- ORIGINAL ARTICLE -

Other potential problems in Qlink.it

Otros problemas potenciales en Qlink.it

Antonio Castro Lechtaler[1], Marcelo Cipriano[1], Edith García[1], Pablo Lázaro[2], Julio Liporace[1], Eduardo Malvacio[1] and Ariel Maiorano[1,2]

[1] *Grupo de Investigación en Criptografía y Seguridad Informática (GICSI), Universidad de la Defensa Nacional (UNDEF), Ciudad Autónoma de Buenos Aires, Argentina*
{acastro,marcelocipriano}@est.iue.edu.ar, {editxgarcia,edumalvacio,jcliporace}@gmail.com
[2] *Dirección de Gestión Tecnológica (DGT), Policía de Seguridad Aeroportuaria (PSA), Argentina*
{plazaro,amaiorano}@psa.gob.ar

Abstract

In previous work we presented preliminary results obtained by reviewing the source code of Qlink.it web application. In this article, after summarizing previous findings, results of the source code review of Qlink.it Android application will be described. This analysis focused on the implementation of cryptographic functionalities. The aim of this publication is also to invite other researchers to analyze the application in order to determine if Qlink.it could be considered secure.

Keywords: Information security, application security, source code review, cryptography, random number generation.

Resumen

En un trabajo previo presentamos los resultados preliminares obtenidos al revisar el código fuente de la aplicación web de Qlink.it. En este artículo, después de resumir los hallazgos anteriores, se describirán los resultados de la revisión del código fuente de la aplicación Android de Qlink.it. Este análisis se centró en la implementación de funcionalidades criptográficas. El objetivo de esta publicación es invitar a otros investigadores a analizar la aplicación para determinar si Qlink.it podría considerarse seguro.

Palabras claves: Seguridad informática, seguridad de aplicaciones, revisión de código fuente, criptografía, generación de números aleatorios.

1. Introduction

In previous work [1] we presented the preliminary results obtained by reviewing the source code of the Qlink.it web application. These results will be summarized below. In this article, the source code review findings of potential security problems in the Qlink.it Android application will be described. This source code is also published on the project repository on github.com [2], and the application can be installed on Android devices from Google Play [3].

This analysis also focused on the implementation of cryptographic functionalities. We'll describe implemented mechanisms that discard key material significantly reducing the security against brute-force attacks, code that reuses the same key and initialization vector applying AES 256 in CBC mode, and insecure ways of seeding secure random generators. At least one problem that could compromise the secrecy of the encrypted message will be described. This problem could be exploited if the qlink was generated in versions of the Android operating system prior to 4.2 and under certain very specific circumstances of message length and timing conditions.

As mentioned in [1], given the news [4,5] about the availability of the Qlink.it source code [6], and considering, that among GICSI objectives, the group studies techniques and mechanisms for the revision of source code, focusing on aspects related to information security in general and to cryptography in particular [7,8]; and, that the DGT has the responsibility, among others, to periodically evaluate alternatives for the secure communication of the Institution's personnel; a first general review of the source code of the Qlink.it web application [9] was

Citation: A. Castro Lechtaler, M. Cipriano, E. García, P. Lázaro, J. Liporace, E. Malvacio and A. Maiorano. *"Other potential problems in Qlink.it"*, Journal of Computer Science & Technology, vol. 18, no. 2, pp. 160-166, 2018.
DOI: 10.24215/16666038.18.e18
Received: February 20, 2018 **Accepted:** July 30, 2018.

carried out jointly.

Our findings on the web application were published in our previous article. Here we describe the results obtained after the review of the source code of the Android application. By the time the first part of the analysis was completed (May 2017), the preliminary results of the review would indicate the existence of potential security problems, for which reason it was decided to consult Qlink.it developers sharing these results.

Although it was a review that did not cover the entire system, and it was not finished, permission was requested to publish, in the form of an article, with the intention of inviting other reviewers to study the application, who could confirm or reject these potential risks, and determine if the system could be considered safe.

A limited summary without all the details of the first results was also published on a website dedicated to information security, Segu-Info [10].

1.1. About Qlink.it

As indicated in the project documentation [6,9], specifically in its FAQ section, *"Qlink.it is a new, very simple and secure way to send confidential information through the internet"*.

In summary, according to the Qlink.it website, in its advanced FAQ [11], the operation of the system is described as follows:

1. *When you enter a message in qlink.it and click the "qlink it!" button, your browser runs a Javascript program which encrypts the message with a given random key, say for instance YYYYYY.*
2. *Afterwards, the encrypted message is sent through secure https protocol to the Qlink.it server.*
3. *At the server, the message (already encrypted with key YYYYY) is encrypted again to be stored, but now with another random key, say for instance XXXXXX.*
4. *Then, the server returns to you a preliminary qlink, in this case https://qlink.it/XXXXXX.*
5. *At that moment, your browser adds at the end of the preliminary qlink the key that only your browser knows to form the full qlink: https://qlink.it/XXXXXX#YYYYYY. Notice that the Qlink.it server didn't have access to the YYYYYY part of the qlink!*
6. *Then, you copy & paste the full qlink and send it to the intended recipient, either by email, chat, WhatsApp, or whatever.*
7. *When the recipient receives the full qlink and clicks on it, the browser only requests to the server the preliminary qlink, https://qlink.it/XXXXXX, because the special character hash mark (#) indicates that what follows should not be sent through the internet! (You can check this feature by using for instance the inspect option in some browsers as could be Chrome.) Therefore, the Qlink.it server never has access to the full key to read the true content of the message!*
8. *When the server receives the request with the preliminary qlink, the qlink has in it the key to look for the encrypted message and partially decipher it. The server then sends back through https secure protocol a message which is still encrypted with the unknown-to-the-server key YYYYYY. At that moment the server makes a secure delete on the encrypted message and is not available any more at the server.*
9. *When the recipient's browser gets the encrypted message, since it kept the last part of the full qlink YYYYYY, it runs a Java script to finally decipher the encrypted message using this last part of the full qlink. Once the message is totally deciphered, the browser displays it on the recipient's screen.*

2. Potential problems in the Qlink.it web application

2.1. Cross-Site-Scripting (XSS) vulnerability

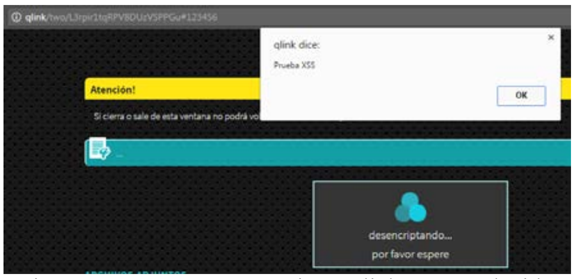
Although Javascript functions are used to filter input fields when generating a qlink, the one that contains the message does not seem to be verified or correctly filtered.

The code below shows that simulating a browser requesting the web-service to generate a qlink, arbitrary Javascript code can be included. After a closing the *textarea* tag element that presents the decrypted message, a *<script>* element with an *alert()* invocation demonstrates the XSS vulnerability.

Algorithm 1 Snippet of Python script written to demonstrate the XSS vulnerability.

```
sess = requests.Session()
r = sess.get(url + '/tokenizer',
    headers=headers, data={})
x_token = r.json()['x_token']
message = "%%A%%</textarea><script>
    alert(' Pru eba    XSS')</script>%%C%%"
password = b'123456'
salt = "ffffffffffffffff" # example
iv = "ffffffffffffffffffffffffffffffff" # example
key = hashlib.pbkdf2_hmac('sha1', password,
    binascii.unhexlify(salt), 100, dklen=32)
```

```
cipher = AES.new(key, AES.MODE_CBC,
    binascii.unhexlify(iv))
coded = base64.b64encode(cipher.encrypt
    (message .ljust(int(math.ceil(len(message) /
    16.0) * 16), b'\0')))
data={'msg':'{"data":"'+coded+'","salt":"'+sa
    lt+'","iv":
    "'+iv+'","iter":100,'x_token':x_token, ... }
r = sess.post(url + '/inject', headers={ ... },
    data=data)
print 'qlink: ' + r.json()['hash'] + '#' +
    password)
```


qlink/two/L3rpir1tqRPV8DUzVSPPGu#123456

qlink dice:

Prueba XSS

Atención!

Si cierra o sale de esta ventana no podrá vo

desencriptando...

por favor espere

Fig. 1 Screen capture accessing a qlink generated with the Algorithm 1 script Preparation of manuscript

2.2. Cryptography implementation

After the review of source code files *public/js/application.js, app/src/Qlink/Models/Utils/ RandomHasher.php* and */app/src/Qlink/Controllers/ LandingNewController.php*, it was noticed that the first part of the qlink, that is, the first 10 characters, for example: *http://qlink/two/XXXXXXXXXX...* Are generated (not exclusively) based on a timestamp (with millisecond precision) -result of the Javascript function *Date().getTime()*- that the browser sends to the server, and therefore, that it could be manipulated. Although the server will register that value for use in the next qlink, using the previous value in the current request, previously registered in the same way, the value is then added to the server's timestamp, to the number of microseconds multiplied by 10^5, and used as a seed -by the *mt_srand()* function- to then obtain values from the *mt_rand()* function.

These functions, based on the Mersenne Twister generator, are not suitable for generating random numbers for cryptographic operations, warning also explicitly noted in PHP official documentation [12]. For example, the following code snippet shows how the seed used to generate the first part of a qlink could be obtained:

Algorithm 2 Python function source code that demonstrates how to obtain the seed used to generate the first part of a qlink.

```
def obtener_semilla(parcial_qlink):
    chars = '0123456789abcdefghijklmnopqr...
    epoch = int(time.time())
    len_prueba = 60*60*24*1000*1 # 1 dia
    ii = 99999 + 999 + epoch * 1001
    max = 0
    for i in xrange(len_prueba):
        php_rand.mt_srand((0xFFFFF... & (ii-i)))
        for j in range(len(parcial_qlink)):
            g = php_rand.mt_rand(0, len(chars)-1)
            if chars[g] != parcial_qlink[j]:
                break
            if (j + 1 > max):
                max = j + 1
                if max == 10:
                    return ii-i
```

Regarding the code that will be executed in the browser through Javascript, although indirectly -through the CryptoJS library and its function *CryptoJS.lib.WordArray.random()*-, random number generation ends up invoking the Javascript function *Math.random(),* which is implemented by most browsers based on variants of the Xorshift128 + generator, which is also not considered safe or recommended for the implementation of cryptography [13,14,15,16].

2.2.1. Estimating date and time of creation of a previous qlink

It was shown that generating a new qlink and using its first ten characters to obtain the seed, it is possible to estimate of when the previous qlink was created. The script included in our previous work used the module or package py*php_rand* [17] as used in the example code of Algorithm 1.

Suppose x as the Unix epoch timestamp from the moment the previous qlink was generated, in seconds; and x_m to the parameter that was sent to the server at that moment, in milliseconds, so for the purposes of this approximate estimate, $1000x < x_m < 1000x + 999$. Also assume t equal to the Unix epoch timestamp of the moment when we generate the new qlink, in seconds. Finally consider u as the amount in microseconds used in PHP, which would be generated such that $0 < u < 99999$.
Therefore, the seed for the qlink that we are generating, s, would correspond to $x_m + t + u$. Then, $s = x_m + t + u$, $s = 1000x + y + t + z$, *with* $0 < y < 999$ *and* $0 < z < 99999$. *Then,* $x = (s - y - t - z) / 1000$. This being an approximation, the term $y / 1000$ could by eliminated, and $z / 1000$ is replaced by a delta d, *with* $0 < d < 99$, *then* $x = [(s - t) / 1000 - 99, (s - t) / 1000]$. So the approximation result corresponds to a range of 99 seconds.

2.2.2. Obtaining the "DN number" from a qlink

Also based on previous examples, it is possible to obtain the tracking code, or "DN number" from the first ten characters of the variable part of a qlink. The proof of concept script included in our previous article could be used against any qlink.

The "DN number" is generated by a function very similar to the one used to generate the first ten characters code of a qlink, laying the difference in the set of possible characters for the mapping of random numbers. In this case, the result corresponds to ten digits. Also, the same timestamp is used for the generation of the first part of a qlink. The function is invoked after just over about 50 lines of code of the generation of the first part of the qlink.

Therefore, another script of our previous article brute-force the time elapsed between the invocation of these two functions and then check the existence of the tracking code in a limited space, with the intention of reducing the amount of tests to be performed.

2.2.3. Insecure random number generation using Javascript library CryptoJS

Regarding issues related to the code executed in the browser, specifically in relation to the generation of random numbers, we have demonstrated potential problems with the use of a library based on the function that browsers provide, implementing the Xorshift128 + generator.

In this case, our test script was an adaptation of another available in [18], which works directly with outputs of the *Math.Random() function*, using the Z3 tool, *"a high-performance theorem prover being developed at Microsoft Research"* [19], for the symbolic resolution of the system of equations given the known partial information. The test example was adapted for resolution with values truncated by *CryptoJS.lib.WordArray.random()*. The way to generate the salt and the initialization vector in qlink was taken as an example to estimate or guess the following possible values of the generator. While the example does represent a risk, it should be considered that the same function is used to generate key material.

3. Potential problems in the Qlink.it Android application

3.1. Potentially insecure password generation

From a *SecureRandom* instance seeded with timestamps and results of previous invocations, the source code snippet in Algorithm 3 shows the generation of the string that then would be encoded in base 64, truncated and used as input for generating the AES key, using *SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1")*, requesting 256 bits of key material, and specifying only 100 iterations.

Algorithm 3 Snippet of source code from the generateRandomString() method.

```
SecureRandom r = new
    SecureRandom(seeded.getBytes());
String characters ="0123456789abcdefghijkt...
String randomString = "";
for (int i = 0; i < length; i++) {
    int jind = r.nextInt(characters.length() - 1);
    randomString = randomString +
    characters.charAt(jind);
```

Considering that truncation con leave only 16 base 64 encoded characters, the password consist of 12 characters in the range [0-9a-zA-Y] (the Z is not included in the set because of an error in the *nextInt()* parameter). This leaves 61 possible characters, so if an attacker would try to brute-force this password, he will need to make, on average, $61^{12}/2 \approx 2^{70}$ tries.

3.2. Key and IV reutilization with AES 256 in CBC mode

It is well known that for symmetric block ciphers operating in Cipher Block Chaining (CBC) mode, the key and the initialization vector should not be used more than once.

The Qlink.it Android application allows to attach multiple files to a message, unlike the web application, here the same key and IV are used for the message and all the attached files.

The following source code snippet in Algorithm 4 shows that the contents of the *fpassword* variable, used also for encrypting the message, would be used for the attached file. This code, as the one from Algorithm 3, are taken from the *src/com/qlink/ar/QlinkActivity.java* Qlink.it source file.

Algorithm 4 Key and IV re-use in Qlink.it Android application.

```
String String[] arrayEnc =
    encFiles.toArray(new
    String[encFiles.size()]);
for (int i = 0; i < arrayEnc.length; i++) {
  JSONObject jof = new JSONObject();
  try {
    jof.put("data",
    util.encrypt(salt, iv, fpassword,
    arrayEnc[i]));
```

3.3. Deciphering qlink messages in Android versions prior to 4.2

Qlink.it Android application can be installed from

Google Play [3] on devices with Android Operating System as old as version 2.3. Google's metrics on version implementation are available at [20].

According to the cryptography entry on Android 4.2 security notes [21], the default implementation of *SecureRandom* was modified. Also from an official Google source [22], it was publicly known that there was a problem with using *SecureRandom()* in the way Qlink.it uses it. Starting with Android 4.2, the default provider is OpenSSL and a developer can no longer override *SecureRandom*'s internal state, but the old implementation allowed overriding the internally generated key for each instance. Developers which attempted to explicitly seed the random number generator, as done in the Qlink.it application, would find that their seed replaces, not supplements, the existing seed. Using the same seed, prior to Android 4.2, invocations of *nextInt()* would always return the same number.

Other problems beside [23,24], the way Qlink.it generates the password may be insecure in these devices because it consist in repeatedly seeding the generator with timestamps (with milliseconds precision) and previous results. Estimating a timestamp range, first user interaction could be brute-forced. For example, in Algorithm 5:

Algorithm 5 Seeding the random number generator (invoking Algorithm 3 function).

```
String public void onUserInteraction() {
  Long curDate = new Date().getTime();
  password = generateRandomString(32,
    curDate.toString() + password);
  password =
    Base64.encodeToString(password.getBytes()
    , Base64.DEFAULT).substring(0, 32);
}
```

The password variable is "updated" in every user interaction with the application, and exactly the same code is executed in *afterTextChanged().* The variable is initialized with an empty string, so the first interaction seeds the generator with the timestamp only. The JSON encoded message that's sent to the server includes a timestamp also with millisecond precision (as seen in Algorithm 1), giving a maximum limit to a potential attacker. Although in our experiments we have tested only very short messages, it was possible to find keys estimating time between interactions and processing time.

Assuming the user would launch the app, tap on the message area (1), type two characters (2x3), and use the button (1) that generates the qlink (1); our debugging showed that a total of 9 invocations of the password update would be executed. However, for example, in most cases the first two of the three invocations per character are executed two milliseconds apart.

This observation among others of the behavior of the application and estimations of fixed processing times were considered to write a simple program just for demonstration purposes (since larger messages and broader timing limits would imply incrementing exponentially the brute-force difficulty), that knowing the last timestamp, tries to decipher a two character ("no") message making 10 x (1 x 3 x $130)^{\#characters}$ x 10 x 10 ≈ $2^{9,96\ +\ 8,60(\#characters)}$ tries. A positive result example run of the far from optimized Java program, that on an Intel(R) Core(TM) i-7 notebook would take approximately five hours (two and a half in average), is shown in the following Algorithm 6.

Algorithm 6 Example output from brute-forcing a qlink generated on Android 4.1.1.

```
cantidad de pruebas: 10000 / 152100000 -
  1949 ms
cantidad de pruebas: 20000 / 152100000 -
  1331 ms
...
cantidad de pruebas: 45690000 / 152100000 -
  1206 ms
cantidad de pruebas: 45700000 / 152100000 -
  1200 ms
*** ENCONTRADO
texto en claro: no
password pre pbkdf2:
  bW9pZkR6SWZmTWtmaGF
```

4. Conclusions

We have shown that the manipulation of parameters is possible, random number generation is not implemented in a secure manner, timestamps are used as seeds and key material can be truncated to an extent that may permit brute-force attacks. Other potential problems remains to be probed, for example, if the date and time could be estimated in the way described in [25], to possibly generate the same qlink repeatedly.

Although most of the problems described may not be exploitable or impose a serious security risk, it is clear that the implementation is not following security best practices nor secure programming techniques from, for example, OWASP [26]. Till other reviewers or the developers confirm or reject the potential risks described, sending sensitive information via Qlink.it may not be recommended.

Acknowledgements

We would like to thank Qlink.it developers for their quick response to our queries and their permission for the publication of our preliminary results.

We are grateful to the CACIC2017 anonymous reviewers for their constructive input on our first article [19].

Cristian Borghello is also thanked for his help in the initial summary publication on Segu-Info [10].

Competing interests

The authors have declared that no competing interests exist.

References

[1] A. Castro Lechtaler, M. Cipriano, E. García, P. Lázaro, J. Liporace, E. Malvacio and A. Maiorano. "Posibles problemas en Qlink.it y librería CryptoJS," *in XXIII Congreso Argentino de Ciencias de la Computación*, pp. 1289-1298, 2017. Available at: http://sedici.unlp.edu.ar/bitstream/handle/10915/63936/Documento_completo.pdf?sequence=1. Accessed on 2018-01-05.

[2] "Android app project," Qlink.it Github repository. Available at: https://github.com/qlinkit/androidapp. Accessed on 2018-01-05.

[3] "Qlink Android application." Google Play. Available at: https://play.google.com/store/apps/details?id=com.qlink.easytech.ar. Accessed on 2018-01-05.

[4] "El físico argentino que creó un sistema de seguridad para e-mails." Revista Noticias. Available at: http://noticias.perfil.com/2017/04/09/el-fisico-argentino-que-creo-un-sistema-de-seguridad-para-e-mails/. Accessed on 2017-05-16.

[5] "El acceso a mensajes encriptados por agentes de inteligencia vuelve al foco de debate." Agencia Télam. Available at: http://www.telam.com.ar/notas/201703/183809-el-acceso-a-mensajes-encriptados-por-agentes-de-inteligencia-vuelve-al-foco-de-debate.html. Accessed on 2017-05-16.

[6] "Qlink.it repository on Github." Available at: https://github.com/qlinkit. Accessed on 2017-05-16.

[7] A. Castro Lechtaler, J. Liporace, M. Cipriano, E. García, A. Maiorano, E. Malvacio and N. Tapia."Automated Analysis of Source Code Patches using Machine Learning Algorithms," *in XXI Congreso Argentino de Ciencias de la Computación*, pp. 1016-1022, 2015. Available at: http://sedici.unlp.edu.ar/bitstream/handle/10915/50585/Documento_completo.pdf-PDFA.pdf?sequence=1. Accessed on 2017-05-16.

[8] "AAP project, GICSI repository on Github." Available at: https://github.com/gicsi/aap. Accessed on 2017-05-16.

[9] "Webapp project," Qlink.it repository on Github. Available at: https://github.com/qlinkit/webapp. Accessed on 2017-05-16.

[10] "Posibles vulnerabilidades en Qlink.it (análisis web)," Segu-Info. Available at: http://blog.segu-info.com.ar/2017/05/posibles-vulnerabilidades-en-qlinkit.html. Accessed on 2017-05-16.

[11] "Qlink.it Advanced Frequently Asked Questions." Available at: https://qlink.it/corp/docs/advanced-faq.pdf . Accessed on 2018-01-05.

[12] "mt_rand() reference," PHP manual. Available at: http://php.net/manual/es/function.mt-rand.php. Accessed on 2017-05-16.

[13] "Math.Random()," Mozilla Developer Network. Available at: https://developer.mozilla.org/en-U-S/docs/Web/Javascript/Reference/Global_Objects/Math/random. Accessed on 2017-05-16.

[14] "Random() implementation in CryptoJS." Available at: https://github.com/jakubzapletal/crypto-js/blob/master/src/core.js. Accessed on 2017-05-16.

[15] "XorShift128+ generator implementation," Mozilla. Available at: https://hg.mozilla.org/mozilla-central/file/tip/mfbt/XorShift128PlusRNG.h. Accessed on 2017-05-16.

[16] "XorShift128+ generator implementation, Chrome Github repository." Available at: https://github.com/v8/v8/blob/master/src/base/utils/random-number-generator.h. Accessed on 2017-05-16.

[17] "mt_rand() and mt_srand() functions for bruteforce and speed." Available at: https://github.com/Gifts/pyphp_rand. Accessed on 2017-05-16.

[18] "Symbolic execution for the XorShift128+ algorithm." Available at: https://github.com/douggard/XorShift128Plus. Accessed on 2017-05-16.

[19] "The Z3 Theorem Prover" Available at: https://github.com/Z3Prover. Accessed on 2017-05-16.

[20] "Android platform versions," Android Developers. Available at: https://developer.android.com/about/dashboards/index.html. Accessed on 2018-01-05.

[21] "Security Enhancements in Android 4.2," Android Source site. Available at: https://source.android.com/security/enhancements/enhancements42. Accessed on 2018-01-05.

[22] "Using Cryptography to Store Credentials Safely," Android Developers Blog. Available

at: https://android-developers.googleblog.com/2013/02/using-cryptography-to-store-credentials.html. Accessed on 2018-01-05.

[23] K. Michaelis, C. Meyer, and J. Schwenk, "Randomly failed! The state of randomness in current Java implementations," *in Proc. Topics in Cryptology-CT-RSA*, pp. 129-144, 2013..

[24] "Some SecureRandom Thoughts," Android Developers Blog. 2013. Available at: https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html. Accessed on 2018-01-05.

[25] B. Argyros and A. Kiayias. "I forgot your password: Randomness attacks against PHP applications", *in 21st USENIX Security Symposium*, 2012. Available at: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/argyros. Accessed on 2017-05-16.

[26] "The Open Web Application Security Project (OWASP)." Available at: https://www.owasp.org/. Accessed on 2017-05-16.