Security Report

April 14 2018

SEng 468 : Spring 2018

Group G12

Heather Cape V00795197
Geoff Lorne V00802043
Tanner Zinck V00801442

Table of Contents

1.0 Introduction

This project was an exercise in making effective trade offs based on the six analysis dimensions discussed in class. In order to be able to finish the course with a working project we decided to build a system that satisfied the business requirements (ie. the Minimum Viable Product) as the main goal, followed by optimizations to increase the TPS of the system. In terms of the analyses: Functional and Structural were the most important as the MVP was built up, Distribution and Performance were the next focus as we worked on increasing the TPS, and Reliability and Security were the tradeoffs.

This report will discuss the assumptions of security based on the project description. It will outline what security practices were upheld in the architecture and in the technology stack chosen. Finally, a critical evaluation of our system will outline areas of improvement for each aspect of the architecture.

One of the definitions given in class of privacy and security was the “ability to enforce system-wide policies and procedures, including audit trails of what happened[1]”. based on this definition this report will be discussing protocols used in web system, along with our audit system. Further discussion will be around prevention larger known vulnerabilities that these technologies are susceptible to, such as SQL injections in a database.

2.0 Assumptions

The largest assumption we made in the development of our day trading system was that any user sending us a command has already been authenticated by some other hypothetical sign in service, meaning we did not implement any form of authentication. Since our environment is a locked down, closed network computer lab, we also assumed that we would not be actively attacked over the network. Because of this, we did not worry about disallowing outside communications, or implement any system for blocking problematic IPs etc.

[1] Slide 27/235 Distributed Systems Software, January - April 2018, Stephen W. Neville

3.0 Overall Architecture

In lecture, general distribution architecture was discussed. Five examples of architectures and their benefits and drawbacks were discussed.[2]

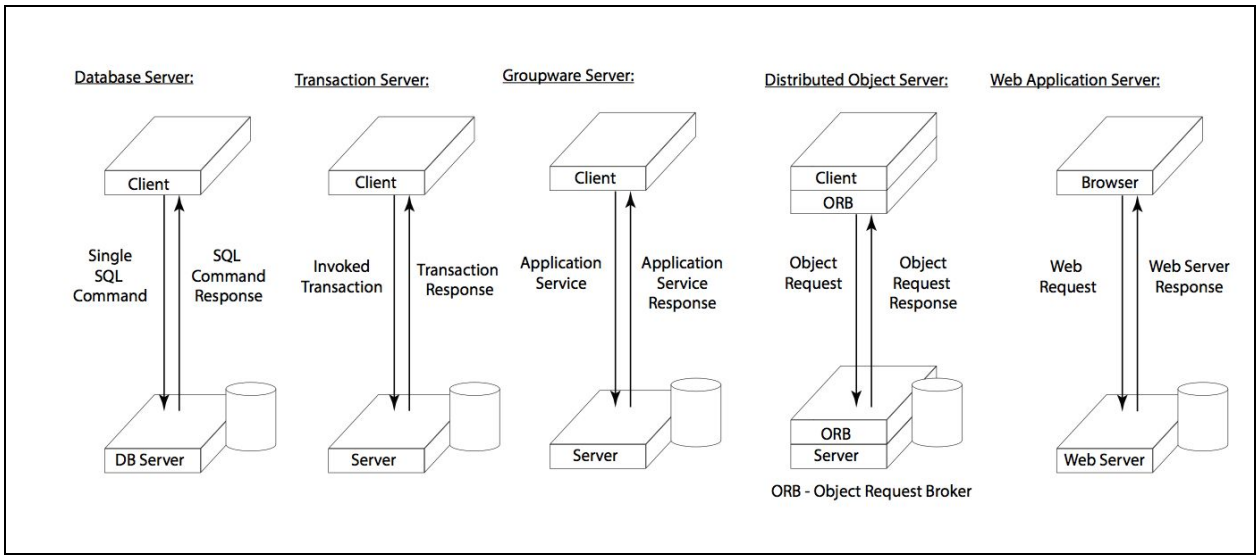

Figure 1: Architecture Types

Our system architecture takes the form of a Web Application Server where the users can access commands through their brower. As discussed, this provides a number of security advantages. One is that the client is abstracted far away from the database information and will never be able to interact with it directly. Modern web browsers include a number of security considerations, which adds one more layer of security mechanisms in all transactions.

However, through doing the emulation testing with a workload generator and workload file, the architecture functioned more like a Transaction Server system as requests would be made directly to the web server instead of through a browser. There is some security here still though, as the set of allowable commands is limited to the user as compared to having direct access like in the Database Server model. Overall this is rather weak protection however, as if our system were to be put live malicious users could easily send commands directly to the web server.

[2] Slide 105/235 Distributed Systems Software, January - April 2018, Stephen W. Neville

4.0 Security Inclusion

Some of the frameworks leveraged from our implementation include some build in security features, some things were added to the project intentionally as well.

4.1 Intentional Implementations

We used prepared statements for all of our database queries to prevent SQL injection. This was done by using Golang's sql.prepare() and stmt.exec() function. These functions parse the parameters received by our endpoints and message queues, and checks that the query is not malformed. The prepared statements can be reused and safely used by multiple threads which is an added performance benefit.

We specified the content type of our http responses in the http header so that the client only accepts the json data from our servers. This is done with the function w.Header().Add("Content-Type", "application/json"). This will prevent clients from accepting potentially malicious data

4.2 Framework Security Inclusions

4.2.1 Docker

The use of Docker adds an extra layer of security between the application and host machines. Containerizing the application prevents it from having direct access to the host machine. Docker creates a set of kernel namespaces for the container which prevents processes from seeing processes running in another container or in the host system.

Containers have their own network stack, meaning that a container does not get privileged access so the sockets or interfaces of another container. Containers can only interact via transport and internet layer protocols. This is as if each container is on its own machine connected via an ethernet switch.

Each container is allotted a fair share of CPU, RAM, disk I/O by control groups. The control groups also prevent one container from using all of the resources of the host machine, preventing other containers from crashing if one container is under heavy load.

In the event a docker container is compromised, these features will help prevent other containers from being affected.

4.2.2 Redis

The Redis security model dictates that Redis should run in a somewhat locked down environment, and that its TCP port should not be exposed to any potential untrustworthy clients. This is exactly the way we have used Redis in our system, due to the fact that entire lab is completely locked down with no internet access, the only potential clients are trustworthy. Obviously this will not be the case in a move to a production environment, and changes regarding this are discussed later in the document. Redis has no string escaping which makes injection impossible under normal circumstances. Redis also uses length-prefixed strings to prevent buffer overflow attacks.

4.2.3 RabbitMQ

All of our RabbitMQ connections were made using the provided “Guest” user. The “Guest” user is only allowed to connect from localhost, effectively makes remote connections to our RabbitMQ instance impossible.

5.0 Future Considerations

Before deploying this day trading system to a production environment, there are a number of things we would implement to improve overall security of the system.

5.1 General Web Practices (Golang)

There are some fairly easy ways to increase the security in Golang web applications; there is an active example project called GoSea (Go secure example application) that follows some basic security practices[3].

Summarized are some of the ways to improve security[4]. Serving over HTTPS rather than HTTP ensures that all traffic between client and server is encrypted. To implement this on the server side, we will just need to generate public and private keys, and use the function http.ListenAndServeTLS to set our endpoint handlers. This will ensure that any day trader using the system will not have their sensitive trade information being vulnerable to anybody else on their network sniffing their data.

A move to production would require us to scrap the assumption we have been working on thus far that all users sending commands to the system have already been authenticated by a third party authentication service. We would need to implement this authentication service to ensure the integrity of the trading service. This authentication service could be implemented using JSON Web Tokens, for which Golang offers a package. We would implement a token handler

[3] https://github.com/komand/gosea
[4] https://blog.rapid7.com/2016/07/13/quick-security-wins-in-golang/

endpoint to generate and sign tokens for incoming users, and then implement a golang middleware to validate tokens on any endpoint deemed necessary. This would be vital in moving to a production system, however it was not in the scope of the project.

On all of the internal services such as the quote server cache, transaction servers, Postgres instances, and the audit server, we only want to allow specific hosts to have access. This would be implemented by maintaining a list of allowed hosts, and checking whether a request came from a host on the list, if not simply block the request and audit it. This protects our internal services from unwanted domains hitting them in ways we may not want them to.

Cross Site Scripting, or XSS, is perhaps one of the most common web attacks seen today, and preventing them within the day trading platform helps secure the experience for our users. We can implement this by setting the “X-XSS-Protection” field in the HTTP headers of all responses. Once this is set, web browsers will take care of the rest by automatically stopping page loading when they detect a possible XSS attack.

In moving to a production environment, Denial of Service attacks become a major threat to bringing down the system. Some measures we would take to mitigate the risk of such an attack occurring include ensuring that our available network bandwidth is much higher than required for typical user traffic so that in the case the system receives an enormous influx of attacker traffic, it can be dealt with without locking up the rest of the system. Additional measures such as third party DDOS prevention service would also be required in order to keep the system available for actual users during a sustained attack. Additionally, rate limiting each user ID to some set rate would prevent compromised accounts being used in attacks.

5.2 General Web Practices (Frontend)

We currently have no rigorous input sanitization in the web interface, meaning malformed or potentially harmful attack commands will make it to the web server and transaction server. This was not an area we focused on during development since we assumed that our workload files would generally be well formed and valid. We do have some basic input validity checking, but implementing full sanitization of all user inputs is vital to minimizing the chance of any attempted attacks probability of propagating through the system.

As previously mentioned, implementing HTTPS from our web server will ensure that all data sent from the web client into the system is encrypted and hidden from any attackers on the same network as our users.

5.2 Other Technologies

5.2.1 Redis

To secure our Redis installations, firewall rules would be put in place such that only connections originating from localhost would be able to connect to each Redis instance, with all remote connections being blocked. This is the simplest way to ensure security and integrity of our Redis instances.

5.2.2 RabbitMQ

In the move to operating the day trading system in production, enabling SSL in our connections to our RabbitMQ message queues is something we would certainly implement, as the time cost involved is very low, it has minimal performance impact, and adds another layer of security to inter-component communications.

5.2.3 Postgres

Since our Postgres instances currently, and most likely in the future as well, reside on seperate machines from the application servers which connect to them, completely disabling remote connections is not something we can do. Instead, setting up a proper firewall to allow only the application servers to connect to the exposed Postgres TCP port on each Postgres machine. We can also specify specific IP address to accept connections from in the Postgres configuration file “pg_hba.conf”.

6.0 Conclusion

We were able to implement some security measures into our system by following best practices and using up to date frameworks. Our system is safe from catastrophic attacks such as SQL injections and buffer overflow. In the real world, security would be a higher priority for us, however, we had to make some compromises for the sake of performance and correctness given the time span of the project.