countermeasures in the report
  - write the part of “Bind address”, “Authentication on the Redis server” and “Firewall for NodeBB” in master’s readme file
  - write the writeup.md in the github

- Proved read the report

Melissa Julsing

- Setup Google Drive document with an outline of the project requirements
- Setup NodeBB and server on my computer
- For problem ‘Unauthorized access’
    - Create branch limit-login
    - Exploring possibilities to limit failed login attempts
    - Worked with Ka Ho to disable the login fields and login butts after the password was entered wrongly three times -> not sufficiently against BotNet and therefore not used
    - Together with Pawat exploring the nginx definition and possibilities
    - Together with Ka Ho working on the implementation and configuration of nginx
    - Together with Pawat testing the nginx login rate
    - Pushing Nginx files to the branch
- For problem ‘database leakage and corruption’
    - Create branch leakage-db
    - Together with Pawat working and configuration file for Redis
    - Together with Pawat changing on the bind and requiqepass settings
    - Together with Pawat testing with get and set auth in redis-cli. Ultimate test with database kill comment.
    - Together with Pawat working on setting up the pftcl and specifying the rule.
    - Together with Pawat testing the firewall rule
    - Pushing Redis and pftcl files to the branch
    - Report: together with Pawat decided on outline for writing the Unauthorized access part of the report
    - Report: pasting in the made screenshots
    - Report: writing the Treads paragraph
    - Report: writing the “Solution 2: Firewall for NodeBB“ paragraph

Pawat Songkhopanit

- Decided and discussed which problem to tackle for this project
- Helped set up the nginx, pftcl, and configuration file for Redis
- Worked on Database leakage and Corruption
    - Came up with both of the solutions
    - Set up the configuration file for the `bind_address` parameter together with Melissa
    - Set up the configuration file for the `requirepass` parameter together with Melissa
    - Verified that the configuration for the `bind_address` works
    - Verified that the configuration for the `requirepass` works
    - Wrote the solution for setting up a firewall on a macOS device
    - Wrote the configuration file for the firewall setup
    - Wrote partly about the threat in the report
    - Wrote partly about the chosen countermeasures in the report
    - Wrote the difficulties in the report
- Worked on some part of limit-login
    - Set up and figured out the solution of limiting the login. This includes:
        - Set up the configuration file for nginx
        - Tried and tested the configuration for nginx
- Proved read the report