Hoot

Dustin Bachrach
Department of Computer Science
Rice University
Houston, Texas
ahdustin@gmail.com

Christopher Nunu
Department of Computer Science
Rice University
Houston, Texas
canunu@gmail.com

Dan Wallach
Department of Computer Science
Rice University
Houston, Texas
dwallach@rice.edu

Matt Wright
Computer Science and Engineering
University of Texas at Arlington
Arlington, Texas
mwright@cse.uta.edu

ABSTRACT
TODO:

1. INTRODUCTION
- Problem: Twitter like semantics w/ encrypted messages - Follow a Hash tag - Take hash tag and create something with crypto strength - Something derived from tags you can search on - But also deliberate collisions (cover traffic)

- Like to have thing that feels like twitter but anonymity properties:

- Twitter/Facebook relevant in Tunisia, (Social media playing big role in revolution across many countries. govt deliberately shut down)

- While we cannot keep them from filtering out service altogether, want to have private communication in plain sight (not stenographic)

- Strong crypto usable by people whispering to each other in streets

- Only trusted channel is not electronic (spoken word), to exchange key.

- complimentary to Tor, solve problems Tor+Twitter does not

- What we are doing – Define a protocol for users to communicate over an insecure public network like twitter with message confidentiality and subscriber anonymity.

- Punting on plain tag distribution

- Vocabulary

2. BACKGROUND
1) History of government censorship, man in the middle - Tunisia code injection - Chinese firewall - Crypto keys for important services to iranian source (Komodo) - Person providing netwrok (even over ssl) might be evil

2) Tor - Trying to work against government censorship

- Group Crypto Keys

3. DEFINITIONS
- What is anonymous

- Cover traffic, k anonymity

- Privacy

- Define anonymity — Sender anonymity — Subscriber anonymity

- Basic Crypto: Message auth codes Hash Functions AES Vanilla crypto

4. DESIGN
The Hoot protocol consists of a message header and a message body. The header contains the identifier, session key, digest key, and integrity digest. Messages are indirect in that they do not reference recipients. Anyone who knows the shared secret must be able to read the message, but should also be able to find the message easily. Rather than attempting to treat every message posted to Twitter as a potential group message, and thus decrypting the entire Twitter stream, the protocol places an identifier into every Hoot so a fellow group member can simply search for the identifier to see all potential messages. With a constant group identifier, readers can also publicly follow that identifier like any other Hash Tag on Twitter.

To create an identifier, we must find a short set of bits that are derivable from the shared secret, but give an attacker little information about the shared secret itself. Hash functions provide a great way to get a set of bits from a shared secret without divulging much information about the original shared secret. In our protocol, the shared

secret is called a *Plain Tag*, which is comparable to a Twitter Hash Tag. The result of hashing the Plain Tag with a given hash function, *H*, is referred to as the *Long Tag*. Thus,

$$LongTag \leftarrow H(PlainTag) \tag{1}$$

The protocol could simply use the Long Tag as an identifier, but this choice leads to several problems. First, to uphold our design goal of being concise and to fit within Twitter's 140 character limit, it is infeasible to use the full output of a hash function. Secondly, strong hash functions do not produce many collisions. If a group is communicating in secret, not only do we want to protect the content of the communication, but we would like to conceal subscription to a particular group. For example, a rebellion group wishes to communicate over Twitter using Hoots, but it can be dangerous for a supporter of the rebellion to listen and subscribe to the identifier of the group. If, however, the identifier of the group can collide with the identifier of a popular Internet topic, like Charlie Sheen, group followers can shadow their rebellious activities with other innocent topics.

To generate a collision, we need to shorten the long tag, generating a *Short Tag*, which will induce more collisions. The shorter the Short Tag, the higher the collision rate will be and the less sure an observer can be of what topic a Hoot reader is listening to. With this greater anonymity comes more computational work, however. Since more group messages will now be belong to the same identifier, a follower must decrypt more messages to find relevant signal. Depending on the required degree of subscriber anonymity, more collisions might be worth the computational overhead. Also, even given a constant Short Tag length, a group can choose a tag that will collide with extremely popular tags to generate even more noise. Given a Long tag of byte-length *N*, we produce a Short Tag by taking the first *K* bytes of it:

$$ShortTag \leftarrow LongTag_{[0:K]} : K < N \tag{2}$$

The header also contains an set of encrypted keys, *(Session Key, Digest Key)*. To combat replay attacks, the protocol uses a random session key to encrypt the plain text. This session key is then included in the set of keys in the header. The protocol ensures integrity the standard way by including an HMAC of the cipher text. The key to the HMAC function is randomly generated and is included in these set of keys:

$$SessionKey \leftarrow randomBytes() \tag{3}$$
$$DigestKey \leftarrow randomBytes() \tag{4}$$

To encrypt this set of keys, we exploit the entropy of the Long Tag. Since the identifier for a message is only the first *K* bytes in the Long Tag, we can use the latter set of bytes elsewhere. By encrypting the keys with part of the Long Tag, a reader can easily know the key to decrypt with since it is simply the hash of the Plain Tag, and we also do not require knowledge of two different secrets. Thus, given an encryption function and corresponding key, E_{key}, we have the encrypted keys as:

$$KeyPair \leftarrow E_{LongTag_{[J:N]}}(SessionKey.DigestKey) \quad \text{where } J > K \wedge J < N \tag{5}$$

The body of the Hoot is the cypher text, *C*, of a message, *M*, encrypted with the Session Key:

$$C \leftarrow E_{SessionKey}(M) \tag{6}$$

The header also contains the integrity digest, *D*, of the cypher text, using the Digest Key in the header. By including an integrity digest, the protocol allows for quick verification that a message is for a specific group without having to decrypt the message. It also verifies that the Hoot has not been tampered with.

$$D \leftarrow HMAC_{DigestKey}(C) \tag{7}$$

Therefore, a complete Hoot appears as:

$$Hoot \leftarrow ShortTag.KeyPair.D.C \tag{8}$$

5. IMPLEMENTATION

In this section we describe an implementation of the Hoot protocol we create for our prototype.

5.1 Generating a Hoot

The Hoot protocol can be implemented in a variety of ways using different encryption and hashing algorithms. We created a Python prototype that fully implements the protocol.

A Long Tag is generated by running a SHA256 hash over the Plain Tag. The first 128 bits of the Long Tag are dedicated to identification. We then take the first m bytes of the Long Tag to get a Short Tag, where m is the desired Short Tag length.

The session key is 128 random bits, and the integrity key is 160 random bits. These random numbers are concatenated together and then encrypted using AES with the last 128 bits of the Long Tag as the key.

The plain text is encrypted using AES with the session key as the key. An integrity check is created by performing an HMAC-SHA1 of the cipher text and the integrity key as the key.

With all components created, a Hoot can be generated by printing out a # symbol, the short tag, a space, the encrypted keys, the HMAC digest, and the cipher text.

5.2 Message Length

Given our goal to conform to Twitter, the final output of a message must be under the 140 character limit. Another goal is to be concise and use as little overhead as possible for the entire encryption process. Twitter has a very broad definition of a character. It is not simply ASCII characters, but UTF-8 as well. Based on the encoding, we can squeeze the encryption into even fewer characters. However, this manipulation has issues since many clients do