Report of Security Audit of Cryptocat

• Principal Investigators:

  • Nathan Wilcox <nathan@LeastAuthority.com>

  • Zooko Wilcox-O'Hearn <zooko@LeastAuthority.com>

  • Daira Hopwood <daira@LeastAuthority.com>

  • Darius Bacon <darius@LeastAuthority.com>

Contents

Overview

The Least Authority security consultancy performed a security audit of the Cryptocat messaging client, on behalf of *Cryptocat*'s sponsor Open Technology Fund. Cryptocat provides end-to-end encrypted chat using a web browser add-on.

This report gives the results of the audit.

Audit Scope

The audit investigated essential security properties such as the confidentiality and integrity protection of *Cryptocat* chat sessions and file transfers. The audit techniques included interactive penetration testing, code and design analysis, and discussion with developers.

For the purposes of this audit, we assume integrity of the *Cryptocat* add-on installed by the user.

A well-known outstanding attack is the side channel of timing information emitted by the implementation of cryptographic algorithms computing on secrets. It is an unsolved problem how to prevent that information leakage with cryptographic algorithms implemented in *JavaScript*. This issue is outside the scope of this audit.

Coverage

Our code audit covered all of the primary *Cryptocat* implementation in `src/core/js`, including `cryptocat.js`, and everything inside `etc/` and `workers/`. We reviewed third party code under `lib/` only when relevant to a particular investigation, which by the end of the audit included all or parts of `bigint.js`, `crypto-js/`, `elliptic.js`, `multiParty.js`, `otr.js`, `salsa20.js`, and `strophe/`.

In terms of feature sets, we focused primarily on cryptographic key management, the entropy system, the newly developed file transfer feature, and relevant aspects of the user interface.

Some notable features which we did not deeply investigate are the *multiparty chat protocol*, the implementation of the *Socialist Millionaire's Protocol*, and whether the *Off-The-Record* chat implementation (`otr.js`) is a compatible implementation of the *OTR* protocol.

For the *multiparty chat protocol*, we spent some time comparing the specification and implementation. However, we shifted focus away from this protocol for several reasons: First, we believe this protocol will be replaced by one with wider support across academia and industry, such as *mpOTR*. Second, the *multiparty chat protocol* specification would benefit from including or excluding more security features, such as transcript soundness and consensus. Finally, the specification focuses on the "how" of implementation and would benefit from more specificity of the "why" of security goals. We discuss our general recommendations for *multiparty chat protocol* in the Future Work section.

Appendix B: Work Log describes our investigation process in fine detail.

Target Code and Revision

All file references are based on the `v2.1.15` git tag of the *Cryptocat* codebase, which has revision id:

```
05ddc47d8c1beff4511199a011859ee046687614
```

All file references use *Unix*-style paths relative to the working directory root.

Findings

Issue A. Disclosure of File Contents Due to Re-use Of Key and IV

Reported: 2013-11-06

Synopsis: The file transfer feature re-uses a symmetric encryption key and *IV* in *CTR*-mode for multiple file transfers in a single *Diffie-Hellman* session.

Impact: An eavesdropper can learn some or all of the contents of the transferred files under some conditions.

Attack Resources: The attacker requires only passive collection of records of the *HTTP* traffic (or the *BOSH* and *XMPP* protocol traffic embedded in those *HTTP* requests).

Feasibility: This attack requires only simple and efficient off-line computation once records of vulnerable ciphertext traffic are known.

This loss of confidentiality occurs when more than one file is transferred through the file transfer feature during a single *Diffie-Hellman* session.

There is *no* loss of confidentiality when only a single file is transferred.

This traffic is present at the *XMPP* protocol layer, which is embedded in *HTTP* (via *BOSH*). The *HTTP* traffic of *crypto.cat*, by default uses *TLS* to and from the server.

A compromise is feasible through *at least* several vectors:

- A server compromise (including malicious insiders) could allow live traffic sniffing to recover these ciphertexts.

- A server host compromise may grant access to log files containing recorded vulnerable traffic. (In particular we know that *ejabberd* logs complete *XMPP* traffic when the log system is set to verbose. Being based on *Erlang* infrastructure, it is feasible to increase log verbosity without any interruption to the service, assuming appropriate permissions on the server.)

- Browser logs or caches may contain recorded vulnerable ciphertexts.

- Where *TLS* is deployed (including the production *crypto.cat* service), an active *TLS* attack against *TLS*-protected servers could be used to leverage this attack.

- Where *TLS* is *not* used, this attack is feasible at an unknown number of routers on the Internet.

Verification: We verified this issue by source code inspection, and by reproducing the use of identical keys for more than one file transfer using debug output from an instrumented copy of *Cryptocat*. We also used the debug output to investigate the extent to which key rollover impacts exploitation of the vulnerability.

Implementation Analysis: *AES* counter-mode encryption is used to protect the confidentiality of the file contents. The *IV* used for encrypting file contents is a fixed constant of 0. For the receiver, see `./src/core/js/etc/fileTransfer.js`, line 255, inside `Cryptocat.fileHandler`. (The sender and receiver must use the same *IV* for decryption to succeed. The *IV* is not transmitted, and instead both parties rely on this implicitly known value. The sender *IV* code results in the same values, and we only refer to the receiver code in this analysis.)

Files are transmitted in *chunks* of sequential bytes, and each chunk is encrypted separately. The *IV* parameter, which begins at 0, is incremented for each chunk, as seen in *./src/core/js/etc/fileTransfer.js, line 200*. The actual *IV* passed to the encryption library is padded using an *OTR* convention as seen in `./src/core/js/lib/otr.js`, line 298 and related functions, so this chunk index is encoded in the high 8 bytes. The low 8 bytes are initialized to 0 and the low 4 bytes incremented by the *AES* implementation for each block. This means that the same block index within the same chunk index is encrypted with the same counter value (which is correct behavior for *CTR* mode).

The secret key used for file transfers is derived during *Diffie-Hellman* session initialization in `src/core/js/lib/otr.js`, line 2163. A new *DHSession* is initialized whenever there is a "round trip"

of *OTR* data packets — client A sends a data packet to client B and then B sends a data packet to A (see `otr.js` line 2338). This sequence of events is not guaranted to occur between file transfers, so *DHSession* rotation cannot be relied upon to prevent this loss of confidentiality. In particular, the server could suppress messages after a file transfer that would otherwise cause a key rotation.

Vulnerability Description: Reuse of the same *IV* with counter-mode stream encryption reveals the *XOR* of the plaintext of two or more messages, given only the ciphertexts of those messages, as follows:

The ciphertext generated by counter-mode is an *XOR* of a span of plaintext with the output of a block cipher. The block cipher input is the secret key and a block *counter*:

```
ciphertext[j] = F(key, counter[j]) ^ plaintext[j]
```

Here `F` is the block cipher, and `j` is the block number. The first `counter[0]` is derived directly from the *IV* and subsequent `counter[j+1]` values are derived directly from previous `counter[j]` values.

If the same counter and key values are ever used on different plaintexts over the entire lifetime of the key, then the *XOR* of the associated plaintexts can be recovered. Suppose `A` and `B` are two blocks of plaintext, and `A'` and `B'` are the associated ciphertexts, then:

```
A' ^ B'
= ( F(key, C) ^ A ) ^ ( F(key, C) ^ B )
= ( F(key, C) ^ F(key, C) ) ^ ( A ^ B )
= 0 ^ ( A ^ B )
= A ^ B
```

This reduces the security to that of a "running-key cipher", which is easily broken.

Issue B. Integrity Key and IV Reuse in File Transfer

Reported: 2013-11-06

Synopsis: Re-use of the *MAC* key potentially allows files to be modified during file transfer, without detection by the receiving client.

Impact: When keys are reused due to the vulnerability described in Issue A, it is possible to "splice" ciphertext chunks between transfers of files with the same number of chunks, without invalidating the *MAC* tag. This gives an active attacker a limited ability to manipulate the contents of files in flight, under certain conditions.

Attack Resources: This is an active attack requiring modification of *HTTP* requests (or the *BOSH* and *XMPP* protocol traffic embedded in those *HTTP* requests).

Feasibility: This loss of integrity occurs when more than one file is transferred through the file transfer feature during a single *Diffie-Hellman* session, and the files have the same number of 64511-byte chunks.

A compromise is feasible through several vectors:

- A server compromise (including malicious insiders) could allow modification or live update of the server code.

- Where *TLS* is deployed (including the production *crypto.cat* service), an active *TLS* attack against *TLS* protected servers could be used to leverage this attack.

- Where *TLS* is *not* used, this attack is feasible at an unknown number of routers on the Internet.

Verification: We verified this issue by source code inspection. The experiments performed for Issue A also support the conclusion that *MAC* keys are reused between file transfers, provided that there has been no key rollover.

Implementation Analysis: *HMAC* with *SHA-256* is used to protect the integrity of the file contents. Each chunk of the file is transmitted with a *MAC* computed on the following fields:

```
the chunk number (rcvFile[from][sid].ctr)
the total number of chunks (rcvFile[from][sid].total)
the chunk contents
```

Due to the key reuse vulnerability described in Issue A, it is possible for two files to be sent using the same *MAC* key (the encryption and *MAC* keys are both derived from the `extra_symkey` created in *Diffie-Hellman* session initialization). When this happens, a *MAC* tag that is valid for a given chunk of one file, will also be valid for the same chunk of the other file, provided that the total number of chunks is the same.

This allows chunks to be "spliced" between the files, violating the expected integrity guarantees.

The requirement that the files have the same number of chunks cannot be considered unlikely; for example, it is common for files to have length <= 64511 bytes, and files that are revisions of the same document are also likely to be similar in length.

The same comments on *DHSession* rotation as for Issue A apply to this issue.

Mitigation for Issues A and B

Reported: 2013-11-06

Live Mitigation: We recommend these immediate mitigations to protect existing live users:

- Notify users that file transfer may lose confidentiality and integrity, and that users with a low risk tolerance should not transfer files using *Cryptocat*.

- Simultaneously, distribute a new stable release of *Cryptocat*, with version number 2.1.16, which disables the file transfer feature (both send and receive), and has no other changes compared to 2.1.15.

We recommend doing these steps right away instead of hurrying to publish an improved file transfer feature, because:

- It protects users, although admittedly it also inconveniences them.

- It delays publishing details that attackers could use to exploit users. This potentially protects users more than if we simultaneously inform attackers of how to exploit users of the old version at the same moment as announcing to users that they should stop relying on the old version.

- Any other mitigation deserves careful analysis before implementation and deployment.

- Future design and mitigation changes may benefit from other findings in this audit. If we immediately patch one problem, only to later discover another vulnerability with an unexpected relationship to the patch, that effort may be thwarted.

Remediation for Issues A and B

Reported: 2014-01-26

Design and Implementation Mitigation: We recommend that after the *Live Mitigation* steps recommended above, the *Cryptocat* team perform the following steps:

- Update the file transfer feature to be secure, for a future release of *Cryptocat*.

- Simultaneously with committing this patch to a publicly-readable source code repository (i.e. github), publish a document (e.g. blog post) describing the details of the vulnerability and how the fixed version avoids it.

We are willing to help design a future file transfer protocol, if desired.

Versioning

One valuable feature of a new file transfer protocol would be *versioning*. Ideally, if one of the two endpoints is running code with the new file transfer protocol and the other is running code with file transfer disabled (e.g. version 2.1.18) or with the old file transfer protocol (version $\leq$ 2.1.15), users will get a graceful failure —such as the file-transfer option being disabled in the UI along with an explanation that the other peer is using too old of a version— instead of an ungraceful failure such as silent failure, or an error message that could frighten or confuse a user.

Similarly, it would protect users of older *Cryptocats* if their *Cryptocat* would not *attempt* to send files to a newer *Cryptocat*, because doing so could expose the contents of their files even if the newer *Cryptocat* will not accept the transfer, so the new protocol could be designed to prevent older *Cryptocats* from attempting to send to it.

We do not currently have a specific protocol in mind to accomplish such versioning, but would be willing to help try to design one.

Generate per-file keys

The *OTR* protocol provides a secure shared symmetric key (called the "extra symmetric key" in the `otr.js` source code), but if *Cryptocat* is going to send *multiple* files, it can't just use that key, but needs a unique key or *IV* for each file.

One way to accomplish that would be to use a Key Derivation Function (*KDF*). A *KDF* (e.g. HKDF) can be used as a function that takes two inputs: secret key and diversifier, and returns a new secret key.

Cryptocat could use the "file identifier" (typically called the "filename" in the source code and protocol) as the diversifier. So, let current encryption key —the one stored at index 0 in the `key` object in the `files` hashtable in `fileTransfer.js`— be called the "master encryption key", and let the current *MAC* key —the one stored at index 1 in the `key` object— be called the "master *MAC* key". Then:

```
file Enc key = KDF(key=master Enc key, diversifier=file identifier)
file MAC key = KDF(key=master MAC key, diversifier=file identifier)
```

Then, the rest of the 2.1.15 *Cryptocat* file transfer protocol could be used as-is, but using the file-specific Enc and *MAC* keys instead of the master Enc and *MAC* keys for encryption and *MAC* respectively.

For this approach to be secure, the diversifier does not need to be confidential, but does need to be unique within the scope of a given master key. The file identifiers in the 2.1.15 *Cryptocat* protocol are random 128-bit values, so they can be relied on to have this uniqueness property. (The master encryption key and master *MAC* key, of course, need to be confidential, just as in the *Cryptocat* 2.1.15 file transfer protocol.)

New standard file transfer protocol

A longer term strategy is to promote and adopt a file transfer standard in the wider secure protocol community. We recognize that the *Cryptocat* team has already solicited feedback from this community on the *OTR* development list, and we hope to advocate for more review and collaboration on this protocol feature.

Issue C. Substitution of File Contents By Hijacking Entry in User Interface

Reported: 2014-01-26

Synopsis: The file transfer feature uses a sender-supplied identifier to index into the receiver's display of received files.

Impact: The attack targets a file transfer from a "victim sender" to a "victim receiver". An attacker can replace that file with another file when the victim receiver tries to download it.

Attack Resources: In order to succeed with high probability, this attack requires passive monitoring of *HTTP* requests (or the *BOSH* and *XMPP* protocol traffic embedded in those *HTTP* requests), in order to find the `sid` of the targeted file transfer. The attacker, acting as another client in the same conversation, must also send their replacement file to the victim receiver.

Feasibility: Passive monitoring of the *XMPP* protocol traffic is feasible through several vectors:

- A server compromise (including malicious insiders) could allow live traffic sniffing,

- Where *TLS* is deployed (including the production *crypto.cat* service), a *TLS* attack against *TLS* protected servers could be used to leverage this attack,

- Where *TLS* is *not* used, this attack is feasible at an unknown number of routers on the Internet.

An attacker can be assumed to know the conversation name, since that is available by the same passive monitoring. Therefore, they are able to send their replacement file in the same conversation.

The replacement file must be sent after the `sid` is known, and its transfer must complete before the targeted file transfer. This is straightfoward if the replacement file is smaller than the targeted one or the attacker has higher bandwidth than the victim sender. Alternatively, since the `sid` increments by one for each unique ID used in a session, it can be guessed in advance of a file transfer, allowing the attacker to start their transfer before the targeted one.

The attack depends on the victim receiver having a one-to-one chat window to the victim sender open at the point when the attacker's file transfer completes. If the victim receiver is expecting to receive a file from a given sender then it is quite likely they will have that buddy's window open.

The victim receiver will also receive a notification that the attacker has sent a file. The attacker can cause the notification to disappear too quickly to be seen, by logging out the buddy used for the attack just after the transfer completes.

Verification: We verified this issue by source code inspection and by experimentation. To simulate the attacker being able to eavesdrop the `sid` field, the experiments were performed using a modified version of *Cryptocat* that forces the `sid` to zero for every file transfer, by changing the `Strophe.Connection.getUniqueId` function.

Implementation Analysis: The `sid` identifying each file transfer is obtained from `Strophe.Connection.getUniqueId`, which starts at a random integer between 0 and 99999 inclusive, and increments by one on each call. (Unique IDs are also used for other purposes that are not relevant to this issue.) While a file transfer is in progress, the user interface element in the receiver's chat window with the sender has a `file=` attribute referencing the transfer's `sid`. This attribute is used by `Cryptocat.updateFileProgressBar` to update the progress bar, and by `Cryptocat.addFile` to replace the progress bar with a download link when the transfer is complete. (It is also used by `Cryptocat.fileTransferError` to signal an error.)

The attack depends on the following code in `Cryptocat.addFile`:

```
$('[file=' + file + ']').replaceWith(fileLink)
```

which performs a global replace of any currently displayed items having the specified `file=sid`, with the *HTML* of the new download link given by `fileLink` .

Since this is a global replace (see the documentation for replaceWith in *jQuery*), it affects all file transfer elements with the same `sid` that are curently visible. Suppose that the victim receiver's client is showing the one-to-one chat window for the victim sender, but `Cryptocat.addFile` is called for a different file transfer (in another chat window) from the attacker to the same receiver. Then, the UI element in the chat window for the victim sender will be incorrectly updated to link to the attacker's file.

Because the *HTML* of the file download link does not contain the `file=` attribute, any subsequent updates of that UI element (including the one that would normally cause it to link to the correct file when the targeted transfer completes) will be ignored.

For the same reason, the progress bar may also be incorrectly updated, causing it to "bounce" between the values for the two transfers with the same `sid`. Similarly, a file transfer error may cause the wrong UI element to be updated.

Live Mitigation: The Cryptocat developers have already (since 2013-11-29) released a version of Cryptocat (v 2.1.16) with file transmission disabled, so issue is already mitigated.

Remediation: Ensure that `sid` values are always scoped to a particular buddy (i.e. scoped to a particular chat window). `sid` values are chosen by the sender under normal operation but could also be chosen by the server (if the server were malicious or had been confused or compromised by an attack), and `sid` values cannot be assumed to be unique in any scope other than the buddy that ostensibly sent the sid. In fact, per strophejs issue 35, `sid` values in the future are all going to start at 0 and increment, and then restart at 0 if that buddy disconnects from and reconnects to the server.

In particular, the global `replaceWith` in `Cryptocat.addFile` can be removed.

Issue D. File Name, Mimetype, and Size Lack Confidentiality

Reported: 2014-01-26

Synopsis: The file transfer protocol relies on the SI File Transfer *XMPP* protocol extension to transmit file metadata prior to transfer. These metadata are transmitted outside of *OTR* and lack its confidentiality features.

Impact: An eavesdropper may discover filenames, *MIME* types, and sizes of transferred files. The privacy impact of this exposure is quite limited:

- Filenames are currently randomly generated for each transfer and not associated with the original source file.

- *MIME* types are restricted to images and zipfiles, so attackers only learn which of these two categories a file is in.

- Sizes are in exact bytes, but the size is revealed in any case by the length of the file ciphertext.

Feasibility: An attacker must have access to the *XMPP* traffic (or the container protocol traffic: *BOSH* and *HTTP*). This access is possible through several vectors:

- A server compromise (including malicious insiders) could allow live traffic sniffing to recover these messages.

- A server host compromise may grant access to log files containing recorded messages.

- Browser logs or caches may contain recorded messages.

- Where *TLS* is deployed (including the production *crypto.cat* service), an active *TLS* attack against *TLS* protected servers could be used to leverage this attack.

- Where *TLS* is *not* used, this attack is feasible at an unknown number of routers on the Internet.

Verification: We verified this issue by using Chrome's developer console to view the IBB messages transmitted over HTTPS, for example:

```
<body rid="3814497183" sid="e58708cb6438d43ffe6732dcd3ff855f1d690978"
      xmlns="http://jabber.org/protocol/httpbind">
  <iq id="3075:si-filetransfer" to="cryptocataudit@conference.crypto.cat/daira"
      type="set" xmlns="jabber:client">
    <si id="3074" mime-type="application/zip"
        profile="http://jabber.org/protocol/si/profile/file-transfer"
        xmlns="http://jabber.org/protocol/si">
      <file name="229f1c684a5324e50fd5c03b996f8d87.zip" size="158"
            xmlns="http://jabber.org/protocol/si/profile/file-transfer"/>
      [...]
    </si>
  </iq>
  [...]
</body>
```

Suggested Remediation: In any newly designed file transfer protocol, ensure that metadata is encrypted.

Issue E. You Log Out, Attacker Logs in with the same Nickname, Your Friend Thinks The Attacker is You

Reported: 2014-01-26

Synopsis: *Cryptocat* uses an identification model in which a client that knows the name of a channel is able to log in to that channel and claim any unused nickname.

Users may believe that an attacker using a given nickname is the same party as the previous user of that nickname.

This risk is intended to be mitigated by the use of the Socialist Millionaire Protocol (*SMP*). Within a pairwise session between two users, it is possible to use the *SMP* to verify shared knowledge of a prearranged secret. However, a pair of users who wished to authenticate all of their communications would need to repeat the *SMP* on *every* pairwise session between those users.

(A "pairwise session" in this sense ends when either user logs out. It is not the same as the *Diffie-Hellman* sessions involved in Issues A and B.)

This issue could be exacerbated if the attacker observes the session status, for example, an attacker could watch the status of the (encrypted) conversation between Alice and Bob, then see that Bob has logged out, then log in and choose the nickname "Bob", then initiate a conversation with Alice and say "One more thing…". The timing of the initiation of the new session, and the natural-sounding "One more thing…" would trigger Alice's social response to a resumed conversation and may make her forget to question whether this is a different user. This is an example of using social engineering as part of an attack.

Another way this issue could be exacerbated is if the attacker can force a user to disconnect. If the attacker controls the Cryptocat server, can Man-In-The-Middle the *HTTP(S)* connections between the clients and the server, or can use a Denial-of-Service attack on one of the clients, they can cause a disconnect. For example, an attacker could observe an (encrypted) conversation in progress, force one party to disconnect from the Cryptocat server, log in and choose the nickname that party was previously using, establish a session with the other party, and then say "Sorry. What were you saying?".

This issue is exacerbated by the presentation of login messages. Suppose that Alice performs an *SMP* verification with Bob; then Bob logs out and someone claiming the nickname "Bob" logs in. Alice's client will display this to her as "+ Bob", but there is no assurance that this is the same Bob.

Using *SMP* to gain assurance of the identity of the counterparty is inconvenient. Each run of the protocol requires 6 mouse clicks and entry of the secret question and answer from the initiating user; the responding user needs to answer the question and make 2 mouse clicks. The initiating user gets an indication that the protocol succeeded, but the responding user does not. Therefore, mutual authentication requires at least 8 mouse clicks in each session from both users, plus one entry of a secret question and two entries of a secret answer from both users, plus any out-of-band communication and thought needed to agree on the question and answer. This assumes that the authentication succeeds in both directions first time, and does not need to be retried.

After the initial dialog in the initiator's client confirming that *SMP* has succeeded, there is no indication in the *Cryptocat* user interface that a successful *SMP* run has been completed with a given user.

Note that temporary network outages may also cause users to log out and then in again. Under some conditions, this could be sufficiently frequent to make it impractical to run *SMP* on each pairwise session.

Verification: Observations of the *Cryptocat* user interface during experiments in which different clients log in with the same nickname.

Suggested Mitigations/Remediations:

- Change the login message to "Someone logged in and chose to be called 'Bob'."
- Distinguish nicknames of users that have completed *SMP* in the current pairwise session.

- Try to reduce the inconvenience of performing *SMP*. For example, in principle it should be possible to achieve mutual authentication with a single run of *SMP*. (A complicating factor is that the initiating user is able to choose the question, which may give an attacker an advantage.)

- Consider changing the identification model to give clients more persistent keys. This would allow implementing the option for a user to "pin" a nickname to a given public key. For privacy reasons this would need to be an explicit user action, and it would need to be possible to delete pinnings.

- Consider assigning random nicknames every time on join. This is done by Google to manage a similar identification issue — unauthenticated users connecting to a shared resource (anonymous animals in Google Drive). This might fit in well with *Cryptocat*'s branding; *users could be assigned cute cat names and icons*!

- Consider preventing the same nickname from being reused with a different public key for some timeout period. (This would occasionally cause false positives, e.g. if a user reloads Cryptocat and it generates a new key pair, they would have to pick a new nickname temporarily.)

We also suggest performing a user study to investigate the assumptions that users have about the current interface and any intended changes.

Issue F. Nicknames Can Be Invisibly Reassigned

Reported: 2014-01-26

Synopsis: A user may see no sign at all of an Issue E attack: in the window for one-to-one chat with a specific buddy, there is no indication when a buddy has logged out. Therefore, if a user is looking at the one-to-one chat, there is no way for them to know that the session for which *SMP* succeeded has ended. In fact, an attacker may be able to force it to end.

A suspicious and diligent user could discover the reassignment by switching back to the main-conversation window and scanning the conversational transcript for the relevant notifications of their buddy parting and joining, potentially buried among chitchat and other events.

There is potentially also an audio notification when any buddy joins or leaves, but this is not specific to a particular buddy, and may be switched off or otherwise not audible.

This vulnerability could also potentially allow an attacker to get away with performing a Man-In-The-Middle attack that is interrupted just during an *SMP* protocol run, in order to allow *SMP* to succeed.

Verification: Observations of the *Cryptocat* user interface, confirmed by source code inspection.

Implementation Analysis: In `cryptocat.js`, `buddyNotification()` tests for 'main-Conversation' and shows the change of status only in that case.

Suggested Remediation: Show the status changes as they occur, and also when the user returns to the main conversation.

Issue G. Capture of Sent Messages by Nickname Change

Reported: 2014-01-26

Synopsis: An attacker is able to break the confidentiality of a one-to-one chat, by diverting the destination of outgoing chat messages and replacing the key used to encrypt them. The attack involves use of nickname-change *XMPP* messages, which are not sent by *Cryptocat* but are acted on when received.

Impact: After the diversion, the attacker receives and is able to decrypt messages sent by the victim in the chat. The original recipient(s) do not receive these messages. The sender(s) see no indication that their sent messages have been diverted; however, they will not receive any further messages or files in that chat from the original buddy or from the attacker.

For example, suppose Mallory is the attacker and is targetting a one-to-one chat between Alice and Bob. Mallory first sets up a chat with Alice (this need only last a short time), and then sends a specific nickname-change message to Alice's client. After that point, Alice's messages sent to Bob will instead go to Mallory, encrypted using the keys established between Alice and Mallory (and so readable by Mallory). Alice will receive no further messages in that chat.

The attack can optionally also be performed in the other direction, causing Bob's messages to Alice to instead be sent to and readable by Mallory. In that case Bob will receive no further messages or files from Alice in the chat.

This attack cannot be used to gain the contents of files. After this attack, files will *not* be sent encrypted under Mallory's key. Instead files will not be sent at all after this attack.

Feasibility: This attack requires only sending the victim (Alice) an *XMPP* presence message indicating a nickname change. An *XMPP-BOSH* server (operating over *TLS* or not) will typically relay such messages without modification, and we have verified that the production *crypto.cat* service does so. Therefore, the attack can be performed by any client that knows the conversation name.

Verification: This issue was verified by source code analysis and by experimentation. See Appendix C: Exploit Code for Issue G.

Implementation Analysis: *Cryptocat* clients do not support changing their nickname once logged in. However, the *XMPP* protocol does support this functionality. A "nickname-change message" is a special case of an *XMPP* presence message using status code 303, such as:

```
<presence xmlns="jabber:client" from="ccaudittest@conference.crypto.cat/mallory"
          to="271647829313905965259042@crypto.cat/854194958139059658455998">
  <status code="303"></status>
  <item nick="bob"></item>
</presence>
```

The *Cryptocat* code has a `changeNickname` function that is intended to respond to such messages, indicating changes of nickname by other *XMPP* clients. This function has an exploitable flaw: it does not verify that the new nickname is not already being used. So if Mallory starts a chat with the victim Alice, and then changes his nickname in that chat to Bob, then the Alice↔Mallory connection replaces the Alice↔Bob connection, overwriting its keys. However, due to an implementation detail of the message handling code explained below, the callbacks for the replaced connection still reference Mallory's nickname. Therefore, further messages sent by Alice to Bob are actually (conveniently for the attack) relayed to Mallory.

The reason why the message callbacks still reference Mallory's nickname is that they close over the original nickname when created, and `changeNickname` does not affect these closures. For example, the handler function created by `otrIncomingCallback(buddy)` at line 130 of `cryptocat.js` closes over the `buddy` argument in its lexical scope, which is always the original nickname; similarly for `otrOutgoingCallback(buddy)` at line 142. In the case of the outgoing callback, this helps the attack by routing messages to Mallory. In the case of the incoming callback, it hinders the attack by preventing Mallory from sending messages to Alice that would be interpreted as coming from Bob; instead such messages would be added to the Alice↔Mallory chat, which no longer exists.

The bug of closure over the original nickname has other effects in the UI, leading us to suspect that `changeNickname` has never been tested. We have verified that the following procedure is sufficient to work around these effects to reproduce the attack:

1. Mallory starts a chat with Alice as normal.

2. Mallory shows the "Display Info" window for Alice, or simulates the effects of doing so. This step is needed to avoid an incidental bug that is triggered when the nickname change occurs before Mallory has sent any message in the Alice↔Mallory chat. It also has the effect of causing the buddy entry for Mallory to disappear "cleanly" in Alice's user interface when the nickname change occurs — whereas if Mallory sent a message, Alice would receive a flashing notification in her buddy entry for Mallory that would not disappear immediately.

3. Mallory sends a nickname-change message to the server using the code in Appendix C: Exploit Code for Issue G. The `to` field of this message is given by his own *JID* ending in `/mallory`, the status code is `<status code="303"/>`, and the new nickname `bob` is specified using `<item nick="bob"/>`. Note that Mallory appears to be sending a message to himself, but the `to=` and `from=` fields get swapped (we do not know why), and so Alice receives a message with `from=` field ending in `/mallory` as required.

4. Mallory now receives Alice's messages to Bob exactly as though they had been sent to him.

Suggested Remediation: Remove the `changeNickname` handler.

Issues Without Known Exploits

This section describes issues for which we have not discovered an exploit in the current *Cryptocat* use. These issues could become exploitable when other code changes, so they represent some potential future security risk.

CTR-mode Overflow

The CTR mode implementation in `mode-ctr.js` fails to carry when the increment of the least-significant word overflows. This means a re-used counter and confidentiality leak for messages longer than 2^32 blocks, which is 2^36 bytes for *AES*, with its 16-byte blocks. This is not exploitable in *Cryptocat* because message lengths are always shorter than this. (The file transfer chunk size is (2^16 - 1025) bytes.)

Future Work

Protocol Analysis, Design, and Implementation

Cryptocat is pushing the boundaries of *usable, secure, and multiparty* messaging. By dint of this innovative niche, it would benefit as much or more from security-aware protocol analysis and design collaboration as it does from code auditing and penetration testing.

Multiparty Chat

A key area of unresolved issues is the group-chat protocol design and related security features. The current *multiparty chat protocol* is an in-house design and would benefit from protocol specification refinement, design analysis, and potential design changes.

In terms of product engineering, our intuition is that replacing this protocol with a community-developed standard will lead to better security in less time. This of course depends on such a community-developed standard emerging.

Unless a community standard emerges very quickly, it is still valuable to improve the security of the *multiparty chat protocol*, and we recommend this general roadmap: First, review the existing specification to empirically determine which security properties it provides, noting ambiguity when present. Second, rewrite the specification to follow from those properties (in contrast to describing the procedures or data formats). Third, separate out the procedures and data formats from the abstract protocol and its security goals. At this point solicit more scrutiny from the community. This work can also contribute to the development of the community standard alluded to above.

File Transfer

Like the *multiparty chat protocol*, the file-transfer features of *Cryptocat* are developed in-house as integrated extensions to both *XMPP* and *OTR*. We suggest that file transfer not be reenabled until a more secure protocol is available; our suggestions for such a protocol are described in Remediation for Issues A and B.

OTR & Cryptographic Libraries

This audit did not focus on the *OTR* implementation, nor the cryptographic libraries used by *Cryptocat*. While we examined these dependencies as necessary for our investigations, these would benefit from focused, targeted audits.

JavaScript Cryptography

There are open unresolved issues with respect to *JavaScript*-based security applications. These are probably more relevant for security research rather than security audit work, but sometimes the lines can be blurry.

Two areas which concern us are delivery and verification of the *Cryptocat* add-on, and side-channel analysis.

Open Questions & Concerns

- Conversations with guessable room names can be "burst-in-on". Does the documentation say to use unguessable room names? (Note that room names are always known to the server.)

- The *OTR* protocol appears to allow an attacker to force messages to be selectively dropped. It protects against message reordering, but forcing a message to be dropped will not prevent subsequent messages from the same buddy from getting though, and will not cause any warning. Verification: reading the OTR v3 protocol specification, and the `(ctr <= sessKeys.rcv_counter)` check in the `handleDataMsg` function from `otr.js`.

- We did not check for the possibility of downgrade attacks to *OTR v2*. (Both OTR v2 and OTR v3 are enabled by default, and this default is not overridden by Cryptocat.)

- We tried to determine whether an attacker exploiting issue F could also prevent the notification in the main conversation window, but were not able to establish whether or not that was possible.

- There may be the potential for inconsistent state between the user interface and the global `currentConversation` variable if there is an exception in `switchConversation`.

- "Major new feature: *Cryptocat* now automatically reconnects to conversations when disconnected, without troubling the user. *Cryptocat* will automatically detect accidental disconnections and wait for the Internet connection to be re-established before reconnecting." (from CHANGELOG.md) Does that mean it is now possible for an attacker to edit out parts of a conversation without troubling the user?

- Question: If the *IV* parameter passed to the `crypto-js` library has the wrong type or is not long enough, `undefined` propagation could compromise confidentiality. What happens if a non-Array general object is passed as *IV*? What happens if a short Array is passed? We suspect this could be a disastrous, silent security failure. We manually inspected every call site, and performed some live tests with assertions within *Cryptocat* proper to gain confidence that the *IV* is the right type, size, and has the correct element types and range.

- Concern: There is a type-dependent return from `Cryptocat.getBytes()` which actually causes calls to `Cryptocat.encodedBytes(1, ...)` to throw an exception.

- Concern: `Cryptocat.fileKeys[nickname]` is used for transfers in both directions. Is this a problem?

    - What happens if Alice begins receiving $FILE from Bob, then initiates a send to Bob?

- `.position > .file.size` seems off-by-one (if `position = .file.size` then the file has all been sent already)

- Why does it use `FileReader` and `readAsDataURL`? This should be documented in a comment. (My guess: the data is in a string of Unicode chars and needs to be converted to a sequence of bytes, and the Unicode-encoding way of doing it is inefficient on chunks this large.) Why not use `readAsArrayBuffer`?

- `otr.js` ignores the *OTR TLV* type 8 4-byte type indicator and assumes it is a filename. This might break compatibility with a future *OTR* standard.

- Why are only certain types (*MIME* types) of files allowed? This should be documented, for example on a web page, wiki, or text file, and the code that enforces that restriction should have a comment saying where to find the documentation of it. Perhaps in https://github.com/cryptocat/cryptocat/wiki/OTR-Encrypted-File-Transfer-Specification, or perhaps a more user-focused manual.

- `Cryptocat.fileSize` should be named `Cryptocat.maximumFileSize`.

- The `seq` parameter in the file-send protocol is maintained in the sender, received by the receiver, and stored by the receiver in the `rcvFile` structure, but is not actually used for anything. Remove all uses of it (since *IBB* protocol requires a `seq` parameter to be sent in the `data` message, but *Cryptocat* doesn't use that parameter, just hard-code it to 0).

- `strophe.js getUniqueId` is documented as resetting to 0 for each connection, but it actually resets to a random integer from [0,10000). Opened ticket https://github.com/strophe/strophejs/issues/35. The ticket was closed by the strophe authors by setting the `uniqueId` to 0. This affects Issue C. Substitution of File Contents By Hijacking Entry in User Interface.

- As documented in Appendix B: Work Log, we concluded that BOSH is resistant to CSRF attacks provided that the `sid` parameter is unguessable. Much later, we realized that the `sid` parameter is not unguessable. Is there anything else protecting BOSH from CSRF attacks?

Recommendations

Coding Practices

- The *Cryptocat* implementation guards against *XSS* attacks by storing potentially attacker-controlled data, such as nicknames, as strings and escaping them close to the point of use. This in practice results in escaping logic being scattered in many places over the source, including *Mustache* templates as part of the source; if any one of the necessary places is omitted, there may be an *XSS* vulnerability. If instead such data were held in an object that is not usable directly as a string, it would be much easier to ensure consistent and auditable validation and escaping.

  (In *JavaScript* all objects have implicit coercions to string; however, the implicit coercion may yield a harmless constant, in which case it is not a security risk for it to be invoked accidentally.)

- A common idiom in *Javascript* code is for a function to behave differently depending on the type of its arguments. This can make it harder for reviewers to correctly trace control flow (as they might misinterpret or misremember which of the behaviors of the function will be executed in a certain case), and can similarly lead developers to call the function incorrectly. Some of *Cryptocat*'s dependent libraries use this idiom. We would recommend to the authors of *those* libraries to instead write separate functions for each separate behavior.

  Examples:

  - `OTR.prototype._sendMsg` in `otr.js` (from the otr.js codebase), which does something different if its second argument is true.

  - `OTR.prototype._sendMsg` in `otr.js` also does something different if its `msgstate` is `CONST.MSGSTATE_PLAINTEXT`.

  - `selectCipherStrategy()` in `cipher-core.js` (from the crypto-js codebase) is scary, because what it does depends on whether the type of its `key` argument is string or other.

- Cryptocat itself uses in one place a similar idiom, of returning different types of argument from a function in different cases. This is in `Cryptocat.getBytes()`, which returns different types depending on whether its first argument is 1 or a number greater than 1. As mentioned in Open Questions & Concerns, `Cryptocat.encodedBytes()` doesn't take into account the fact that `Cryptocat.getBytes()`'s return value is of varying type, so if you invoke `Cryptocat.encodedBytes(1, …)`, it will throw an exception. We recommend making each function return the same type of object in all cases.

- The `crypto-js` codebase makes heavy use of a `.extend` prototypical inheritance by copy-then-modify. Additionally it has a very deep abstraction hierarchy for only a few actual ciphers and modes. These two styles make it extremely cumbersome to audit by source.

- The `key` structure which has two slots, 0, and 1, should instead be a struct with named slots. Recommendation: Use named properties rather than fixed Array indices (tuple-style), or if tuple style has some advantage, define constants for the indices, rather than using magic constants.

Appendix A: The life cycle of the Cryptocat file transfer

Here are our notes describing our understanding of the Cryptocat file transfer protocol, along with the parts of the rest of the protocols that are necessary to evaluate the security of the file transfer protocol. This is described in chronological order of one (or more) file transfers.

1. The server tells a client there is a Presence session, with a Nickname.

   Note: there is no attempt to enforce constraints on what Nickname gets used, other than that it can't be currently in use (see Issue E. You Log Out, Attacker Logs in with the same Nickname, Your Friend Thinks The Attacker is You).

2. Now the server can deliver messages between clients, which *OTR* uses to do its protocol, resulting in a *Diffie-Hellman* shared secret.

   Note: if the user does not perform the optional Socialist Millionaire Protocol authentication, then this is vulnerable to a Man-In-The-Middle attack (see Issue E. You Log Out, Attacker Logs in with the same Nickname, Your Friend Thinks The Attacker is You).

   The resulting *OTR* keys are stored in an `OTR` object, which is stored in a hashtable named `otrKeys`, indexed by the Nickname.

3. *OTR* generates a new *DH* shared secret "on every round trip" (see below for precisely what that means). After it generates a new *DH* shared secret, it begins using it to protect all messages that it sends from that time on.

   By "on every round trip" means: after a new *DH* shared secret is generated, then the next *OTR* Data Message sent will contain an advertisement of a new *DH* public key. After that advertisement is received by the peer, then the next *OTR* Data Message that the peer sends will contain an acknowledgement of his receipt of that new *DH* public key. Once that acknowledgement is received by first party, it will begin using the new *DH* public key which will result in a new *DH* shared secret.

4. Whenever a client initiates a file send, then all the following things happen (in order and synchronously) in the *Cryptocat* client on the file transmitter side:

   a. The file transmitter generates a random 128-bit number encoded in hexadecimal, and appends the file's extension. We'll call this the "file identifier", although in the source code it is usually called the "filename".

   b. The `OTR` object sends the file identifier (called a "filename" in this protocol), encrypted and authenticated, through *OTR* (using the current *DH* shared secret), and calls back to *Cryptocat* to deliver an "extra symmetric key" (which is derived by *OTR* from the current *DH* shared secret).

      (See the call to `on('file', …)` in `handlePresence` in `cryptocat.js`.)

   c. The file transmitter diversifies the extra symmetric key into an encryption key and a *MAC* key, and stores the pair of keys (encryption key and *MAC* key) in the hashtable named `fileKeys` under the index of the nickname of the intended file-receiver and then under the index of the file identifier: i.e. if the source code used this terminology, the indexing into `fileKeys` would be written `fileKeys[receiversNick][fileIdentifier]`.

      N.B. The same *OTR*-generated key can be used for multiple file transfers here (see Issue A. Disclosure of File Contents Due to Re-use Of Key and IV and Issue B. Integrity Key and IV Reuse in File Transfer).

   d. The file transmitter then generates an `sid`, which is guaranteed to be unique within the scope of that *Cryptocat* client's current connection to the *XMPP* server.

      N.B. If the client disconnects and reconnects to the *XMPP* server, then subsequently generated sids could collide.

   e. The file transmitter stores the filehandle (giving access to the file on the local filesystem), the nickname of the receiver, the encryption and authentication keys, and a counter in a hashtable

named `files` in `fileTransfer.js`, indexed by the `sid`. (See `Cryptocat.beginSendFile` in `fileTransfer.js`.)

    f. The file transmitter initiates a strophe file transfer, which sends the `sid` and the file identifier (called a "filename" in this protocol) over an unencrypted and unauthenticated protocol. (This message is unencrypted and unauthenticated at *this* layer, not at the underlying client↔server transport layer; i.e. the server is going to see and have the opportunity to manipulate those values, and unless both clients use *TLS* to the server, then other parties will as well.) In the same message, the file transmitter client also includes a file size and *MIME* type.

(See `Cryptocat.beginSendFile` in `fileTransfer.js` and `send` in `strophe.si-filetransfer.js`.)

    g. The file transmitter then deletes `fileKeys[receiversNick][fileIdentifier]`.

5. Now the following events might eventually occur in the intended file-receiver, as caused by some of the network sends in step 4 ("Whenever a file transfer is initiated"), above.

    a. Whenever the encrypted and authenticated file identifier is received over the *OTR* protocol in the receiver, the `OTR` object calls back to the *Cryptocat* code to deliver the file identifier and the extra symmetric key. The *Cryptocat* code in the file receiver diversifies the extra symmetric key into an encryption key and a *MAC* key, and stores the pair of keys (encryption key and *MAC* key) in the hashtable named `fileKeys` under the index of the nickname of the file sender and then under the index of the file identifier: i.e. `fileKeys[sendersNick][fileIdentifier]`.

    b. Whenever the unencrypted and unauthenticated strophe file-transfer message is received in the file receiver, the message comes with a `from` field containing the nickname of the sender, as supplied by the server. The file receiver takes these five fields: from, sid, file identifier (called "filename" in the protocol and in the source), size, and mime-type, and stores them in a hashtable named `rcvFile` in `fileTransfer.js`. They are indexed in `rcvFile` first by `from` and then by `sid`. The strophe implementation sends back an acknowledgement message (a "noop" in the strophe protocol) to indicate to the sender that the *IBB* (In-Band-Bytestream) protocol is supported.

        a. N.B. The file transmitter can choose to send anything it likes for sid, file identifier, size, and mime type. The file transmitter *could* choose an sid that matches an `sid` used by a different peer of the file receiver client, or that matches an `sid` used by the file receiver client, if it chose. (The `sid` could be learned by other peers or by the server, or could even be guessed "blind" if necessary since they have only approximately 10,000 possible values and are generated by `Math.random`.) The file transmitter *could not* choose a file identifier that matches a file identifier used by a different peer of the file receiver client (file identifiers are too large to be guessed and are generated with cryptographic-quality randomness). The file transmitter *could* choose a file identifier that matches a file identifier used by itself previously or concurrently. The file transmitter *could* choose a file identifier used by the file receiver client in a previous or concurrent file-*send* operation in the opposite direction — from the client currently operating as file receiver, if it previously or concurrently sent a file.

        b. N.B. The server can choose anything that the file transmitter could choose (from 5.b.a. above). (There is no end-to-end encryption or authentication to prevent the server from seeing and altering these values as it likes.) In addition, the server can send the `from` field set to whatever it likes (there is not, at this point, any cryptographic authentication showing that the controller of a certain nick sent these fields). In addition, the server knows the exact `sid` used by each client and could send a `sid` chosen to match any of them. In addition, the server knows the file identifiers used by all clients (since the file identifiers are sent unencrypted in the strophe protocol, in addition to being sent encrypted in the *OTR* protocol), so it could choose to send a file identifier equal to any of them.

6. Now the following event might eventually occur in the file transmitter, as caused by the network send from the receiver's strophe implementation in 5.b:

a. Whenever the strophe acknowledgement message from the file receiver arrives (indicating that the receiver is capable of *IBB*), then the file transmitter sends an *IBB* 'open' message, which contains the receiver's nickname, the sid, and the chunksize. This message is transmitted unencrypted and unauthenticated.

   - a. N.B. the file transmitter can choose anything it likes to send for the sid and the chunksize.

   - b. N.B. the server can choose anything that the file transmitter could choose (from 6.a.a above). In addition, the server can send the "from" field set to whatever it likes. As mentioned above, the server also knows the sids of all clients.

   (See `Cryptocat.beginSendFile` in `fileTransfer.js`.)

7. Now the following event might eventually occur in the file receiver, as caused by the network send from the file transmitter's 6.a.:

   a. Whenever an *IBB* 'open' message is received, the client uses the `from` and `sid` values from the message to retrieve the file identifier from the `rcvFile` hashtable, i.e. `fileIdentifier = rcvFile[from][sid].filename`, and then fetches the keys from the `fileKeys` hashtable, i.e. `key = fileKeys[from][fileIdentifier]`, and then stores the keys in the `rcvFile` hashtable, i.e. `rcvFile[from][sid].key = key`, and then deletes the key from the `fileKeys` hashtable, i.e. `delete fileKeys[from][fileIdentifier]`.

   The file receiver client then sends an *IBB* "result" message back to the file transmitter.

   (See case 'open' in `Cryptocat.ibbHandler` in `fileTransfer.js` and `open` in `strophe.ibb.js`.)

8. Now the following event might eventually occur in the file transmitter, as caused by the network send from the file receiver's 7.a., or as caused by the network send from the file receiver's 9.a.:

   a. Whenever an *IBB* "result" message is received in reply to the *IBB* "open" message, the file-transmitter client executes `Cryptocat.sendFileData` (from `fileTransfer.js`) with its `to` argument set to the intended receiver's nick, and its `sid` argument set to the sid. (See `Cryptocat.beginSendFile` in `fileTransfer.js`.)

   b. Whenever the file transmitter's `sendFileData` method is invoked, the file transmitter looks up the file-being-sent's position, filehandle, counter, and encryption key, and total size from the `files` hashtable in the `fileTransfer.js`, under the index of the `sid`.

   The file transmitter then computes the bounds of the next chunk of the file (where a "chunk" is 64,511 bytes long, or shorter if there are not that many bytes left in the file), starting from the current "position", sets the position (on the object in the `files` hashtable under the `sid` index) to the index number of the next byte after the chunk, and increments the counter (on the object in the `files` hashtable under the `sid` index).

   The file transmitter then reads the chunk from disk, and when the chunk is loaded into memory, it encrypts it with *AES-256* in *CTR* mode using the encryption key and counter from the object stored under the `sid` index in the `files` hashtable.

   It then generates a header consisting of the counter value and the total number of chunks and computes a *MAC* over that header plus the ciphertext chunk, using the *MAC* key from the object stored under the `sid` index in the `files` hashtable.

   It then appends the *MAC* tag to the ciphertext chunk and sends an *IBB* "data" message with the data consisting of ciphertext chunk followed by the *MAC* tag. The *IBB* "data" message includes, in addition to the data, a `from` and `sid`.

   - a. N.B. the file transmitter can choose anything it likes to send for the `sid`.

   - a. N.B. the server can choose anything that the file transmitter could choose (from 8.b.a above). In addition, the server can send the "from" field set to whatever it likes. As mentioned above, the server also knows the sids of all clients.

   When/if the file transmitter receives an *IBB* "result" message in response to this send, then it invokes its `sendFileData` method again.

(See `Cryptocat.sendFileData` from `fileTransfer.js`.)

9. Now the following event might eventually occur in the file receiver, as caused by the network send from the file-transmitter's 7.b.:

   a. Whenever an *IBB* 'data' message is received, the client uses the `from` and `sid` values from the message to look up the object in the `rcvFiles` hashtable indexed under the `from` and then the `sid`.

      From that object it reads an `abort` flag, and if that flag is set it returns.

      Next, from that object it reads the encryption and *MAC* keys, counter, and total number of blocks. It generates a header containing the counter and total number of blocks, computes a *MAC* on that header using the *MAC* key, parses out the *MAC* tag from the data from the message, and compares its generated *MAC* tag to the one from the message. If they differ, it sets the `abort` flag on the object indexed under `from` and `sid` in the `rcvFile` hashtable and returns.

      Next, it decrypts the chunk using the key and counter. It appends the plaintext chunk data to an attribute named `data` of the object which is indexed under the `from` and `sid` in the `rcvFile` hashtable, and increments the counter in that object.

      (See the 'data' case in `Cryptocat.ibbHandler` in `fileTransfer.js`.)

Appendix B: Work Log

We checked the Chrome store archive against the git tag and discovered a discrepancy. We created Issue 500 to highlight this issue.

- Part of this issue is that the Chrome store automatically transcodes images which changes the archive contents and hash from what a developer submits. This inhibits an auditor from building a Chrome package to compare against the Chrome store release. We attempted to notify the Chrome store about this issue.

We examined changes in the `otr.js` dependency.

- Discovered `extra_symkey` which is used for file transfer.

We examined `otr.js prepareMsg()` function which is used for both chat messages and to initiate file transfers.

- Things we *didn't* cover: *SMP*-based authentication, the user interface when it fails, the code implementing same.

We examined file transfer thoroughly throughout the `cryptocat.js, etc/fileTransfer.js, otr.js,` and `lib/crypto-js/*.js`.

- We mainly focused on the call stack in the `crypto-js` dependency, rather than analysing the entire codebase, so we focused on the *AES* and *CTR* mode implementations.

    - Note: We have a concern about a potential security failure if *IV* is the wrong type, size, or element type/range when using counter mode in this dependency. We analyzed *Cryptocat*'s current call sites and believe it uses *IV*s of the correct type and size.

- We investigated a concerning counter rollover behavior in `crypto-js` and verified that *Cryptocat* currently will never cause this rollover in file-transfer encryption. However, this is a danger for future development.

    - The restriction on file transfer *MIME* type is described in the UI but not documented or justified in the Wiki Specification of File Transfer.

We investigated the entire source and use of `etc/random.js`:

- Examined how seeds are distributed to web workers: `workers/keyGenerator.js` and `workers/smp.js`.

- We examined the API and call sites to gain assurance that there are not insufficient-entropy flaws due to buffer encodings as have been discovered in the past. The new API makes the encoding much more explicit using explicitly-named encodings. We believe the correct encoding is used at each call site, so we have confidence this kind of flaw is not present in the current codebase.

We also briefly skimmed the complete source to `lib/salsa20.js` to identify any glaring problems, but saw none. We did not thoroughly verify the implementation correctness, such as by comparing test vectors against other implementations, nor did we analyze side-channel issues which may leak secret state.

We examined portions of `lib/strophe/` relevant to understanding interactions between *XMPP*, *OTR*, and *Cryptocat* data structures and event interleaving.

We did minimal analysis on the potential for *XSS* and *CSRF* vulnerabilities:

- We believe *BOSH* is resistant to *CSRF* attacks provided the `sid` parameter is unguessable. Viewing requests and responses in the browser's developer tools suggests this is true. We did not analyze if participants in a group chat can learn sufficient details to create a *CSRF* attack vector against other users. (Also, if this were a problem, it would be common to all *BOSH* implementations. We have not yet investigated if this is a commonly known issue.)

- We performed only a few *XSS* tests, such as injections in the username or chat contents. The former is inconclusive because we did not bypass client-based input-side restrictions that a malicious client could bypass, and the latter was unsuccessful.

- Our investigation into these kinds of web-frontend attacks was not very thorough.

We did investigate `lib/bigint.js`:

- Found and filed a performance bug which we do not believe is security relevant, OTR Issue 41.

- Examined implementation of Maurer's algorithm before realizing it is unexercised in *Cryptocat*.

We investigated the potential for cross-site `postMessage()` abuse against the web workers; however, they are anonymous web workers, and thus protected against this attack vector by *JavaScript* referential semantics.

Investigated authentication, mainly in one-to-one *OTR* chat:

- Filed *Issue 506* about a dubious time-based retry loop for sending public keys.

Experimented with nickname reuse between clients at different times in the same channel.

Appendix C: Exploit Code for Issue G

The following code, triggered on connection to the server, was used to verify Issue G:

```javascript
if (Cryptocat.myNickname == 'mallory') {
    try {
        var delay = 30;
        var newNickname = 'bob';

        console.log('Hacking commences in ' + delay + 's');

        // Cryptocat.xmpp.connection.muc on trunk
        var muc = Cryptocat.connection.muc;
        // Cryptocat.xmpp.conferenceServer on trunk
        var roomPrefix = Cryptocat.conversationName + '@' + Cryptocat.conferenceServer + '/';

        var fromJID = muc._connection.jid;
        var presence = new Strophe.Builder('presence', {to: roomPrefix + Cryptocat.myNickname,
                                                         from: fromJID})
                            .c('status', {code: '303'}).up()
                            .c('item', {nick: newNickname}).tree();
        console.log(presence);
        window.setTimeout(function() {
            console.log('Sending the presence message:', presence);
            muc._connection.send(presence);
            console.log('Sent it');
        }, delay*1000);
    } catch(e) {
        console.log(e, e.stack);
    }
}
```