Report of Initial Security Audit of Crypton

draft 3, 2013-12-13

- 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 which protect the end-user, even in some respects against malicious servers.

This revision of this report is a draft audit results. A final revision will be delivered after the audit concludes.

Audit Scope

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

The primary focus is in single-user application (those without cross-user interactions) security features, especially those intended to protect users against malicious servers.

Scope Limits

A well-known outstanding attack surface which 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 examine excludes that issue.

Given that this is a short audit, we are de-emphasizing cross-user interactions, "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 excluded from the scope of this audit.

This preliminary audit is focused on one particular use case, which is the one represented by the `diary` example app. The `diary` example app allows a user to append timetamped 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 their diary.

Target Code and Revision

This audit targets 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* private key which is stored in `base_keyring.keypair`.

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

Feasibility: An attacker which 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 which 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 which clients use to encrypt their *ElGamal* keypair is also a `pbkdf2` computation with the same password, which 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.

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 which 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 the client's `Account` object to have those properties 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()`.

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 which has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection.

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.

As well as being vulnerable to a malicious or compromised server, the design choice of storing only diffs (as opposed to, say, diffs together with hashes of the expected results) has the consequence that Crypton relies on the *correctness* of the `jsondiffpatch` library (and `google-diff-match-patch` library if used?) for security.

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 which has compromised the server can easily implement this attack.

Verification: We verified this issue by source code inspection.

Implementation Analysis: Crypton core.js passes `paranoia == 0` to `sjcl.ecc.elGamal.generateKeys()`, which means that on a browser without `crypto.getRandomValues()` (e.g. the current release of TorBrowser — TorBrowser 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 movments 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.

Issue E. Encryption Key Disclosed To Server

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

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

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

Verification: We verified this issue by source code inspection.

Implementation Analysis: Crypton *account.js* `Account.prototype.serialize` returns `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 — `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`.

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 authentation, 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 nor the *sjcl* library, except for the parts that the `diary` app directly relies on.

Commendations & Recommendations

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.

Based on our initial code review, one recommendation we have is to move the property generation from out of `crypton.generateAccount` into the `Account` abstraction to improve encapsulation. After writing that 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.

Design and Implementation Recommendations

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

Immutable versus Mutable APIs

We prefer immutable APIs when possible, rather than mutating APIs.

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, so it is 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 would return a new object with only the resulting output properties. Or, it may be convenient in this case it may return a new fully initialized `Account` instance.

Future Work

This section is unfinished as of yet, and will contain all unresolved issues, questions and concerns at the conclusion of the audit. Please see the `[UNRESOLVED]` sections of the Work Log for issues to be moved into this section.

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 contained questions, ambiguities, or misunderstands which were resolved later in the report. We keep track of changes over time to different revisions of this report.

Issue Resolution Convention

We keep track of whether chronological issues are resolved or not with this convention:

- Unresolved questions or investigations are prefixed with `[UNRESOLVED]`. The final report will convert all of these labels into intradocument links to a relevant area of the future work section.
- Resolved questions or investigations are marked prefixed with `[RESOLVED]` and are always intradocument links to an area with the answer, such as a vulnerability description, a nonvulnerability, or a coverage description.

Auditing To-Do

Change the `[UNRESOLVED]` labels to `[RESOLVED]` intradocument links as we resolve each issue.

Auditing To-Do - Final Draft

Any remaining `[UNRESOLVED]` labels need to be covered in the future work section, and the labels need to point to the relevant future work.

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 which 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.

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 entroy value (with some high probability), but we'll leave that for code review:

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

> ***Auditor To-Do***
>
> Incorporate this information from a developer interview into this document:
>
> From Alan: `uid` added by `nginx`. We are not sure about `crypton.sid` yet.

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. This will answer some coarse-grained questions about the implication of server-side saving:

- `[UNRESOLVED]` If a client does not perform a save, are there security features which protect a user from a malicious server? If so, what are they?
- `[UNRESOLVED]` If a client does perform a save, which security features are present which protect them from a malicious server, or a presently non-malicious server which may become compromised in the 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.

- `[UNRESOLVED]` The database stores `base_keyring.container_name_hmac_key` which contains an encrypted key, which is unencrypted 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?

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 - ???

Auditor To-Do

Summarize entries from individual work logs for this day here.

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.

Auditor To-Do

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*.

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

- `[UNRESOLVED]` Can the owner of the *elGamal* secret key derive the `kem` key given the `tag` portion?
- `[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.

Auditor To-Do

Investigate how *sjcl* entropy generation / gathering works, and how `paranoia` interacts with it.

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.
  - `[UNRESOLVED]` 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.
  - `[UNRESOLVED]` 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).
  - `[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.
- `[UNRESOLVED]` How does the server verify the client's `response.challengeKey`?
- 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`.
  - `[UNRESOLVED]` 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.

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.
- `[UNRESOLVED]` What role does sessionKey play? It appears to be set in `decryptRecord`, but where is it used?
- `[UNRESOLVED]` `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?

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.)
- `[UNRESOLVED]` In the anonymous function at line 1454 of `session.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`.

- `[UNRESOLVED]` 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? (See http://stackoverflow.com/questions/4222394/using-object-prototype-tostring-call-to-return-object-type-with-javascript-n for why it might not.)

Miscellaneous

- `[UNRESOLVED]` Are TLS certificates verified?
- `[UNRESOLVED]` Are GCM nonces unique?

- `[UNRESOLVED]` Is *elGamal* really El Gamal?