Cure53 Public Pentest Report: Cryptocat 2

Testing & Documentation: Dr.-Ing. Mario Heiderich, Krzysztof Kotowicz & Maxim Rupp
Proof-Read & Editing: Paula Pustulka · 9th of November 2012

Table of Contents

Introduction

Cryptocat 2 is an open source web application intended to allow secure and encrypted online chatting. Utilizing client's side for the encryption purposes, Cryptocat 2 trusts and uses the already encrypted data on the server only. Cryptocat 2 is delivered as a browser extension, offering plugins for Google Chrome, Mozilla Firefox and Apple Safari. The aim of Cryptocat 2 is to provide means for impromptu encrypted communication that guarantees more privacy than services such as Google Talk and the like. In comparison to various high-level encryption platforms, Cryptocat 2 strives to maintain a high degree of accessibility, while preserving various functionalities, such as that of multiple users connecting to a single chat room.

During the testing, the Cryptocat's 2 source code was analyzed and audited. The code was specifically monitored for concatenation patterns, suspicious function calls, string-to-code sinks, DOMXSS sources and implementation flaws in cryptographic protocols used (Multiparty Protocol Specification, OTR). In addition, for debugging purposes, the incoming server messages were tampered with or simulated, all in the attempt to see whether a malicious server can compromise this client-level security. The following tools were employed during the debugging process: Firefox with Firebug, Google Chrome with Debug Tools.

Note that all vulnerabilities mentioned in this final report and pentest overview are currently repaired and fixes are being consequently verified at the time of writing.

Vulnerabilities

Stored XSS/HTML Injection via Conversation-/Nick-Name *(High)*

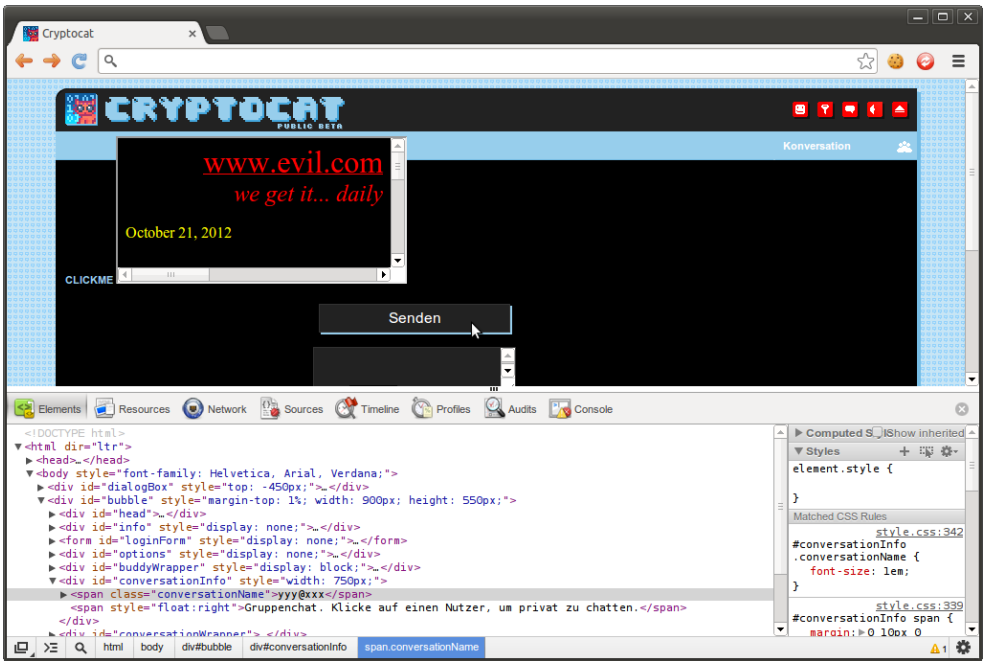
A malicious Cryptocat 2 user can disable the client-side conversation-/nick-name validation feature by removing the following code from the file */cryptocat.js*:

```
else if (!$('#conversationName').val().match(/^\w{1,20}$/)) {
    loginFail(Cryptocat.language['loginMessage']['conversationAlphanumeric']);
    $('#conversationName').select();
}
```

It is through this removal that HTML can be submitted to a server of attacker's choice. This HTML is displayed in the conversation overview without any additional encoding or filtering. Based on the CSP protection in place for the Chrome extension, we may assume that executing JavaScript within the application context is impossible; however, an attacker can inject an HTML form and thereby intercept conversations, collect keystrokes and affect the privacy promise given by Cryptocat 2 in other respects.

Note that the Chrome Extension CSP permits form submissions to external HTTP resources, therefore enabling successful data exfiltration.

Examplary Screenshot:



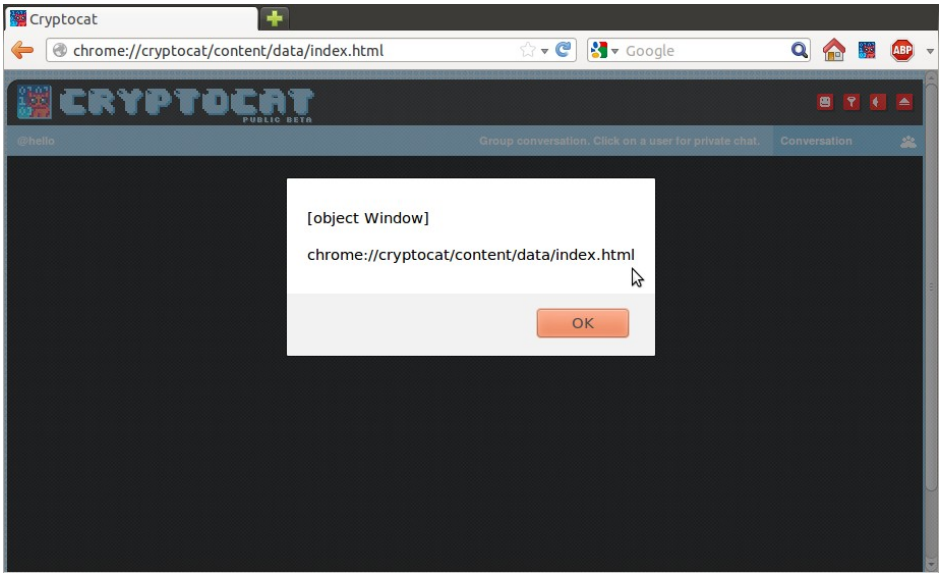
Any input coming from the server or different users should be considered untrusted and should therefore be displayed only when a condition of the proper output filtering is met. In this situation, usage of DOM methods such as *escape()* is recommended.

Remote Code Execution via Conversation-/Nick-Name *(Critical)*

Unfortunately, Firefox Cryptocat 2 extension has proven vulnerable to the very same problem with unfiltered reflection of user-name and/or conversation name. In addition, unlike the Chrome browser, Firefox provides no CSP or similar browser protection.

This elevates this bug in severity, making it a potential RCE vulnerability, as the injected JavaScript can access objects available exclusively in the privileged code. This way, the attacker can create and execute arbitrary files and code segments on the victim's machine, running in the normally sanctioned privileged context of the Firefox browser hosting the Cryptocat 2 extension.

Examplary Screenshot:



As a result, only with proper output filtering in place can an input coming from the server or different users be displayed, as it clearly must be considered untrusted. Analogically to a previous case, this situation also calls for usage of DOM methods such as *escape().*

De-Anonymization / Local Exploits via malformed Data URIs *(High)*

Cryptocat 2 allows encrypted file transfers between its users. The feature relies on a file upload form (inactive in the tested version), which is capable of converting files to data URIs through the DOM functionality. Upon successful upload and conversion, the data URI string is sent to the recipient.

Although the feature itself is not active, the recipient will be able to download the received file because Cryptocat 2 - *despite inactive file transfer feature* - converts all messages that follow a certain pattern into a link pointing to the mentioned data URI. The vulnerability is caused by the regular expression testing for occurrences of data URIs in the received messages.

```
/data:image\/\w+\;base64,(\w|\\|\/|\+|\=)*$/
```

This regular expression allows an attacker to, for instance, transmit arbitrary content applied with the MIME part *image/foo*. A click on the link will trigger a download. In the event that the user opens the downloaded file, the operating system will attempt to sniff its content and, based on internal heuristics, it will decide on how to open or execute that very file.

During the tests, the following data URI was used:

```

```

The file string contains HTML data that will lead the operating system or display manager to store the file locally in a */tmp* folder. It will then be opened with the default browser, thus displaying the embedded content. The result is a possible de-anonymization and local exploitation for cases when the attacker decides to embed JavaScript, Java applets or similar malicious content. The following three screens illustrate the process in stages - from reception of the data URI to the payload execution.

The feature should be fixed through an application of the following best practices:

- A proper whitelist should be used for the permitted MIME types
- `/data:image\/(png|jpeg|gif)\;`
- Cryptocat 2 should not blindly accept arbitrary base64 transported in data URIs
- File data should be validated with use of MIME type & magic bytes of the data URI

Math.random() usage for unpredictable numbers *(Medium)*

There are still occurrences of *Math.random()* usage in the libraries, clearly pinpointing to an insecure PRNG. These should be replaced by *Cryptocat.random()*. In the example below, *Math.random()* is used to generate request ID (rid) for BOSH. As per BOSH documentation [link]:

"The session identifier (SID) and initial request identifier (RID) are security-critical and therefore MUST be both unpredictable and non-repeating (see RFC 1750 for recommendations regarding randomness of SIDs and initial RIDs use for security purposes)."

Math.random() is not considered unpredictable and the state-recovery attacks happening in the past, e.g. in early Chrome, stand to prove this point.

/cryptocat-chrome/js/strophe/strophe.js:

```
1771      this.jid = "";
1772      /* request id for body tags */
1773:     this.rid = Math.floor(Math.random() * 4294967295);
1774      /* The current session ID. */
1775      this.sid = null;
 ....
1839      reset: function ()
1840      {
1841:         this.rid = Math.floor(Math.random() * 4294967295);
1842
1843          this.sid = null;
 ....
2746          this.sid = null;
2747          this.streamId = null;
2748:         this.rid = Math.floor(Math.random() * 4294967295);
2749
2750          // tell the parent we disconnected
```

Next example depicts *Math.random()* usage for generating client nonce for HTTP Digest Authentication scheme. Although there are no specific requirements for unpredictability of client nonces in RFC 2617, the safest bet is to use a secure PRNG for the generation purposes.

/cryptocat-chrome/js/strophe/strophe.js:

```
....
3071
3072          var challenge = Base64.decode(Strophe.getText(elem));
3073:         var cnonce = MD5.hexdigest("" + (Math.random() * 1234567890));
3074          var realm = "";
3075          var host = null;
```

A few other *Math.random()* calls exist for generating unique IDs yet those usage instances are considered safe.

/cryptocat-chrome/js/strophe/strophe.js:

```
1808      this._data = [];
1809      this._requests = [];
1810:     this._uniqueId = Math.round(Math.random() * 10000);
1811
1812      this._sasl_success_handler = null;

 ....
1863
1864          this._requests = [];
1865:         this._uniqueId = Math.round(Math.random()*10000);
1866      },
1867
```

Potential DOM XSS within user nickname alteration *(Medium)*

The function *handlePresence()* is defined in *cryptocat.js* processes for notifications that arrive from XMPP server. This includes the nickname change which might potentially pose a threat. While nickname alterning is not implemented in Cryptocat, it is possible to request a change via another XMPP client or a rogue XMPP server. New nickname is being processed without sanitization.

```
// Detect nickname change (which may be done by non-Cryptocat XMPP clients)
if ($(presence).find('status').attr('code') === '303') {
       var newNickname = $(presence).find('item').attr('nick');
       console.log(nickname + ' changed nick to ' + newNickname);
       changeNickname(nickname, newNickname); // reassigns OTR keys etc.
       return true;
}
```

For this problem to be solved, the code should be change to the following:

```
var newNickname = Strophe.xmlescape($
(presence).find('item').attr('nick').match(/\w+/)[0])
```

This certifies that only alphanumeric characters are used during nickname-changing process.

Invalid HTML code in link markup decorator *(Low)*

The function *addLinks()* is used for converting plaintext links sent in message contents to HTML *<a>* elements. If links with same prefixes are inserted multiple times, the function is being abused in hopes of producing an invalid HTML code. In combination with other

vulnerabilities and/or HTML parsing issues, it might be come useful in an exploitation of a DOM XSS / HTML injection vulnerability.

```
// Convert message URLs to links. Used internally.
function addLinks(message) {
	if ((URLs = message.match(/((mailto\:|(news|(ht … \/){1}\S+)/gi))) {
		for (var i in URLs) { // all links are processed one by one
			var sanitize = URLs[i].split('');
			for (var l in sanitize) {
				if (!sanitize[l].match(/\w|\d|\:|\/ … \.|\&|\;|\%/)) {
					sanitize[l] = encodeURIComponent(sanitize[l]);
				}
			}
			sanitize = sanitize.join('');
			message = message.replace(
				sanitize, '<a target="_blank" href="'
				  + sanitize + '">' + URLs[i] + '</a>'
			);
			 // current link text is replaced globally (i.e. multiple times)
		}
	}
	return message;
}
```

```
> addLinks("http://a http://a")
<- "<a target="_blank" href="<a target="_blank" href="http://a">http://a</a>">http://a</a>
   http://a"
> addLinks("http://a/b/c http://a/b http://a")
<- "<a target="_blank" href="<a target="_blank" href="<a target="_blank"
   href="http://a">http://a</a>/b">http://a/b</a>/c">http://a/b/c</a> http://a/b http://a"
```

The code was later being fixed by using a simple transformation algorithm, allowing the regular expression to determine whether a URL-string was already decorated or had decoration pending.

Multi-party HMAC implementation inconsistent with specs *(Low)*

According to Multiparty Protocol Specification, messages are authenticated with HMAC. The authenticated text is “the concatenation of all cipher-text arranged by sorting the recipient nicknames lexicographically”. However, the Cryptocat nicknames are not sorted at all. Instead, the *sharedSecret* properties are just disorderly iterated:

```
multiParty.sendMessage = function(message) {
	var encrypted = {};
	var concatenatedCiphertext = '';
	for (var user in sharedSecrets) { // no key order guaranteed
	encrypted[user] = {}; // new property will be set in msg
		encrypted[user]['message'] = encryptAES(
			message, sharedSecrets[user]['message'], 0
		);
		concatenatedCiphertext += encrypted[user]['message'];
	}
```

In *multiParty.receiveMessage()* HMACs are validated according to the order in which the users are specified in the (untrusted!) message:

```
var concatenatedCiphertext = '';
for (var user in message) {
       concatenatedCiphertext += message[user]['message'];
}
if (message[myName]['hmac'] === HMAC(concatenatedCiphertext,
sharedSecrets[sender]['hmac'])) {...
```

This makes Cryptocat implementation incorrect as far as specifications are considered. Additionally, any future implementations of the application might not be inter-operable since different cipher-texts would be signed and validated each time. Use of *Object.keys()* and *Array.sort()* for sorting array keys lexicographically is highly recommended.

Typo in multiparty key request implementation *(Low)*

In multiparty.js there are two occurrences of key requests strings:

```
11:    multiParty.publicKeyRegEx = /^\?:3multiParty:3\?:PublicKey:(\w|=)+$/;
149:   answer[user]['message'] = '?:3multiParty:3?:PublicKey:' + myPublicKey;
```

Per specification, public key message uses **publicKey** with lowercase character p:

```
{nickb:{"message":"?:3multiParty:3?:publicKey:publicKeya"}}
```

While this minor error has currently no real impact, in presence of other implementations of the specifications, Cryptocat might send and process public key messages differently, potentially allowing a Denial-Of-Service and resulting in no interoperability among clients.

Remote kick / user impersonalization in multipart chat *(Critical)*

Cryptocat can only allow the creation of users through alphanumeric nicknames. This is assured during the login form verification. Upon receiving messages from XMPP server, the user nickname is truncated on first non-alphanumeric character.

```
var nickname = Strophe.xmlescape(
   $(presence).attr('from').match(/\/\w+/)[0].substring(1)
);
```

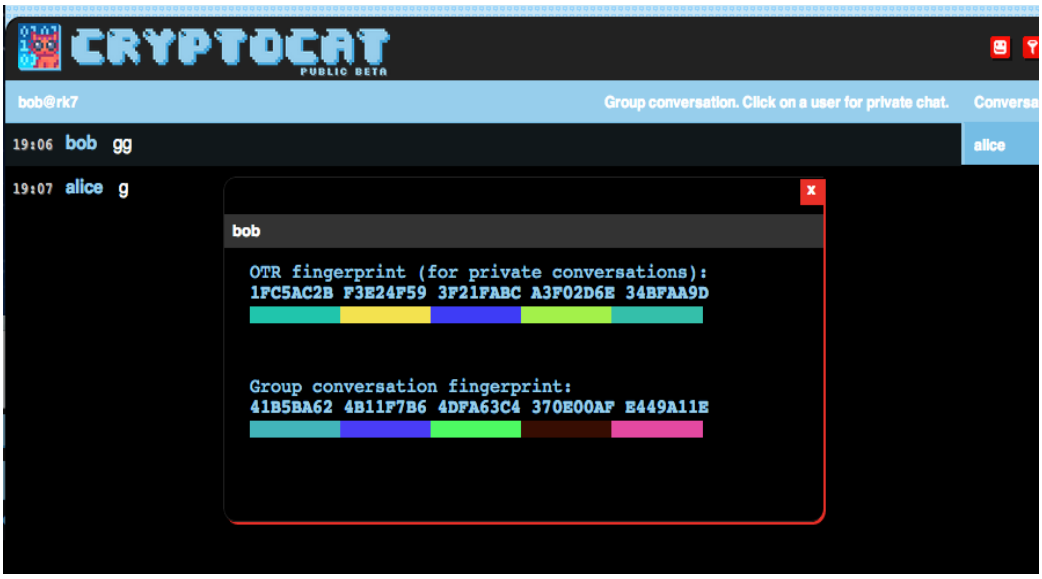
Only the first alphanumeric substring is used for processing as the user's nickname (in *handleMessage* and *handlePresence* functions). Thus, Cryptocat will, for example, interpret *foo@conference.crypto.cat/user-imposter* as *user*. By abusing this functionality with modified Cryptocat client (or other XMPP client), one may be able to remotely kick a user of other users' members lists and instantaneously replace them. New multiparty keys will be generated and all future conversations will be made with the new user. Present users of the chat will only notice a slight animation happening as the user is being replaced.

Having exemplary users 'alice' and 'bob', the procedure is as follows:

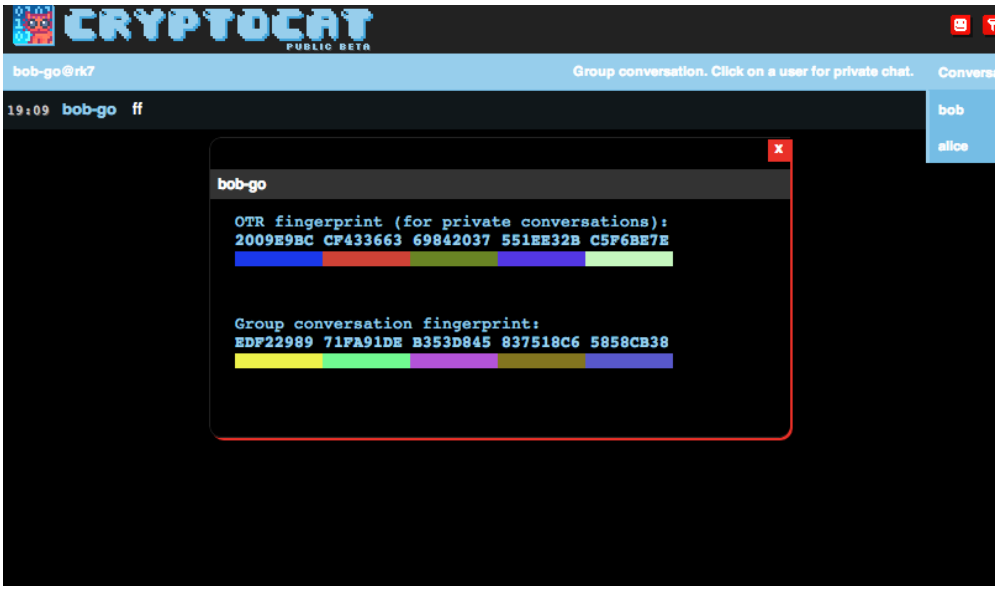
- Step 1: Join a chat as a user '*bob-kickout*' and immediately log out. That will generate *<presence type="unavailable">* message with varied implications. For 'alice', user bob will be logged out, his keys destroyed and so and forth, whereas for 'bob' user, the chat is now a black hole as he will not get new messages from other members who have had his key removed.
- Step 2: Immediately log in as the user *bob-imposter*. 'Alice' will create a user entry '*bob*' with imposter keys. This will subsequently generate a race-condition situation with old *bob** user's *<presence>* messages coming in, yet it is perfectly feasible to win the race.

The screenshots below serve as illustrations of the procedure's effects.

User Alice only sees bob user with group fingerprint EDF22989



Original Bob can still see alice but has a different (old) key



With modified client, bob-go can communicate with alice, pretending to be (original) bob

To conclude, Cryptocat 2 should deny processing of *<presence>* and *<message>* messages from users with non-alphanumeric characters, ceasing the truncating of the nickname on the first such character.

Usernames capable of altering the logic of Cryptocat 2 *(Low)*

In multiple parts of an application, nicknames are used as keystore object keys. For example:

```
// handlePresence
if (nickname !== 'main-Conversation' && otrKeys[nickname] === undefined) {

//multiParty.receiveMessage
if (!publicKeys[sender]) {
	var publicKey = message[myName]['message'].substring(27);
	if (checkSize(publicKey)) {
		publicKeys[sender] = publicKey;
```

Due to Javascript prototype-chain behavior, keystore objects (e.g. *publicKeys*, *otrKeys*, *sharedSecrets*) equally have alphanumeric properties of *Object.prototype* e.g. *valueOf*, *toString*, *constructor* etc. By using these names as nicknames, it is possible to trigger unspecified logic flaws in Cryptocat. For example, chat ceases to work when a '*constructor*' user joins in. It is advisable to use Object.hasOwnProperty instead of array access operator when verifying property's existence.

XMPP request IDs potential disclosure of OTR chat activity *(High)*

Multi-party (group chat) and OTR messages share the same XMPP-over-BOSH connection implemented via *Strophe.js* library. Each message from a user is sent to the server with a unique ID value. Server then redistributes the message to the appropriate chat participants accordingly. Due to the implementation details in *Strophe.js,* the ID is not only unique but it is an incrementing number, starting from a random value.

```
// strophe.muc.js, in various places
message: function(room, nick, message, html_message, type) {
	var msg, msgid, parent, room_nick;
	room_nick = this.test_append_nick(room, nick);
	type = type || (nick != null ? "chat" : "groupchat");
	msgid = this._connection.getUniqueId(); //

// strophe.js
getUniqueId: function (suffix)
{
	if (typeof(suffix) == "string" || typeof(suffix) == "number") {
		return ++this._uniqueId + ":" + suffix;
	} else {
		return ++this._uniqueId + "";
	}
},...
```

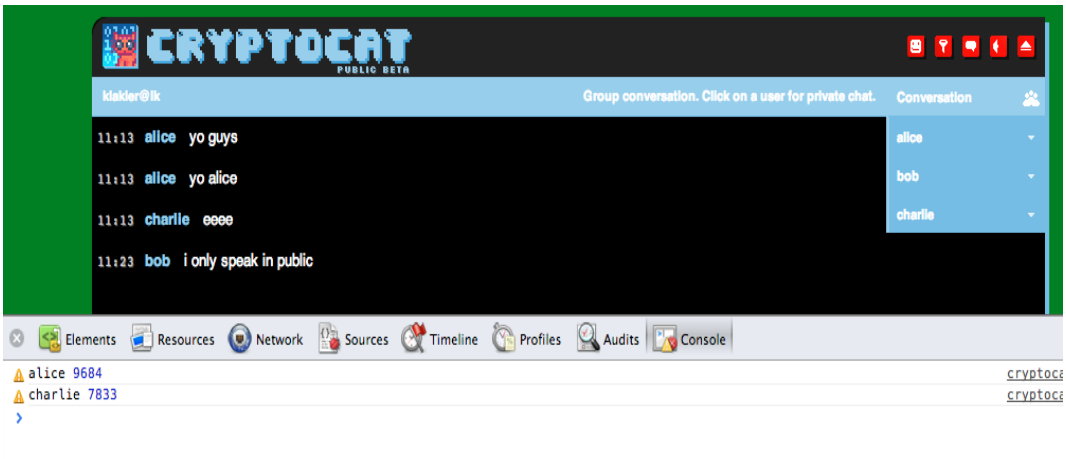
For that reason, tracking user message IDs is possible through this property. Details as to whether a user is currently engaged in an OTR conversation and how many messages has he sent in such conversation thus far can be detected and extracted through tracking processes. With next 'groupchat' message of such user, missing ID numbers will be observed.

```
var oldId = (typeof window.lastIds[nickname] !== "undefined") ?
window.lastIds[nickname] : 0;
window.lastIds[nickname] = parseInt($(message).attr('id'), 10);
if ((oldId + 1) !== window.lastIds[nickname]) {
	console.warn(nickname, window.lastIds[nickname]);
	// missing IDs detected
}
```

Even an unskilled attacker can periodically force every user to send invisible public messages by asking for OTR fingerprints in the likes of:

```
for (var nickname in otrKeys)
	otrKeys[nickname].sendQueryMsg();
```

Combining these two methods may appear quite trivial, but it serves it purpose of building a modified Cryptocat client that constantly monitors users of a group chat, detecting their OTR activities as they happen.



Modified Cryptocat client detects that Alice and Charlie are engaged in OTR conversation. To simply eradicate that behavior, message IDs should not be incrementing numbers. To guarantee uniqueness, a keyed hash of an incrementing number could be sent instead.

Cryptocat Chrome extension's cross-origin detection *(Low)*

Cryptocat Chrome extension uses manifest version 2, which by default offers protection from Chrome extensions fingerprinting. Nevertheless, *img/keygen.gif* image file, used in Cryptocat notifications, has a specific permission in the *manifest.json*, so that it can be a *web_accessible_resource*. This gives any website the possibility to determine if current visitor has the extension installed. The code for such detection routine is demonstrated below:

```
<img src="chrome-extension://[extension-id-from-chrome-web-
store]/img/keygen.gif" onload=alert(/hascat/) onerror=alert(/hasnot/) >
```

If the premise of Cryptocat is for it not to be detected by other websites, *web_accessible_resources* array should be left empty.

The Cryptocat team has evaluated and accepted this risk; the problem will not be addressed in the foreseeable future.

OTR implementation vulnerable to poisoning in rare cases *(Medium)*

OTR uses Socialist Millionnaire protocol for a key exchange. This protocol is "vulnerable to poisoning whereby either Alice or Bob chooses (a_2, a_3) or (b_2, b_3) to be zero to be able to predict the result" (Wikipedia). a2 and a3 are chosen randomly in *otr.js*:

```
HLP.randomExponent = function () {
    return BigInt.randBigInt(1536)
}

this.a2 = HLP.randomExponent()
this.a3 = HLP.randomExponent()
```

To make sure that neither party chose (or deliberately set) a_2, a_3 to zero, a verification check should be made on the values of (g2a, g3a) (own) and (g3a, g3b) (incoming). All of them should not be equal to 1. This check is currently missing in the code. However, during rudimentary testing it seems that Socialist Millionaire protocol never gets triggered when using OTR chat within Cryptocat (further testing needed).

Other findings

The following paragraphs will list potential problems that might at some point lead to bugs and/or vulnerabilities. Future versions of the software might be tackling these issues.

Entropy Pool Misuse

Application uses a secure PRNG provided by *window.crypto.getRandomBytes()*. Unfortunately, due to the way it is used, a lot of entropy is wasted when converting raw byte values to a float in *Cryptocat.random()* occurs. Each random byte produces one decimal digit for a float value, thus reducing 8 bits entropy to 3.32 bits.

```
// cryptocatRandom.js , Cryptocat.random()
var buffer = new Uint8Array(1);
while (output.length < 16) {
       window.crypto.getRandomValues(buffer);
       if (buffer[0] < 250) {
              output += buffer[0] % 10;
       }
}
```

Cryptocat.random() is mostly used for choosing an array key.

```
// bigint.js
r=pows[Math.floor(Cryptocat.random()*512)];
```

The same could be obtained with *Math.floor(Cryptocat.randomByte()/255 * array_size)* without the entropy pool being wasted.

Another usage method is to simply get a random 32-bit word, so that raw bytes would be even more useful, improving the overall performance.

```
// crypto-js/core.js:
274  var words = [];
275  for (var i = 0; i < nBytes; i += 4) {
276     words.push((Cryptocat.random() * 0x100000000) | 0);
277  }
```

The suggestion would be to add *Cryptocat.randomByte()* / *randomWord()* functions and use them throughout the application, especially for key generation. The clear benefit will be in the improved process speed.

Multiparty public key distribution inconsistent with the specification

Public key distribution is different than described in the specification which reads:

“Once a user receives another user's public key, they must send them their own public key in return. Therefore, a user `a` may demand another user `b`'s public key by sending them their own public key.”

On the contrary, in the code *multiparty.receiveMessage()* there is no sending back of the key:

```
if (message[myName]['message'].match(multiParty.publicKeyRegEx)) {
	if (!publicKeys[sender]) {
		var publicKey = message[myName]['message'].substring(27);
		if (checkSize(publicKey)) {
			publicKeys[sender] = publicKey;
			multiParty.genFingerprint(sender);
			multiParty.genSharedSecret(sender);
		}
	}
	return false;
}
```

Additionally, there is no key sent in exchange to a specific key requesting message:

```
else if (message[myName]['message'].match(multiParty.requestRegEx)) {
	multiParty.sendPublicKey(sender); // this doesn’t send, just generates!
}
```

Instead, one’s own public key is distributed to a new user whenever they are added to a buddy list (*newBuddy* function), which is an effect of a different *<presence>* message.

It is worth noting that following the specification will make it possible to separate joining a multiparty chat (currently shown in the UI) from the process of distributing keys. Potentially, this allows for e.g. joining a chat as an invisible user (attacker sends *<presence type=unavaliable>* followed by multiparty public key requests). Recommended action is to mark when the keys are added / dropped from the keystore in the UIs, especially when the other party remains outside of the displayed users-list.

General Comments and Security Advice

The following paragraphs will cover general security recommendations and discuss several findings that were not exploitable but may become so if the luck is pushed one day. In other words, the here-described issues are defense-in-depth recommendations , deemed to be of low priority but worth looking into when spare time is available for development. We will list our suggestions one by one in the advice-style format.

Avoid untrusted input in jQuery selectors

During the JavaScript security code audit, several “almost injectable” entry points were located. One of them needs to be mentioned specifically, mainly for two reasons. For one, the barrier between security and vulnerability is rather slight and, secondly, it could be broken in case the library maintainers - namely the jQuery authors - decide to handle a specific critical feature differently.

The Cryptocat 2 application often employs user-generated content in code sequences such as: `$('#buddy-' + nickname)`. Here, the sole existence of the sharp character (#) introducing an ID-based DOM query, results in preventing a jQuery-based DOMXSS from being a possible tool to be used against Crypotcat 2 (given an unsanitized value for nickname, s.a.). Once a complex query is utilized, for instance `$('.buddy-' + nickname)`, an attacker is capable of using the nickname `\x3cimg\x20src=\x20onerror=alert(1)\x3e` to trigger a DOM element creation via the jQuery selector. This instantly leads to a possibility of executing arbitrary JavaScript and, eventually, a potential code execution issue in Firefox. While none of the JavaScript files we examined during the code audit were vulnerable, it needs to be underlined that this concatenation pattern might become dangerous in the future, in a not-so-unlikely scenario where either jQuery changes its behavior or Cryptocat 2 starts using new features where user input hits a complex selector. Detailed documentation for this issue can be found at: http://ma.la/jquery_xss/

Avoid all non-alphanumeric characters in nicknames

Aside from what has been already said, it is worth mentioning the problems that may arise when a user-name is passed directly to jQuery selector and even ‘innocent’ characters like “,” or whitespace can be used to trigger logic flaws.

Let’s have a look at the code below:

```
if (!$('#buddy-' + nickname).length) {
       return true;
}
```

Nickname *whatever,div* can be used here to return early from the function *newBuddy()*. This is currently unexploitable but may become important if validation rules for a nickname change in the future. It is advisable to use dedicated *document. getElementById()* functions instead of creating jQuery selectors with untrusted input in-place.

Implement centralized filtering method

Another recommendation is to use a centralized filtering method for all values that can be tampered with by adversaries and actors other than the potentially affected user. This holds for the user-name, the conversation name as well as values being sent by a potentially rogue communication server instance. One must take it into consideration that even data such as the OTR key or other instances might be tampered with by motivated attackers. Therefore, a centralized filter tool which would continuously ensure proper output filtering for any incoming byte-string is urged, as it would assist in keeping Cryptocat 2 and any upcoming versions of the tool as safe and secure as possible. Depending on the usage scenario, even the browser locale might be considered a valid attack vector which leads to an exploit against the concatenation in the *language.js* file. Our tests have demonstrated Denial-of-Service (DoS) potential against Cryptocat 2 with tampered *locale* values - the impact would thus be comparably low. At the same time, we recognize Cryptocat 2 as an extremely security-critical application, which stands to reason why even the most abstract attack vectors are worth mentioning.

Avoid allowing SVG files

Ultimately, it needs to be underscored that the task of properly securing the upcoming file transfer feature is going to be critical and not an easy one at that. The damage potential of unregistered MIME types as well as SVG images needs to be kept in mind. While we already demonstrated the problems caused by unregistered MIME types in this very report, future versions of Cryptocat 2 might allow SVG images to be transferred from user to user. Depending on the specific browser in use, it could cause XSS and even worse attacks to become possible. The security implications connected to SVG images have been documented in depth in Heiderich et al. http://dl.acm.org/citation.cfm?id=2046735

Consider removing wildcard host permissions in Chrome extension

Chrome extension port of Cryptocat uses wild card host permissions in *manifest.json*:

```
"permissions": [
      "http://*/*",
      "https://*/*"
]
```

In the event that HTML injection vulnerability is exploited, the extension may provide means for launching a universal XSS against any domain. In effect, c&c / data extraction backchannel pointing to any domain can be set up as well . As the most common use case is a simple calling of the *conference.crypto.cat* and *crypto.cat* domains, this could be prevented by exclusively specifying those domains in the manifest. If a user tries to connect to a custom server, this could be allowed via optional permissions mechanism. Surely it will impact the usability to a small degree, as the user confirmation is required, but will only affect users connecting to their own servers. Unfortunately, this in-depth defense mechanism cannot be used in Firefox.

Consider moving key storage and encryption logic to WebWorker thread

Cryptocat currently uses WebWorkers threads only to avoid blocking UI during DSA key generation and processing of a file upload. They can also be used to separate all kinds of private key generation, signing, encryption and validation of messages. This defense-in-depth approach will become handy in case of a DOM XSS vulnerability being discovered (in the environments where CSP is not available). When exploiting the main thread of Cryptocat, the attacker would not be able to extract private keys used by the application. This, however, requires a substantial reworking of the application code.

Develop protection against timing attacks

JavaScript libraries used by Cryptocat do not use constant-time calculations when operating on cryptographic primitives. This creates the possibility of timing attacks. Application, being just a browser extension, does not have a total control over execution environment. To give an example, the application will be run under different JavaScript engines, each with its own optimization techniques, further complicating the fixing of this issue. It might be appropriate to introduce a time-padding when data is being sent over the network in order to reduce the availability of a timing side-channel option for a remote attacker.

Conclusion

Cryptocat 2 has reached a great maturity level in a very short period of time. It is commendable that the development team has proven great expertise in the creation of secure code, despite the complexity of the task at hand. While communication process is critical in the dynamically updated framework of audits (both during the assignment and following its completion), it was exceptionally well-handled in this case, resulting in the discussed issues acquiring almost immediate fixes. Let us illustrate that by saying that on several occasions feedback with successful fix notification has managed to reach us concurrently to follow-up email's preparation!

Nevertheless, the problems we have spotted underline the importance of a well-planned and thoroughly implemented security architecture within browser extensions. One has to be reminded that a vulnerability that causes a rather harmless script execution in the web application context, might turn out to become a detrimental privilege escalation or remote code execution when it is discovered and exploited in a browser extension. Cure53 would like to thank Radio Free Asia, the entire Cryptocat development team and Nadim Kobeissi partciularly, for this challenging and all-round professionally-handled project.