EZShare Security and Subscribing

Team ALAS

**Ao LI**

aol3@student.

unimelb.edu.au**Lilian CHAN**

lilianc2@student.

unimelb.edu.au**Alston CHU**

[a.chu@student](mailto:a.chu@student).

unimelb.edu.au**Sheng WU**

shengw4@student.

unimelb.edu.au

# INTRODUCTION

EZShare, the resource sharing system that was developed in Project 1, contains some useability issues due to its lack of security implementations. Some security mechanisms are now put in place to lessen the risks posed by attackers on the system.

This included development of a ‘secure’ port that exists in conjunction with the ‘unsecured’ port and only accepts secure connections. Queries that are sent from clients from their secure ports are now only relayed to other servers on their secure ports, and they are managed separately from those with unsecure connections.

Furthermore, a ‘subscribe’ command was implemented, through which clients can receive an unbounded number of responses of resources that match the template in real time. The connection remains open until the client sends an unsubscribe command in the form of pressing ‘enter’.

# 2. SECURITY ISSUES

Given the domain of hobbyist resource-sharing, therefore placing exchange of money out of scope, the major security concerns are as follows: resource access privacy for clients, integrity for those resources, continued service provision in face of an attacker, and resource authorisation. In general, users must express a certain trust in the certificate authority to evaluate protocol adherence and security measures of relevant applications before issuing certificates. Otherwise, they would need to place hope that every client and server is correctly coded.

While the use of SSL may mask exact message contents on the network, it doesn't prevent an attacker from seeing messages being sent, nor can this guarantee the confidentiality of server logs. Depending on the implementation, metadata analysis might determine message length, greatly narrowing a message's potential type and/or contents. Overall, while privacy is not a particularly promised trait of the service, it is one that users may wrongfully assume has been made.

The use of SSL does, however, guarantee that the resource the user receives is very likely the resource the server sent. That is to say, man in the middle or other interception attacks are unlikely. The breadth of file types incorporating mobile code-like vulnerabilities encroaching beyond executable binaries places this in fair importance. Even a relatively savvy user may feel safe downloading media or document files that could present arbitrary code execution-level risks. With SSL, it is practically very difficult for someone to alter or fabricate messages in a way that is not visible to the protocol, let alone in ways that can be guided to a specific aim.

Currently, implementation of a connection interval limit restricts that one client can only send a single request to the server at the set interval time. This allows control over the number of subscriptions a single client can make to the server. In part, this will address the issue of denial of service attacks, among other vulnerabilities. However, as exchanges are largely single commands, IP spoofing could be used to circumvent the interval limit, allowing a single connection to overload a server. One could flood each server in a network with relay commands. Even a stream of publish, relayed subscribe, and exchange commands would be enough to steadily degrade performance, as the server cross checks each new resource against every subscription, each new subscription against every resource, and each new server with a set of relayed subscription requests.

A major problem with the owner/channel authorisation system is that there is nothing to prevent a client from sending this presumably secret information in plaintext. An eavesdropper could then attain the same access to resources as the original sender, ranging from unauthorised resource access, to rerouting published URIs, to removing resources. Similarly, an overheard plaintext share command gives any listener the ability to download any file local to the server that they can guess the URI of. The spec also doesn't specify a particular method or routine for changing the secret beyond random generation on server start, which is bad practice if not as severe a problem. Removing the functionality for clients to send and servers to process owner or channel-containing commands without the use of the secure port easily remedies the former problem, and requiring certified servers to be coded to change secrets with some regularity remedies the latter. Both solutions add performance overheads, but the largest of which apply only to commands that need owner and/or channel, and which are well justifiable.

Relying on certificates, one could more confidently relay queries with owner and channel intact. Previously, anyone would have been able to use an exchange command to register themselves to a server to harvest relayed owner and channel information without the need for a valid certificate (or indeed an actual server implementation). The crux is, however, that a simple password over the line system will always be ill-fitted to provide strong authorisation and authentication promises. A challenge-response protocol or a ticket server system would be a great improvement, but with a correspondingly significant complexity increase. With domain in mind, and the desire to maintain compact, easily certifiable programs, it doesn't seem practical to attempt many hard guarantees on this level. Greater reliance on relaying, and restricting ownership-relevant commands to local users solves many authentication issues, but compromises certain functionality. Removing resource-overwriting, adding expiry, and relying on user-level file encryption solves many of the other issues, but again compromises functionality. To conclude and reiterate, the concept and domain don't lend themselves particularly well to any solution that is both lightweight and secure, but this may be acceptable in the current application.

# 3. SUBSCRIBING

Clients can send a ‘subscribe’ command to the server in order to continuously receive responses that match the specified query template and the server will likewise automatically make subscription requests to all the other known servers on its list, using the same resource template. There will be separate lists of ‘secure’ and ‘unsecure’ servers, and the relevant servers are contacted depending on which port the client has sent the request through. However, since the secure port and insecure port share the same resources, all matched resources will be returned to the client regardless of secure connection or insecure connection. The only difference of using the secure port and insecure port is that the servers are contacted through different ports.

In the current implementation, multiple clients can connect with the server at once. A single client can also send multiple subscription requests to the server by using different terminals, meaning they can send subscribe commands to the server with the same IP address but different port number. Each of these subscriptions is managed through a separate thread. Likewise, the server connects to each server on its exchange list through individual connections for each subscription. When new servers become known, the subscription request is extended to that server. Connections are automatically closed when old servers are detected to have crashed. All of the connections between the servers are persistent asynchronous connections, through which resources can be intermittently shared as they become available, for an indefinite amount of time or until the ‘unsubscribe’ request is made through ‘enter’, or a line of standard input.

Using this method, the client can receive the most updated list of resources at any time, as they are received immediately when they become known. No blocks would need to be put in place throughout the communication. However, it can be predicted that this system would require many different connections to be opened simultaneously. A problem may be encountered when the server reaches its maximum allowed number of connections, so that new subscription requests cannot be made. Servers are also slow at detecting connection timeouts, so that resources are not used efficiently when operating at the maximum number of connections.

An alternative approach is to use a synchronous connection between the server and the client. The server would not have to subscribe to all the servers on its list and can alternatively repeatedly ‘query’ them at a specified interval time. All the matches returned and newly available resources can be stored in the server, then sent back to the client at once. Hence the client will receive responses only in blocks at the set interval times.

An advantage of using this approach is that not as many connections need to be opened, preserving computing power and reducing the risk of overloading the server. It is a relatively simplistic model that results in easy-to-understand architecture and does not require as much programming in regards of thread management.

However, the disadvantage would be that responses are not received in real-time and there are delays in achieving consistency in the resource list. Moreover, if one server B within server A’s list crashes before server A makes a query request, the newly available resources on server B are not returned. Each server would need more memory for storing the resources because there are duplications of resource files across the servers in this model. This would be a problem if there are many large files, videos or images that are constantly being shared. Another disadvantage is that the new approach is not scalable, because the workload for the server that is doing the querying has been increased, having to constantly obtain resources from other servers and check them against the template. The complexity of the model will be increased at an exponential speed. Lastly, this model requires that all servers need to have a high-performance in terms of memory and processing speed.

Many considerations need to be taken regarding the purpose and requirements of the system when selecting the optimal model for system architecture.