System Security Proposal

Annie Edmundson

September 7, 2012

1 Introduction

This proposal describes necessary features for the security and privacy of information associated with the OpenComm application. More specifically, the goals of this proposal are to outline specific methods of protecting user confidentiality, maintaining information integrity, and allowing application and information availability to appropriate users. It should be noted that in most cases the security of a system is inversely related to the usability of the system; this tradeoff provides difficulty when designing the system. First, we must decide what the users' priorities are, and what we want to protect against.

1.1 Priorities

A recent study has shown that more smartphone users are considering the privacy and security implications of the applications they install [3]. It also reports that users' are becoming more wary of how much personal information they have to share with the application. While the average person seems to care more about privacy than security, the two are not always black and white. A person's privacy can be breached if the essential security measures are not implemented. On the other hand, people will not use this application if it is not easy to use; for example, it would be unreasonable to implement a cryptographic protocol on every message sent if it would make the application run at a much slower rate. An optimal solution would be to implement security features for the most secure system as possible without affecting the usability of the application. Because users do not see the technical details of the system, it is important for them to know that we are collecting minimal information and not sharing it with any third parties.

1.2 Threat Model

For the purposes of the security design, we assume that an attacker possesses some combination of the following characteristics.

- Any person with access to the application - does not necessarily have to be a registered user.
- Any person that wants to obtain, steal, or alter another user's personal information.
- Any person that wants to eavesdrop on other users' conversations.
- Any person that wants to inject arbitrary messages/packets into other users' conversations.
- Any person that wants to alter messages of other users' conversations.

- Any person that wants to steal the identity of another user and send messages/initiate conversations with other users.
- Any person that wants to end other users' conversations.
- Any person that wants to prevent other users' from using the application or sending messages.
- Any person that wants to alter another user's contact list.

2 Current State of Security

Thus far, the OpenComm team has been focused on application functionality before implementing security features. The application, as of now, does not take many security measures, but allows for their future implementation.

2.1 Issues with Current Security

Here I will describe a basic overview of what is missing in terms of security implementation; the following sections will describe in more detail what should be implemented. There has not been a systematic and complete implementation of methods for verifying what user X can and cannot do; for example, user X can edit his/her own personal information, but not user Y's. Some of the user rules have been implemented because they are also a part of the functionality. The system currently has a way of registering and logging in, which are parts of the authentication process. This can be slightly modified for a more secure process. The application is lacking an audit trail; the functionality of this feature is simple and helpful. Lastly, there are small additions we can make to provide for a more secure application as well as instilling a sense of security and privacy in the users.

3 Proposal of Security Features

The following sections expand on necessary features for authorization, authentication, and audits.

3.1 Authorization

An authorization mechanism governs whether a requested action is allowed to proceed [2]. The main method of implementing an authorization mechanism is via an access control mechanism; there are many access control mechanisms, such as access control lists, capabilities, and role-based access control.

3.1.1 Access Control

The OpenComm application involves users that are all at the same access level, meaning there is not a hierarchy of levels and there are no differentiating roles (with the exception of the administrator). With this in mind, it would be optimal to implement a capabilities-based access control mechanism; this is a system where each user has certain capabilities. For example, user X has the capabilities: editing user X's personal information, starting conferences, inviting anyone in his/her contact list to the conference, etc. If user X tries to execute an action that he/she does not have the capability for, then the action should be denied. If user X deactivates his/her account, then the capabilities of everyone in connection to him/her must be updated; for example, all users with user X as a contact must have their contact lists updated.

3.2 Authentication

An authentication mechanism associates a principal and perhaps those it speaks for with actions or communications [2]. A successful authentication process relies on the creation, use, and storage of passwords as well as the registration and login functionality.

3.2.1 Passwords

We are using a knowledge-based authentication mechanism: a user-supplied password. In order to prevent malicious users from easily guessing other users' passwords, we should impose restrictions on length and characters/numbers used. A security expert reported that the length of a password is more important than the complexity of the characters used in the password [5]. For this reason, we should have a length requirement of 8 characters, and restrict usable characters to alphanumeric characters. The restriction to alphanumeric characters is to limit code injection; see Section 3.4 for more information about code injection prevention.

We also need to consider the transport and storage of the password. A password should never be stored anywhere in plaintext as this leaves all passwords vulnerable to attackers. A solution is to use a one-way hash function on the user-supplied password concatenated with a random salt value. This would result in the grouping ¡username, n, H(pass+n)¿, where H() is the one-way hash function and n is a nonce (the salt value). The server would store an encrypted file containing each user's grouping. The following demonstrates this process.

1. User X inputs and sends username and password to the system.
2. The system retrieves user X's grouping based on username.
3. The system concatenates the recently sent password with the nonce recorded in the user X's grouping.
4. The system applies the hash function to the concatenated values.
5. If the result of this hash function is equal to the third item in the grouping, then user X is authenticated.

3.2.2 Cryptographic Protocols

There are two cryptographic protocols that are necessary: (1) when authenticating the user and (2) when sending all other messages either between the client and the server or between clients. For both of these protocols, another popular and widely-used video-conferencing application was referenced [1].

First, I will discuss the encryption method used for sending all regular messages; regular messages are defined as all messages (audio and non-audio) not involved in the authentication process. For regular messages, we will use a shared-key (symmetric) cryptosystem; both the system and the user use the same key to encrypt and decrypt messages. The specific protocol suggested is AES 256-bit encryption.

In order for both parties to obtain the same shared key, there has to be a sufficient cryptographic protocol used when authenticating the user. For this step, we will use a public-key (asymmetric) cryptosystem. In this protocol, each entity has a public and private key pair; if Alice wants to send a message to Bob, then Alice must know Bob's public key. Alice encrypts her message with Bob's public key and sends it to him. When Bob receives the message from Alice, he decrypts it using his private key. A user's private key is not shared with anyone else; if this private key is somehow discovered, it nullifies the protocol. When the user successfully logs into the application, he/she will generate a public and private key pair. The user will then send the server his/her public key. The server will generate a random shared key, encrypt it with the user's public key, and send it to the user. The user will decrypt the message (shared key) with his/her private key. It would be beneficial to verify the shared key by having the server send a random number encrypted with the shared key to the user; then the user would use the recently received shared key to decrypt this random number, add 1 to it, encrypt this new number, and send it back to the server. The server will verify this number against the one it originally sent; if the user passes this test, then the application continues as normal using the shared key and removing all traces of the public and private keys associated with that user. The specific protocol suggested is either 1536 or 2048-bit RSA encryption. It should be noted that public-key encryption is much slower than secret-key encryption, and thus should solely be used for authentication.

3.3 Audit Log

An audit mechanism records system activity, attributing each action to some responsible party [2]. An audit log is important in the case that an attack does occur. Looking back at the audit log, it would be possible to figure out when and how the attack occurred. While this information could potentially be very useful, we want to make sure this log is securely stored and that no one (except the administrator) has access to the log; the administrator should only have read access to the log. We also want to carefully consider what information gets logged; collecting too much information could become a privacy implication of the application. In order to recreate the events of an attack, we would need to know what the event was, who executed the event, and when the event occurred; an event constitutes any addition, change, or deletion of information as well as all actions associated with conversations/chats and modifications to contact lists. The log would be stored server-side.

3.4 Other Features

Prevention of Dictionary Attacks. A mechanism to prevent automated login attempts should be implemented. Two possible solutions are: (1) a limited number of login attempts per time period or (2) use a slow one-way hash function when checking login credentials in order for login time to be delayed.

Prevention of Code Injection. All user input should be sanitized and/or validated so no code injection (PHP or SQL) is possible.

Password Confidentiality Verification. An easy way to make the user more confident that there password has not been compromised is to display the last time/date that they logged in whenever they log in. If they notice a discrepancy, they can reset their password.

Prevention of Replay Attacks. Each message/packet should include a counter or a unique identifier. This identifier should be checked by the receiving entity to ensure it has not been previously received. This prevents a malicious user from stealing a packet and then resending it later.

Storage and Transportation Integrity. It is necessary to ensure a malicious user does not tamper with any of the data. We should implement integrity checks on data being transported as well as data that is being stored to make sure the data has not been changed by unauthorized users. This could be implemented by applying a known hash function on the packet/message; the result of this hash is the checksum. The checksum would be appended to the message/packet, and the receiving entity would apply the hash function to the message and compare the result to the checksum sent with the message. This should also be implemented on the storage system.

Storage Confidentiality. All files in the database or stored otherwise, should be encrypted using a key or password only know to the server and/or the administrator.

4 Implementation Timeline

The implementation of security features could be done in 4 milestones:

1. Authorization
2. Authentication
3. Audit
4. Other Features

Each of these milestones involves different quantities of work, and would not all take the same amount of time. Additionally, the time these milestones take depends greatly on the number of people implementing them. It would be ideal to finish these by the end of the semester; more concrete deadlines for each milestone can be specified when more information is known about the number of people working on implementation.

5 Notes on Android Security Environment

This section introduces a basic overview of Android Security and how it pertains to OpenComm, but a more detailed information can be found at [4]. The security features discussed above are for a contained system, but as an Android application we want to ensure that other applications cannot affect the OpenComm application. It is not necessary to analyze security implementations in the operating system or hardware, but it should be noted that the Android platform is built on the Linux kernel; this means the Android platform has the same security as the Linux kernel and has secure inter-process communication. Having secure communications ensures us that other applications' processes cannot affect OpenComm processes. Knowing this, we will next focus on the Android Application Runtime. All applications run in the same environment: the Application Sandbox (this is mandatory). Each application is assigned a specific user id and is run as that user in a separate process. As all applications are run as separate processes and the default of the Android system is to not allow applications to communicate with each other (and have limited access to the operating system), we do not need to worry about problems arising from the Android platform.

All applications (as an aggregate) are allocated a section of the filesystem, but OpenComm does not store any information locally, so this is not applicable.

Having said the above, there are still vulnerabilities that many Android applications contain. The following is a list of the most common vulnerabilities:

- Unauthorized Intent Receipt
- Intent Spoofing
- Persistent Messages: Broadcast Spoofing
- Insecure Storage
- Insecure Network Communication
- SQL Injection
- Overprivileged Applications

There is more information about these vulnerabiities at [6]; we will need to protect against possible exploits of the applicable vulnerabilities.

A note about Android version: the more recent versions have more security features than older versions.

5.1 Android Permissions

The Android permissions are specified by the developer (under certain constraints); for example, if the application uses the Internet, then it needs the user's permission to use the Internet. Before an application can be downloaded, the user must grant all of these permissions. We want to make sure we do not request any permissions that are not necessary for our application.

References

[1] Does Skype use encryption? `https://support.skype.com/en-us/faq/FA31/does-skype-use-encryption`, 2012.

[2] Fred B. Schneider. Real World Physical Security. `http://www.cs.cornell.edu/courses/CS5430/2012sp/paper.chptr01.pdf`, 2007.

[3] Kashmir Hill. Sorry, Smartphone Owners, But You're More Likely To Have Your Privacy Invaded. `http://www.forbes.com/sites/kashmirhill/2012/09/06/sorry-smartphone-owners-but-youre-more-likely-to-have-your-privacy-invaded/`, 2012.

[4] Open Source Project. Android Security Overview. `http://source.android.com/tech/security/index.html`.

[5] Roger A. Grimes. Password size does matter. `http://www.infoworld.com/d/security-central/password-size-does-matter-531`, 2006.

[6] Yekaterina Tsipenyuk O'Neil and Erika Chin. Seven Ways to Hang Yourself with Google Android. `http://www.eecs.berkeley.edu/~emc/slides/SevenWaysToHangYourselfWithGoogleAndroid.pdf`, 2011.