Report of Initial Security Audit of Crypton

Version 1, 2013-12-20

- Principal Investigators:
  - Nathan Wilcox-O'Hearn <nathan@LeastAuthority.com>
  - Zooko Wilcox-O'Hearn <zooko@LeastAuthority.com>
  - Daira Hopwood <daira@LeastAuthority.com>
- Organization Name:

  Least Authority

Contents

Overview

The Least Authority security consultancy performed an initial security audit of crypton on behalf of SpiderOak. The *crypton* JavaScript framework simplifies creating browser-based applications with privacy and security properties that protect the end user, even in some respects against malicious servers.

This report gives the results of the initial audit.

Audit Scope

The audit was a two-week [*] time-boxed investigation of essential *crypton* security properties and their implementation. The audit techniques included interactive penetration testing, code and design analysis, and developer interviews.

We focused on single-user application security features (those without cross-user interactions), especially those intended to protect users against malicious servers.

[*] It took three weeks to finish this two-week audit.

Scope Limits

A well-known outstanding attack surface that can compromise client-side security of web-based applications, given a malicious server, is to deliver backdoored client code. For the purposes of this audit, we assume a user has received the correct JavaScript code, and the attack surface we examined excludes that issue.

Since this was a short audit, we de-emphasized cross-user interactions and "standard" web attack surfaces, such as *Cross-Site Scripting (XSS)* or *SQL Injection*.

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.

This preliminary audit focused on one particular use case, represented by the *diary* example app. The *diary* app allows a user to append timestamped free-text entries to a personal diary. It is intended to allow that user to read and write their own diary, but not to allow anyone else to read or write into that diary.

Target Code and Revision

This audit targeted the public crypton source code focusing on this revision:

```
6c85ded20421b6872581cfe4cbef3e1c3283963f
```

Findings

Issue A. ElGamal keypair recovery vulnerability

Synopsis: A server can send the client's `base_keyring.keypair_salt` instead of the expected `base_keyring.challenge_key_salt` in the authentication challenge protocol. This causes the client to respond with the symmetric key used to protect the *ElGamal* keypair, which includes the private key stored in the `sec` property of `base_keyring.keypair`.

Impact: Recovery of this private key allows the attacker to read all container contents and forge new records.

Feasibility: An attacker that has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection.

Implementation Analysis: An attacker with access to the `base_keyring.keypair_salt` database field for the victim client, and the ability to modify the challenge `HTTP` response, can use this vulnerability to recover the keypair. An attacker that additionally has the ability to read the `base_keyring.keypair` can follow a recovery of `keypairKey` with a decryption of the client's *ElGamal* private key.

Authentication uses a simple challenge/response protocol where the server sends a challenge salt and the client responds with the `pbkdf2` output of that salt keyed with their secret password:

```
response <- pbkdf2(password, challenge)
```

The client code can be found in `client/src/core.js` inside `crypton.authorize` on line 145:

```
response.challengeKey = sjcl.misc.pbkdf2(passphrase, body.challengeKeySalt);
```

The typical challenge value is initially selected by the client and stored in `base_keyring.challenge_key_salt` in the database.

Meanwhile, the symmetric key that clients use to encrypt their *ElGamal* keypair is also a `pbkdf2` computation with the same password, as can be seen in `crypton.generateAccount` of the same file on line 100:

```
var keypairKey = sjcl.misc.pbkdf2(passphrase, keypairSalt);
```

If a malicious server sends a challenge such that `body.challengeKeySalt` in `crypton.authorize` is identical to `keypairSalt` in `crypton.generateAccount`, then the resulting `response.challengeKey` will be identical to `keypairKey`.

The `keypairKey` value is computed in `crypton.generateAccount` to encrypt the *ElGamal* keypair on line 106:

```
account.keypairCiphertext = sjcl.encrypt(keypairKey, JSON.stringify(keypair.sec.serialize()), crypton.cipherOptions);
```

The same value is computed on the client in `account.js` within `Account.prototype.unravel` and then used to decrypt the *ElGamal* keypair on lines 66-70:

```
// regenerate keypair key from password
var keypairKey = sjcl.misc.pbkdf2(this.passphrase, this.keypairSalt);

// decrypt secret key
var secret = JSON.parse(sjcl.decrypt(keypairKey, JSON.stringify(this.keypairCiphertext), crypton.cipherOptions));
```

By recovering this symmetric key, an attacker may then decrypt `base_keyring.keypair` to recover the original *ElGamal* keypair.

Suggested Mitigation: The immediate problem could be addressed by ensuring that the two uses of the PBKDF have distinct inputs (for example, by prepending a diversification parameter to the passphrase); however, we suggest considering other passphrase-based authentication protocols for which such issues are likely to already have been analyzed.

Issue B. Server Can Overwrite Client's Secrets

Synopsis: A server can send unexpected values in response to a client login, which values will overwrite the client's `Account` object's properties, including its `passphrase`, `keypairCiphertext`, `containerNameHmacKeyCiphertext`, and `hmacKeyCiphertext`.

Impact: The server can forge data to be displayed to the client, including the ability to overwrite any data that was written by the client before the server used this exploit. In addition, the server gains the ability to read all data that is written by the client after the server uses this exploit.

Feasibility: An attacker that has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection.

Implementation Analysis: In `core.js`, in `crypton.authorize` each property from the response body (parsed as JSON) is copied into the `Account` object. A server can therefore send a response body containing the JSON representation of properties named `passphrase`, `keypairCiphertext`, `containerNameHmacKeyCiphertext`, and `hmacKeyCiphertext`.

This causes those properties on the client's `Account` object to be overwritten to contain values provided by the server.

Immediately after copying those properties, the client calls `Account.unravel()`, which uses those four properties to set the values of several other properties: `secretKey`, `pubKey`, `symKey`, `containerNameHmacKey`, and `hmacKey`. Since the server can control the values of the initial four secrets, it can also control the values of those other properties that are set by `Account.unravel()`.

Suggested Mitigation: Separate those fields that should be stored on the server into an object distinct from the client-side `Account` object, and only read the serialization of that object from the server without overwriting client-side fields.

Issue C. Modification of Container Contents By Filtering And Reordering Patches

Synopsis: A server can omit one or more records of the server's choice when delivering a container to the client. It can combine this with reordering the records.

Impact: The server can manipulate the contents of the resulting container. For example, if the client set the initial state of the container to the following JSON structure:

```
{
   "name": "Bob Lawblaw",
   "balance (USD)": "0"
}
```

And then in the second state, the client changed the container to:

```
{
   "name": "Bob Lawblaw",
   "balance (USD)": "10000"
}
```

And then finally the client changed the container to:

```
{
   "name": "Bob Lawblaw",
   "balance (USD)": "0"
   "beneficiary": "Alice Salissa",
   "amount sent to beneficiary": "10000"
}
```

Then the server would be able to reorder the records to apply the last diff and then the middle diff, causing the container to have the final value of:

```
{
   "name": "Bob Lawblaw",
   "balance (USD)": "10000"
   "beneficiary": "Alice Salissa",
   "amount sent to beneficiary": "10000"
}
```

Feasibility: An attacker that has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection, and by running the above example through `jsondiffpatch`.

Implementation Analysis: Crypton always uses `jsondiffpatch` to diff and patch the container contents. The resulting diffs are authenticated-encrypted independently and stored in transaction records. In correct operation, a client that is writing to a container must demonstrate knowledge of the last transaction id applied by the server; in that case, records will not be reordered. However, since there is no end-to-end authentication over the ordering or grouping of diffs, a server can freely reorder or selectively drop diffs without detection.

Suggested Mitigation: We put forward the following mitigation options:

1. Include a hash of the preceding record in each new record, and check this hash when reading.
2. Include a hash of the current container contents in each new record, and check this hash when reading.

Option 1 ensures ordering of the records directly (each record authenticates its predecessor, including the hash of that record's predecessor, and therefore the entire ordering).

Option 2 ensures that the final contents are correct. This does not strictly speaking ensure record ordering (for example, a record that had no effect could be dropped), but an attacker cannot choose an order that results in incorrect container contents.

Both options protect against the effects of reordering by a malicious or compromised server, provided that the implementation of diffing and patching is correct. Option 2 additionally means that any mistake in computing the final container contents when the ordering is correct (for example due to a bug in the `jsondiffpatch` library, or any replacement for it), would be detected. We believe this is a significant advantage since the possibility of such a bug is difficult to discount.

It is also possible to use both options, i.e. include and check both hashes. If information about the ordering of records might be used in addition to the final container contents, this would be preferable.

Note that a server will not be prevented from truncating the list of applied records, i.e. rolling back to a previous state. We believe this limitation is an inherent consequence of the fact that the client does not store long-term state. It should however be documented so that users of the framework are aware of it.

Issue D. Guessable Private Keys On Browsers Without crypto.getRandomValues()

Synopsis: A client will generate an elliptic curve private key without sufficient entropy, unless the browser provides the `crypto.getRandomValues()` API.

Impact: An attacker who can get a copy of the ciphertext can read the client's plaintext data.

Feasibility: An attacker that has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection.

Implementation Analysis: Crypton `core.js` passes a zero value for the `paranoia` argument to `sjcl.ecc.elGamal.generateKeys()`, which means that on a browser without `crypto.getRandomValues()` (e.g. releases of *TorBrowser* as recent as 2.3.25-15), the private key may be recoverable by an attacker. Although mouse movement events and the timestamp of an `onload` event are used to collect entropy, there is no guarantee that any mouse movements will occur, and the entropy of the timestamp is only a few bits. Provisionally, we believe this is likely to be insufficient to prevent the private key being recovered.

Suggested Mitigation: This issue could be addressed by increasing the `paranoia` argument, but it would be necessary to consider how the case of insufficient entropy should be reported to an end user.

Issue E. Encryption Key Disclosed To Server

Synopsis: A client discloses to the server the symmetric encryption key that protects payloads.

Impact: The server can read and forge all container contents.

Feasibility: An attacker that has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection.

Implementation Analysis: `Account.prototype.serialize` in `account.js` returns an object containing `this.hmacKey`. The return value from `Account.prototype.serialize` is sent to the server (in `Account.prototype.save`). `this.hmacKey` is the key that is used to encrypt and authenticate the payloads, so disclosing it to the server enables the server to read and forge all payloads. An encrypted copy of `hmacKey` is *also* sent to the server, as `hmacKeyCiphertext`.

Note that the client does not read and use the cleartext copy `hmacKey`, but instead gets the key from `hmacKeyCiphertext`, decrypted and authenticated using `symKey`, in `Account.prototype.unravel`.

Suggested Mitigation: Separate those fields that should be stored on the server into an object distinct from the client-side `Account` object, and send only the serialization of that object to the server.

Coverage

The Appendix A. Work Log contains our investigation process which clarifies the current coverage.

Functional Areas

As of this report revision, we've analyzed only the cryptographic design and implementation related to authentication, account generation and access, data confidentiality, and integrity issues. Additionally we've done some black-box examination of the `HTTP` protocol level.

Areas with Less Coverage

We did not comprehensively examine the *crypton* framework outside the parts that the *diary* app directly relies on, nor the sjcl library (except inasmuch as necessary to complete the focused audit on the single-user features of Crypton).

Commendations & Recommendations

Commendations

We commend the developer deployment process. There were a few hurdles with the *ansible* / *LXC* based-deployment, but overall the process was quick and painless.

The code is easy to follow because it has no unnecessary layers of indirection or inheritance.

Recommendations

We recommend to move the property generation from out of `crypton.generateAccount` into the `Account` abstraction to improve encapsulation. (After writing this recommendation in our initial interim draft 1 of this report, we discovered that this lack of encapsulation is related to a security issue — Issue B.)

Split account properties between "public" properties that the server can see and that are stored on the db (maybe call it `AccountDBRecord`?) and an object holding account secrets.

Immutable versus Mutable APIs

We prefer immutable over mutating APIs, when possible.

As an example, the `Account.prototype.unravel` method is mutating: It expects *some* properties to be set as a precondition, and as a side effect it updates *other* properties. This documentation leaves these side effects ambiguous, making it difficult to determine which properties are input, which are output, and which are read/write.

An immutable API would take the input properties as explicit parameters, instead, and return a new object with only the resulting output properties. Or, it may be convenient in this case to return a new fully-initialized `Account` instance.

Remove unused variables and fields

Some variables and fields — for example, `sessionKey` and `keyshmac` — appear to be unnecessary. Removing these would simplify the code, and also the security review by removing blind alleys.

Randomness generation

`crypton.randomBytes` is defined as:

```
function randomBytes (nbytes) {
  return sjcl.random.randomWords(nbytes);
}
```

which is confusing because the argument to `sjcl.random.randomWords` is measured in 32-bit words. (So for example, `hmacKey = crypton.randomBytes(8);` generates a 256-bit key, not a 64-bit one as might be expected.)

Future Work

This section contains all unresolved issues, questions, and concerns at the conclusion of the audit.

SJCL

We have some concerns about SJCL and would like to look into SJCL more deeply. Some of the things that we noticed when looking into SJCL are:

- When is it ever appropriate to pass `paranoia == 0`, meaning that it is okay to read random numbers from the random pool even if that pool has not yet been initialized? Would that be when you don't actually need a secure random number generator, but a non-cryptographic "random" source would do? Perhaps the `paranoia` parameter, the "default paranoia" setting, and the `_PARANOIA_LEVELS` hash should be removed, and instead it should be hardcoded to require that the estimated entropy has reached some basic cryptographic level, such as 256 bits.

  It is difficult to verify by source code auditing that the SJCL random pool is doing the right thing for your specific case, since it is a complex state machine with a lot of state values and configuration settings. Removing the `paranoia` configuration setting would make it easier.

- Catch clauses should cover as little code as possible in order to avoid unintentional suppression of errors.

  In the anonymous function at line 1454 of `sjcl.js`, there is a `try` block with an empty `catch` around the attempt to use `crypto.getRandomValues`. This is presumably intended to ignore cases where the browser does not support `crypto.getRandomValues`, but it may suppress reporting of other errors, e.g. in `sjcl.random.addEntropy`.

- In `sjcl.random.addEntropy`, the idiom `Object.prototype.toString.call(data) === "[object Uint32Array]"` is used as a type test for `Uint32Array` objects (for example). Does this work in Internet Explorer? (Refer to this StackOverflow question for why it might not.)

Things Excluded from Scope

Source Code Authenticity/Integrity/Authorization

As mentioned in Audit Scope, there is a fundamental issue that affects users of all web-based software: how can the user, or security auditors acting on the user's behalf, identify or control what software is executed at runtime, or which of the user's capabilities that software can exercise?

Without this, it is trivial for an attacker who has taken over the software distributor to backdoor users. (In the case of web-based software, the software distributor is at least the origin web server, which may delegate to other web servers as well.)

In fact, this issue is more fundamental than just “web-based software” — it also affects desktop and mobile apps as well, but in the desktop and mobile arenas, mechanisms are already in place that partially mitigate this issue.

We would like to see this important problem solved, and would be willing to consult on such work or to audit proposed solutions. However, we believe that a truly secure and user-friendly solution will require extensive support from the web browser, and cannot be implemented solely by application code like Crypton in current web browsers.

Multi-user interaction

Crypton provides multi-user sharing features, which can introduce subtle security problems.

Typical Web Attack Surfaces

As mentioned in Audit Scope, we did not audit for "standard" web attack surfaces, such as *Cross-Site Scripting (XSS)* or *SQL Injection*.

Side Channels

As mentioned in Audit Scope, there is timing information and potentially other side-channel information from the Javascript implementations of cryptographic operations on secret values (i.e. keys).

Cookies

How are the cookies `crypton.sid` and `uid` consumed? If their usage has any security implications, how do -

- clients protect themselves from values altered maliciously by a server or active MITM?
- servers protect themselves from values altered maliciously by a client or active MITM?

We notice the `uid` is not a high-entropy value (with some high probability).

From interview with Alan Fairless: `uid` is added by `nginx` What about `crypton.sid`?

Use of TLS

The security of the TLS connection between the client and server is largely the responsibility of the browser and out of the control of client JavaScript. However, there may be ways to configure the server, and to a limited extent the client code (for example, setting cookies to be TLS-only), that improve security.

Further cryptographic analysis

The security of GCM mode relies on unique nonces; we did not verify that uniqueness is ensured.

`AES-GCM`, which is length-revealing, is used in several places to encrypt JSON encodings (for example, `sessionKey`, `hmacKey`, and record payloads). Does the length of the input JSON leak important information, and if so should the format of the input be changed?

The use of the "elGamal" public key scheme (which we suspect may not be El Gamal encryption as originally published) needs to be more closely analyzed.

The login authentication protocol has weaknesses described in Issues A and B, and there may be other weaknesses arising from the server's ability to choose `challengeKeySalt`. Investigate alternative protocols that avoid these weaknesses, and perhaps an alternative PBKDF algorithm or parameters to increase the cost of a dictionary attack against the passphrase.

It appears that the cryptography applied to the container name could be simplified, by using a single layer of encryption and eliminating `base_keyring.container_name_hmac_key`.

Acknowledgments

Thanks to Darius Bacon <darius@LeastAuthority.com> and Amber Wilcox-O'Hearn <amber.wilcox.ohearn@gmail.com> for help editing this report.

Appendix A. Work Log

This section gives a coarse summary of the kinds of analysis performed during the audit. The entries are chronological, so they may contain questions, ambiguities, or misunderstandings that were resolved later in the report. We keep track of changes over time to different revisions of this report.

Day 1 - Source Checkout, Developer Installation, Documentation Pass

We cloned / pulled *Crypton*, read documentation, and began learning how to do a local development deploy.

Day 2 - Developer Installation

We installed *Crypton* using the recent *ansible* developer deployment configuration. (Thanks to *ddahl* and *alan*!)

Day 3 - Blackbox/UX Exploration

We prefer to begin an audit by playing with the software without reviewing design documentation or source code. This keeps our initial impressions closer to the user experience and guides our initial focus on the design and implementation. However, unlike true blackbox penetration testing, we make a note and move on when we have a question that is more efficiently answered by code review.

So we began by experimenting with the *diary* application and making observations about it detailed in this section.

Initial Exploration

1. Registered a new user.
2. Created a new entry.
3. Examined the *Chrome* developer console to get a feel for the *HTTP* layer protocol, and noticed:
    - The username is visible to the server, and it used in *URL* paths.
    - There appear to be REST-ful apis at `/account` and `/transaction`.
    - There appears to be a separate polling interface at `/socket.io`.
4. Transactions seen so far appear to have a *create, modify, commit* lifecycle, with one or more modifications. Modifications have request payloads, whereas *create* and *commit* appear to be without request payloads.
5. When a new entry was created, the request payload to `/transaction/12` was (ciphertext truncated for readability):

```
{
  "containerNameHmac": "feb831cc366f12a073b266d7bc96e119d5ef6e6f5c41d2c9734664d1fbbe7ae3",
  "payloadCiphertext": "{\"iv\":\"uqY6MO+yLrbrC7Pc0vu6Qg\",\"v\":1,\"iter\":1000,\"ks\":128,
    \"ts\":64,\"mode\":\"gcm\",\"adata\":\"\",\"cipher\":\"aes\",\"ct\":\"g3JZOF[...]\"}",
  "type": "addContainerRecord"
}
```

6. After editing that entry and saving again, a new transaction update had this payload:

```
{
  "containerNameHmac": "feb831cc366f12a073b266d7bc96e119d5ef6e6f5c41d2c9734664d1fbbe7ae3",
  "payloadCiphertext": "{\"iv\":\"QhJmJAlN9OiJJUp5bP/wrA\",\"v\":1,\"iter\":1000,\"ks\":128,
```

```
    \"ts\":64,\"mode\":\"gcm\",\"adata\":\"\",\"cipher\":\"aes\",\"ct\":\"dBZHSP[...]\"}",
  "type": "addContainerRecord"
}
```

This raises a question about the container name:

- How is `containerNameHmac` calculated?

The container name does not appear to be in cleartext to the server, so we expect it to be present inside the ciphertext `ct`. However, the `containerNameHmac` cannot provide integrity over the whole *entry*, because it did not change when we changed the entry text, and we see the resulting ciphertext `ct` field is different between *entry* revisions. This leads to a second question:

- How is the integrity of an entry revision protected?

Also, we don't notice any overt cleartext about revision ordering. The server providing storage must choose whether to store a given entry revision, and it probably provides some assertion about revision ordering, so there are two more questions:

- Does the server provide revision ordering features? For example, deleting all but "the most recent" entry is a kind of ordering.
- How does the server enforce this ordering?
- Can malicious network eavesdroppers replay entries to confuse clients or the server as to proper order?

While investigating requests to the `/container` URL path, which is a `GET` with the `containerNameHmac` as the last URL path component, we noticed cleartext fields such as `containerRecordId` and `transactionId`. So our questions are refined to:

- Are `containerRecordId` or `transactionId` or some other mechanism used for ordering container records?
- Do these cleartext fields have integrity protection?

Session Management Investigation

After the initial pass over using the *diary* example application, we started to explore account and session management behaviors.

Work log - cookies

Leaving the given tab open (in *Chrome Private Browsing Mode*), we opened a new tab and pasted the *diary* URL, which loads the login screen. Reloading a tag similarly brings us to the login screen. This suggests the session state is managed without cookies. However two cookies are present:

```
crypton.sid: s%3AgyFqVnAOkHXh91QJqND5j52I.%2ButTmaHvvJ7dXkECNSM8ZTp1bwYh44NAnAPbk9PQxHI
uid: AAAAKlKeaz8ILk2QAwMEAg==
```

We notice the `uid` is not a high-entropy value (with some high probability):

- How are the cookies `crypton.sid` and `uid` consumed?
- If their usage has any security implications, how do -
  - clients protect themselves from values altered maliciously by a server or active MITM?
  - servers protect themselves from values altered maliciously by a client or active MITM?

Incorporate this information from a developer interview into this document:

From Alan: `uid` added by `nginx`. We are not sure about `crypton.sid` yet.

`[RESOLVED]`: future work Cookies.

Login Protocol

The login sequence appears to have a challenge/response system, which implies the user and server share a secret, or asymmetric cryptography may be used to provide authentication.

- How is the shared secret or asymmetric secret key derived in the client from username and password?
    - Are there other inputs to this secret derivation?
- How does the server associate the shared secret or asymmetric public key with a given user?
- How are the authentication secrets related to the user's content confidentiality or integrity secrets?
    - Are the user content secrets are derived from the same source as the authentication secrets?
        - If so, can a server or eavesdropper learn the user content secrets given knowledge of the authentication shared secret or public key or any other datum related to authentication?

          If so this would be a vulnerability in confidentiality or integrity protections against the server.

          It may be essential to consider the roles of separate web servers in this investigation of the interaction between user data and authentication secrets. For example, there may be one authentication website, and a distinct user-data-storage website, for a separation of concerns.
- Is the challenge/response protocol an adequate protocol for proving knowledge of a secret without revealing too much information to an eavesdropper (or the server)?

Day 4 - Code Analysis

We used a depth first approach to code analysis, starting with the open questions from the previous UX exploration and examining the example *diary* application.

Authentication

We began by drilling down the stack from `diary.js` through the `crypton` client framework to understand how the username / password entry serves to protect the users data from a malicious server or active MITM.

The entry point for user input to register a new account is the `actions.register` handler function in `diary.js`, which performs some UI functionality around a call to `crypton.generateAccount`. The latter function generates many cryptographically relevant values, then optionally saves them, depending on an option from the caller. Before we analyze their generation and use, we want to understand which values are stored on the server during a save in which manner.

Further analysis will need to consider what security features protect a user from a malicious server, or a server that may become compromised in future.

While investigating the `client/src/account.js` code, we see that the `Account` constructor is empty, and the API depends on callers properly setting properties on the resulting instance. This API style concerns us, because it forces callers to maintain proper invariants and lacks encapsulation.

Day 5 - Code & Design Analysis and Reporting

Reporting

At the end of each week, we write up our progress and findings into a report revision. Each week's report is a newer revision of the same document, so on Day 5 most of our time was spent addressing unresolved questions and To-Do's in the report.

Day 6 - Developer Interview & Design Audit

We had a call with the Crypton developers to present the current findings and solicit feedback. The developers requested we investigate the authentication system and challenge/response protocol, as well as to investigate the *inbox* mechanism for cross-user interactions.

We agreed to focus on the authentication system for now. The *inbox* mechanism is out of our scope, but we may glance at it if we have time.

- `[RESOLVED]` The database stores `base_keyring.container_name_hmac_key` which contains an encrypted key, which is decrypted and used to generate `containerNameHmac`. Since `base_keyring.container_name_hmac_key` is already encrypted with some other key, *K*, why not just encrypt the container name with *K*?

  What benefits and drawbacks are introduced by this extra layer? See Further cryptographic analysis.

> ***Note***
>
> It would be beneficial to have clear terminology to disambiguate cryptographic keys from database indexing keys. (This is a general security engineering issue.) The word *index* can refer either to a lookup datum or the mapping structure.

Day 7

And on the seventh day, we rested.

Also Daira Hopwood joined the audit and set up an environment to run the Crypton code.

Day 8 - Account Generation Audit

We verified that `diary.js` does not touch `sjcl` directly and instead the cryptographic operations are completely encapsulated in the *crypton* framework. We investigated the `crypton.generateAccount` function closely to understand the cryptographic relationships between the parameters generated within it.

We investigated the *pbkdf2* parameters to verify that it uses 1000 iterations with `HMAC-SHA256` and a 64-bit salt. We are not certain that 1000 iterations is sufficient.

- `[RESOLVED]` Decide if we should recommend any change to these parameters. Perhaps *scrypt* is a better candidate, although it does not appear to be available in *sjcl*.

  See Further cryptographic analysis.

We are learning about the `kem()` method of *elGamal* key pairs:

- `[RESOLVED]` Can the owner of the *elGamal* secret key derive the `kem` key given the `tag` portion?
  - Yes, that's as intended.
- `[RESOLVED]` Why is `paranoia == 0` and what impact does that have?
  - It specifies how much entropy is required in order to proceed with the given computation.

    See Issue D. Guessable Private Keys On Browsers Without crypto.getRandomValues().

Day 9 - Furiously Writing Up Results

Finished and delivered "draft 2, 2013-12-13".

Day 10 - Hunting For More Bugs

`sessionKey` is never actually used for anything, just encrypted and decrypted and passed back and forth. `keyshmac` is unused.

The encryption key used to encrypt payload contents is named `hmacKey`. That seems to be inaccurate, possibly vestigial of an earlier design? It is used as the symmetric encryption key for AES-GCM, not for HMAC.

`hmacKey` gets sent to the server in `Account.prototype.serialize`.

Challenge/Response Protocol

Inside `crypton.authorize`:

- `[RESOLVED]` How is `challengeKeySalt` selected by a server?
    - The `challengeKeySalt` is normally chosen by a client during `crypton.generateAccount`, then stored by the server and repeated back to the client on each authentication. So the server may send the same challenge for multiple authentications. This may allow a replay attack by someone who records a challenge/response (if they can defeat the TLS layer), even though the eavesdropper does not know the passphrase.
    - `[RESOLVED]` If the server stores or knows `challengeKeySalt`, it can perform an offline brute force attack. This is common to all (?) passphrase authentication protocols where the server holds a verifier and salt, but the client only knows the passphrase and cannot store anything else. See Further cryptographic analysis.
    - `[RESOLVED]` The server can select a `challengeKeySalt` in an authorize request, and therefore can choose a salt that facilitates attack by a precomputed rainbow table (applicable to all users). See Further cryptographic analysis.
    - `[RESOLVED]` See Issue A. ElGamal keypair recovery vulnerability - If the Account's `keypairSalt` is sent as the `challengeKeySalt`, then the challenge response, computed as `challengeKey = pbkdf2(passphrase, challengeKeySalt);` is the same as `keypairKey = pbkdf2(passphrase, keypairSalt)` so the response will expose the `keypairKey` to the server which it can use to decrypt the *elGamal* keypair in `keypairCiphertext`. This is a result of using the PBKDF for two purposes without diversification.
- `[RESOLVED]` How does the server verify the client's `response.challengeKey`? See Further cryptographic analysis.
- The server values are overwritten on top of `session.account` inside `crypton.authorize`, and then `session.account.unravel()` is called. Because there is not end-to-end integrity check, this allows a server to potentially violate the expectations of `Session.prototype.unravel`.
    - `[RESOLVED]` What attacks are possible from the server through this mechanism?
        - The server can set the account properties so that within `Account.prototype.unravel` all parameters are chosen such that they decrypt to server-chosen values:
            - `this.passphrase` and `this.keypairSalt` can be selected by the server.
            - The other ciphertexts can be selected by the server based on the above so that they decrypt to server-selected values.

      See *Issue B. Server Can Overwrite Client's Secrets*.

Account objects

The account fields sent to a server on a save are found in `Account.prototype.serialize` in `account.js`:

- `challengeKey`
- `containerNameHmacKeyCiphertext`
- `hmacKey`
- `hmacKeyCiphertext`
- `keypairCiphertext`
- `pubKey`
- `challengeKeySalt`
- `keypairSalt`
- `symKeyCiphertext`
- `username`

This is the same set of fields initialized in `crypton.generateAccount` with the exception of `hmacKey` which `generateAccount` does not store directly in a field, which leads to a question about the source of that field:

- `[RESOLVED]` How is the account's `hmacKey` field set prior to a save? Under what conditions?
  - The `hmacKey` is set in `Account.prototype.unravel` so a call to `Account.prototype.save` after an unravel will send this key to the server. This misnamed key is used as an `AES-GCM` key to provide integrity and confidentiality protection of container records (see `Container.prototype.save` and `.decryptRecord`), so once the server has it, it can read and modify container records.
- `[RESOLVED]` What role does sessionKey play? It appears to be set in `decryptRecord`, but where is it used? See Remove unused variables and fields.
- `[RESOLVED]` `AES-GCM`, which is length-revealing, is used in several places to encrypt JSON encodings (for example, `sessionKey`, `hmacKey`, and record payloads). Does the length of the input JSON leak important information? See Further cryptographic analysis.

Work log - Randomness generation

- `[RESOLVED]` `crypton.randomBytes` is defined as:

```
function randomBytes (nbytes) {
  return sjcl.random.randomWords(nbytes);
}
```

  which is confusing because the argument to `sjcl.random.randomWords` is measured in 32-bit words. (So for example, `hmacKey = crypton.randomBytes(8);` generates a 256-bit key, not a 64-bit one as might be expected.) See Randomness generation.

- `[RESOLVED]` In the anonymous function at line 1454 of `sjcl.js`, there is a `try` block with an empty `catch` around the attempt to use `crypto.getRandomValues` . This is presumably intended to ignore cases where the browser does not support `crypto.getRandomValues`, but it may suppress reporting of other errors, e.g. in `sjcl.random.addEntropy`. See SJCL.

- `[RESOLVED]` In `sjcl.random.addEntropy`, the idiom `Object.prototype.toString.call(data) === "[object Uint32Array]"` is used as a type test for `Uint32Array` objects (for example). Does this work in Internet Explorer? (Refer to http://stackoverflow.com/questions/4222394/using-object-prototype-tostring-call-to-return-object-type-with-javascript-n for why it might not.) See Randomness generation.

Miscellaneous

- [RESOLVED] Are TLS certificates verified? See Use of TLS.
- [RESOLVED] Are GCM nonces unique? See Further cryptographic analysis.
- [RESOLVED] Is *elGamal* really El Gamal? See Further cryptographic analysis.

Day 11 and later - Writing up

Subsequent days were spent completing the write-up of this report, including issues D and E.