DeadDrop:
Message Passing Without Metadata Leakage

By Davis Arndt

Computer Science Department
College of Engineering
California Polytechnic State University
San Luis Obispo

June 2018

Table of Contents

Introduction

Problem:

Cryptographic communication schemes, even when paired with anonymized routing, tend to be vulnerable to extraction and analysis of metadata. Those that require users to trust a central server or service could potentially yield a great deal of information if the trusted resource is compromised, and any system that sends data across an unsecured network is vulnerable to traffic analysis to some degree. The purpose of this project was to explore a solution to the problem of metadata collection and analysis.

Scope:

The scope of this project was limited to creating a data transfer system that places no trust whatsoever in the centralized system while eliminating, to the greatest degree possible, the ability of an outside observer to analyze and correlate traffic. This system would consist of a single node and and API designed to interact with it. Other features necessary or beneficial to cryptographic messaging, such as key exchange and storage, authentication, and anonymized routing, are not implemented by this project, though it is designed to be able to incorporate them.

Solution:

In order to prevent a central server or outside observers from gleaning anything from traffic patterns, it needs be impossible to correlate incoming and outgoing traffic. To accomplish this, I designed a system in which the server is unaware of the intended recipient of any given message, and all recipients receive a copy of every encrypted message currently on the server. Because the server has no way of knowing where to send a message once received, it is up to each client to check for and retrieve messages of their own accord.

Background

Hybrid Cryptography:

Asymmetric cryptography, in which each participant in a conversation holds a different key, is extremely useful for purposes of identification and overall cryptographic robustness. However, it relies on extremely inefficient mathematical operations and is therefore slow. Symmetric cryptography, where participants have the same key, is much faster. Hybrid cryptography gains the strength of both by using an asymmetric algorithm to encrypt a symmetric key, which is sent to the recipient alongside and used to encrypt the message [1]. Done properly, such a system can be even more secure than either type used alone.

Traffic Analysis:

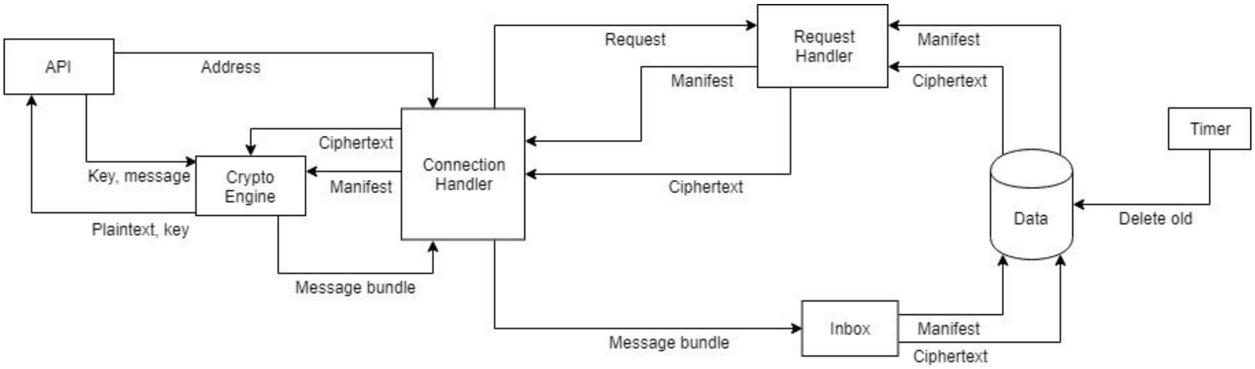
Even when internet traffic is encrypted and routed anonymously, an observer can often determine its origin by analyzing it. For example, if an observer sees a user send a packet of a certain size into a routing system, then sees a similarly-sized packet exit elsewhere a short time later, it is a hint that the two are the same [2]. This can allow an observer to know information such as who is talking to whom, with what frequency, and how much they are saying. On its own, such information is not always terribly harmful. Combined with other small pieces of data, however, a great deal of information can be built up without ever breaking the protections on the data being transferred.

Server Trust:

Cryptographic systems that make use of a central trusted server are extremely convenient and powerful. Such a system enables easy key exchange, efficient routing, and a variety of other benefits. However, placing trust in a server that you don’t control is a risky proposition. A server that participates in facilitating every conversation, such as those used with the Signal cryptosystem, is able to form a detailed record of who is conversing when [3]. Even if you can trust the organization in charge of the server not to abuse their power, there is always the chance that they will be coerced by a government agency to collect and turn over data [4].

Design

<u>Overview Diagram</u>



Repository: https://github.com/DaveArndt/DeadDrop

Server:

Designing the server was the simplest part of this project. The general idea is a system that handles client connections and acts as an intermediary between client requests and the backend MySQL database. It doesn't manipulate the data in any way, and as discussed earlier, lacks the information required to meaningfully manipulate it.

Interaction with the server follows one of three paths. First, a client can request to store a message, providing it to the server as encrypted binary data. This is stored to the database, along with an auto-generated ID and the time of upload. Second, a client can request the server's manifest. Manifest entries, consisting of the recipient's ID, sender's ID, and AES symmetric key, all encrypted with the recipient's public key, are constructed by the sender before uploading and are mapped to the corresponding message in the database. This allows the client to identify which messages, if any, are intended for them, as well as providing the symmetric key and sender's ID. Third, a client can request any given message by ID number, which is returned alongside a signed hash computed by the sender.

Because the server can't authenticate, or even identify, individual users, users have no way to delete messages from the system once they are posted. To prevent messages from piling up and bogging down the system, a parallel thread, running on a timed duration given at startup, deletes messages older than a certain time period, also given at startup. Should the parent thread crash, this cleanup thread will kill itself the next time it runs.

Client:

The client is responsible for all data processing, including cryptography. It was designed as an API instead of a runnable application both to increase overall flexibility and stay within the scope of the project.

The client API communicates with the server in the ways outlined above, but its real work is done in the manipulation and packaging of data to facilitate communication with other clients. After downloading the manifest, it iterates through and decrypts each entry using the recipient’s private RSA key. If the recipient ID matches that given to the method, the message database ID, sender ID, and AES key are appended as a tuple to a master list. When done, another method is used to download each message from the server, ignoring those not present in the list. Finally, each message is decrypted using its AES key and the signature is verified. What to do in the case of an invalid signature is left to whomever is using the API. In addition, it is up to the user to manage key storage and retrieval, though helper methods are provided to facilitate storing to and retrieving RSA keys from files.

Data Structure:

Data internal to this system comes in three forms. One of each exists for each message: ciphertext, signature, and manifest entry. The ciphertext is just the plaintext, padded and truncated to 1024 bytes and encrypted with a 256-bit AES key randomly generated for each uploaded message. The signature consists of a SHA256 hash of the padded plaintext appended to the AES key, signed using the sender’s private 2048-bit RSA key. The inclusion of the AES key in the signature is intended to offset the small message size by making rainbow table attacks infeasible. The manifest entry is comprised of the recipient and sender usernames, each padded to 16 bytes, joined with the AES key for this message, all encrypted with the recipient’s private RSA key. This is the most important piece of data, as the manifest is what allows a client to check for messages without parsing unnecessary data.

Evaluation

Security Requirements:

First, the system must protect the data contained in the message. The message itself is protected by an AES cipher, the key to which exists in two locations: the signed hash, and the manifest, which itself is protected by an RSA cipher and a user-provided key. As key management is outside the scope of this project, it is up to the user of the API to ensure its security. As long as neither RSA nor AES are broken, the data should remain safe.

Second, the system must not provide the server with any information directly. Because all data is either encrypted or hashed, and the server is never told to whom the message is intended, the data provided to the server amounts to nothing more than binary gibberish.

Finally, the system must protect metadata as much as possible. The server could theoretically apply traffic analysis to incoming messages. This would potentially allow it to determine information about the sender, possibly discerning patterns or correlating received messages with each other. Likewise, it might be possible to fingerprint a recipient based on patterns of checking messages or the speed of their connection. Neither of these are ideal, but both are mitigatable through the supplementary use of anonymous routing, and at worst the server or observers would simply learn that an individual uses the service. Because all outgoing data is identical for a given moment in time, and because data is not immediately routed to its destination, it is impossible to correlate incoming and outgoing communications by analysis of packet contents or timing. Because not all metadata is perfectly obfuscated, this requirement is not completely met. However, it does prevent observers from connecting senders to recipients.

Load Handling:

This metric was always going to be the most difficult to optimize. By necessity, this scheme is extremely inefficient during message retrieval. As the number of messages in the database increases, performance degrades. This is somewhat mitigated through the use of hybrid encryption to minimize time spent decrypting messages, but the rate-determining step is the download itself. The performance for each user varies greatly based on computer and network performance, especially if using anonymous routing. On a computer with middling hardware and network connections running through localhost, I recorded the following execution times to download and decrypt the given number of messages:

Number of Messages	Execution Time (s)
10	0.422
20	0.798
30	0.719
40	0.880
50	3.122
60	1.158
70	1.247
80	1.481
90	1.633
100	1.693
110	1.815
120	2.036
130	2.106
140	2.312
150	2.374

Even with relatively small numbers of messages and minimal network latency, the lag was noticeable, if tolerable. With a sufficient user base, the problem would only get worse. In its current form, this system operates best when used by a relatively small group, but more users increases its ability to occlude metadata. These opposing interests limit the overall feasibility.

Conclusions

Successes:

With this project, I accomplished exactly what I set out to do. The system I created, when used correctly, hides a great deal of information that would otherwise be available to anyone with the resources to catch it all. It is small and simple enough to be inserted into almost any network system, and is flexible enough to fit into and work alongside other security systems. Most importantly, I learned at least as much just sitting down and building this as I did in most of my classes.

Failures:

I can't claim that using this system is worthwhile to anybody but the most paranoid of groups. Notice that I specified "groups." The nature of this system is that it slows down heavily as use increases, but barely provides any protection if there aren't enough other users with which to blend in. I speculate that this system would work best when the people using it are coordinated in some way so as not to overtax the system, and therefore would be best suited to a medium-sized organization. However, doing so would defeat much of the purpose of hiding your messages' destinations in the first place.

In addition, while it does provide protection from many forms of analysis, it isn't perfect. It is still possible to identify individuals connecting to the server, though their messages can't be tracked.

Improvements:

Had I the time, I might be able to mitigate the scaling issue by downloading a random subset of all messages instead of grabbing all every time. This could allow the system to support a much larger user base with minimal slowing. It would weaken the decoupling between sender and recipient provided by the current build, though, as well as make it easier for an observer to identify a specific user. I couldn't think of a way to incorporate it without negating the rest of the system, so I didn't include it.

References

[1] Kuppuswamy, Prakash and Saeed Q. Y. Al-Khalidi. (March 2014) *Hybrid Encryption/Decryption Technique Using New Public Key and Symmetric Key Algorithm.* MIS Review Vol. 19, No. 2. https://pdfs.semanticscholar.org/87ff/ea85fbf52e22e4808e1fcc9e40ead4ff7738.pdf

[2] Northcutt, Stephen. (May 2007) “Traffic Analysis.” SANS Technology Institute. https://www.sans.edu/cyber-research/security-laboratory/article/traffic-analysis

[3] Rottermanner, Christoph; Kieseberg, Peter; Huber, Markus; Schmiedecker, Martin; Schrittwieser, Sebastian (December 2015). *Privacy and Data Protection in Smartphone Messengers*. Proceedings of the 17th International Conference on Information Integration and Web-based Applications & Services (iiWAS2015). ACM International Conference Proceedings Series. Retrieved 27 February 2018. https://www.sba-research.org/wp-content/uploads/publications/paper_drafthp.pdf

[4] Fox-Brewster, Thomas. “Forget About Backdoors, This Is The Data WhatsApp Actually Hands To Cops.” Forbes, 22 Jan. 2017, www.forbes.com/sites/thomasbrewster/2017/01/22/whatsapp-facebook-backdoor-government-data-request/#4976977a1030