Boxcryptor Code Audit

Final Report, 2020-06-24

FOR PUBLIC RELEASE

Contents

1 Summary **2**

2 Executive Summary **3**

3 Methodology **4**

3.1 Code Safety 4

3.2 Cryptography 4

3.3 Technical Specification Matching 5

3.4 Notes 5

4 Findings **6**

KS-SCBX-F-01 XML deserializer potentially allows insecure object deserialization 6

KS-SCBX-F-02 Settings JSON parsing could lead to arbitrary object deserialization 7

KS-SCBX-F-03 Low iteration count for PBKDF2 8

5 Observations **10**

KS-SCBX-O-01 Presence of easily colliding digests 10

KS-SCBX-O-02 Unnecessary and strange bitwise XOR in CryptName 10

KS-SCBX-O-03 The codebase contains dead code or stub functions 11

KS-SCBX-O-04 Padding is handled manually 11

KS-SCBX-O-05 Authenticated encryption uses a hash as a MAC 12

KS-SCBX-O-06 Boxcryptor is not ensuring obliviousness 13

6 About **14**

1 Summary

Secomba is developing Boxcryptor, which encrypts files on the fly to protect them when they are located on a cloud storage, while still allowing synchronisation with multiple devices.

Secomba hired Kudelski Security to perform a security assessment of the Boxcryptor Windows application, providing access to their source code and documentation.

This document reports the security issues identified and our mitigation recommendations, as well as some observations regarding the code base and general code safety. A “Status” section reports the feedback from Secomba’s developers, and includes a reference to the patches related to the reported issues. All changes have been reviewed by our team according to our usual audit methodology.

We report:

- 1 security issue of medium severity
- 2 security issues of low severity
- 6 observations related to general code safety

The audit was performed jointly by Nicolas Oberli – Cybersecurity Expert, and Yolan Romailler – Senior Cryptography Engineer.

2 Executive Summary

Kudelski Security has audited the codebase and documentation provided by Secomba. It included:

- the Boxcryptor Windows Desktop App source code
- internal documentation made available by Secomba
- and the one on https://www.boxcryptor.com/en/technical-overview/.

The application uses standard library functions and standardized methods as often as possible and we notably covered the following main components among others:

- `Secomba.Common`: contains most notably the cryptographic operations
- `Secomba.Common.Net45`: which is mostly a proxy to the crypto provider
- `Boxcryptor.Core`: contains actual encryption logic, has high level APIs and does the PKI management
- `Boxcryptor.Desktop`: contains the file operations, including bulk ones and read/write operations.

All these components were logically correct and did not show any significant weakness under scrutiny. It is important to note that the codebase we audited was not showing any signs of malicious intent neither. The few findings we had were not critical flaws, or dangerous design mistakes. As such, we are confident the Boxcryptor codebase we audited is actually doing what it is expected to do.

Overall the cryptographic primitives used and the architecture of the system are adequate to achieve the security goals we were presented.

Finally, in respect to the code made available for review, Kudelski Security researchers worked with the assumption that the code in exam will actually be used by the deployed application. In practice, this cannot be checked without reverse engineering the application executable whenever there are updates, or without a trusted build system.

3 Methodology

In this code audit, we performed four main tasks:

1. informal security analysis of the original protocol;
2. actual code review with code safety issues in mind;
3. assessment of the cryptographic primitives used;
4. compliance of the code with the technical documentation provided.

This was done in a static way and no dynamic analysis has been performed on the codebase. We discuss more in detail our methodology in the following sections.

3.1 Code Safety

We analyzed the provided code, checking for issues related to:

- general code safety and susceptibility to known vulnerabilities;
- poor coding practices and unsafe behaviour;
- leakage of secrets or other sensitive data through memory mismanagement;
- susceptibility to misuse and system errors;
- error management and logging.

3.2 Cryptography

We analyzed the cryptographic primitives and components, as well as their implementation. We checked in particular:

- matching of the proper cryptographic primitives to the desired cryptographic functionality needed;
- security level of cryptographic primitives and of their respective parameters (key lengths, etc.);

- safety of the randomness generation in the general case and in case of failure;
- safety of key management;
- assessment of proper security definitions and compliance to the use cases;
- checking for known vulnerabilities in the primitives used.

3.3 Technical Specification Matching

We analyzed the provided documentation, and checked that the code matches its specification. We checked for things such as:

- proper implementation of the documented protocol phases;
- proper error handling;
- adherence to the protocol logical description.

3.4 Notes

It is important to notice that, although we did our best in our analysis, no code audit assessment is per se guarantee of absence of vulnerabilities. Our effort was constrained by resource and time limits, and in the scope of the agreement between Secomba and Kudelski Security.

In assessing the severity of some of the findings we identified, we kept in mind both the ease of exploitability and the potential damage caused by an exploit.

4 Findings

This section reports security issues found during the audit.

The “Status” section includes feedback from the developers received after delivering our draft report.

KS-SCBX-F-01: XML deserializer potentially allows insecure object deserialization

Severity: Medium

Description

Under certain circumstances, the `XmlSerializer` class allows for unsafe objects deserialization. In the code, this serializer is used for parsing various messages coming from online storage services but also within the WebDAV client, which is more problematic since the server could be controlled by a malicious actor.

We reviewed the `Secomba.Common/Storage/Implementation/WebDAV/WebDAVXml.cs` file that contains the class definitions for WebDAV messages, and the only possible problematic element is the `Collection` abstract class which comes from the `System.Collections` package.

Recommendation

While we could not make sure that this object could be used to load arbitrary objects, we would recommend reviewing this object and make sure that a type is defined for the `Collection` class.

For more information about objects deserialization, please see:

- https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A8-Insecure_Deserialization
- https://speakerdeck.com/pwntester/attacking-net-serialization

Status

The code in question being legacy code, Secomba has removed this section entirely. They also checked all other locations where XML files are handled and made sure none of them use object serialization.

KS-SCBX-F-02: Settings JSON parsing could lead to arbitrary object deserialization

Severity: Low

Description

The application settings are stored in flat files in the `%APPDATA%` directory in JSON format. These settings are then deserialized using a library called Json.Net, which in certain conditions allow for arbitrary objects deserialization.

If an attacker is able to modify the application settings, it is possible to make the JSON deserializer load a malicious object and ultimately execute code in the Boxcryptor execution context. This is caused when the `TypeNameHandling` property of the deserializer is set to an other value than `None`.

During this audit, we found two occurrences of the problem :

```
// Secomba.Common/Storage/Settings/SettingsStorage.cs:106

106 loadedSettings = JsonConvert.DeserializeObject<T>(serializedSettings,
107   new JsonSerializerSettings {
108     TypeNameHandling = TypeNameHandling.Objects,
109     Formatting = Formatting.None
110   });
```

```
// Secomba.Common.Analytics/Pipe/Persister.cs:105

105 var typeNameHandlingSettings = new JsonSerializerSettings {
106   TypeNameHandling = TypeNameHandling.All
107 };
```

Fortunately, none of the serialized object prototypes allow for unsafe objects to be loaded, so this does not induce a vulnerability by itself.

Recommendation

Use a `SerializationBinder` to only allow defined typed to be loaded. (see (https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.serializationbinder?view=netframework-4.5) for more information).

If possible, disable `TypeNameHandling` totally when serializing settings, and use a simple data structure and instantiate objects in the code.

For more information about objects deserialization, please see:

- https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A8-Insecure_Deserialization
- https://speakerdeck.com/pwntester/attacking-net-serialization

Status

Secomba has implemented the recommended `SerializationBinder`.

KS-SCBX-F-03: Low iteration count for PBKDF2

Severity: Low

Description

In the `Boxcryptor.Core/Encryption/EncryptionService.cs` file, we can see that the constant `PasswordKdfIterations` is set to 5 000, which is not consistent with the constant `DEFAULT_KDF_ITERATIONS` set to 10 000 in `Boxcryptor.Core/UserManagement/Domain/User.cs`.

While the key generation code appears to be using using the latter, a value of 10 000 remain relatively low as per today's standards and computing power.

Recommendation

We recommend using a higher value, such as 100 000 is this is not impacting performance too much. This is motivated by the fact that user supplied passwords are typically low entropy, and thus one wants to increase the difficulty of a dictionary attack as much as possible.

We would also recommend setting a default minimal length to the user supplied passwords. A back-of-the-envelope calculation tells us that an 8 characters password has typically 48 bits of entropy, while 10 000 PBKDF2 are considered to "add" roughly 13 bits to it, which means we only have a security level of roughly 61 bits, which is too

low as per the latest advances[1] in computing power and brute force. Increasing the PBKDF2 to 100 000 barely crosses the 64 bit marks.

Better than increasing the PBKDF2 iteration count, if FIPS compliance is not required, we recommend migration to Scrypt or Argon2 as it would further increase the security of the system since these are memory-hard.

Status

Secomba has enforced a password length of at least 8 characters in a first iteration, and will think about adding more restrictions to ensure better entropy in a second iteration.

The iteration count for PBKDF2 will remain at 10 000, because it appears some client devices (Internet Explorer, mobile devices) are still too slow to justify an increase to 100 000 iterations. Secomba has increased the iteration count for the password hash, `PasswordKdfIterations` to 10 000 as well, to have equal security properties. However this iteration count is no longer fixed in the .NET codebase, but delivered by an endpoint on Secomba's servers. This has also been updated in the documentation in https://www.boxcryptor.com/en/technical-overview/. Notice that the PBKDF2 salt is now also fully randomized, generated by the client upon creation, but stored on the servers.

Existing users will go through a staged roll-out, while for new users the change is effective immediately.

[1] See for instance how https://sha-mbles.github.io/ showed the 64 bits level was breached.

5 Observations

This section reports various observations that are not security issues to be fixed, such as improvement or defense-in-depth suggestions.

KS-SCBX-O-01: Presence of easily colliding digests

In certain parts of the code base, the following function found in `Secomba.Common/Cryptography/HashUtils.cs` is used:

```
17 public static ulong Mac64(byte[] data) {
18         var h = new byte[8];
19         for (var i = 0; i < (data.Length - 1); ++i) h[i % 8] ^= data[i];
20 
21         var value = (ulong)h[0];
22         for (var i = 1; i < 8; ++i) value = (value << 8) | h[i];
23 
24         return value;
25 }
```

But this is basically a rolling XOR sum with an 8 bytes window, which is very easily susceptible to collisions. The different usages of it do not appear to be security issues.

However, we recommend not using this function for other purposes, except when required to comply with the EncFS scheme.

Status

Secomba has moved the `Mac64` method and other similar ones into the filename encryption class and made them private, so they can only used for that purpose. They removed all possibilities to call them from somewhere else. As such, the insecure methods are only usable and used for their original purpose in the frame of the EncFS scheme used by the filename encryption, reducing significantly the risks of future misuses and improving maintainability.

KS-SCBX-O-02: Unnecessary and strange bitwise XOR in CryptName

In the function `CryptName`, in `Boxcryptor.Core/Encryption/FilenameCipher.cs`, one can read:

```
118 public byte[] CryptName(byte[] source, IAesKey key, byte[] mac, CryptoStreamMode
    ↪  streamMode) {
119         byte[] temp;
120         byte[] iv = GetIv(key, mac);
121         ulong seed = (HashUtils.Mac64(iv)) ^ 0;
```

Notice that here the 0 is hopefully a `ulong` as well, and as such this XOR is not reducing the seed to a mere 8 bits integer as one could have feared, yet this bitwise XOR with 0 is not doing anything and does not seem to serve any purpose.

We recommend to avoid such dummy or unnecessary code as much as possible, especially when there are no comment explaining it, as it does not help with maintainability of the codebase.

Status

Secomba has removed it.

KS-SCBX-O-03: The codebase contains dead code or stub functions

For example the `SecAesPlainServiceProvider` class appears to be an insecure stub class, as it does only implement padding without any actual encryption, but it is also unused in the codebase we were provided.

We recommend removing dead code as much as possible, and also to add comments to stub functions that are not meant to be secure, in order to increase maintainability of the code.

Status

Such unused code will be removed upon discovery. In particular, Secomba has removed the `SecAesPlainServiceProvider` class completely.

KS-SCBX-O-04: Padding is handled manually

It appears that in the current codebase, padding is often handled manually. We recommend considering moving padding management into a self contained function in order to ease migration to newer padding schemes, shall it be required in the future, or to completely rely on the Crypto Provider to handle padding as it should.

This notably occurs in:

- `Boxcryptor.Core/IO/SecFile.cs:624-642` is doing PKCS7 padding.
- `Secomba.Common.Net45/Cryptography/Implementations/SecAesPlainServiceProvider.cs:39-43` is doing PKCS7 padding.
- `Boxcryptor.Core/Encryption/FilenameCipher.cs:198-201` is doing zero padding, which is fine with CFB, since it really is just a keystream XORed with the plaintext. But one would expect the Crypto Provider to handle this directly.
- `Secomba.Common/Cryptography/CryptoConvert.cs:59` is using left padding with zeros, which is fine for big number as it is used, but it might be documented.

As a whole we recommend documenting padding, and the reasons for it whenever it is performed, as padding oracles are a threat we recommend keeping in mind, especially when using AES CBC.

Status

Secomba has removed the custom padding occurrences, except for the one in `SecFile.cs` as it is part of an experimental feature that is not in production yet and not activable by the users, and the one in `CryptoConvert.cs` where it is expected to be done like that and has now been documented.

KS-SCBX-O-05: Authenticated encryption uses a hash as a MAC

In the `SecRsaCryptoServiceProviderBase` class, the functions `EncryptAuthenticated` and `DecryptAuthenticated` are relying on the SHA256 hash of the input data as a MAC, but the input data is simply the plaintext (since this is done in a MAC-then-encrypt fashion). Since a MAC must be unforgeable under a chosen message attack for it to be a MAC, a simple hash function cannot be used as a MAC unless it's keyed, which it is not there. Notice this is not the case for the `SecAesCryptoServiceProviderBase` class, where a keyed hash function is used as MAC.

Status

This is known behavior and is documented in Secomba's internal documentation.

At this point, Boxcryptor is not enforcing integrity and is not advertised as doing so either. Adding effective integrity is a future goal.

The incorrect naming is a deprecated left-over, implemented back then for consistency between the interfaces. Newer code is no longer calling this mechanism an actual MAC.

Thanks to the public nature of RSA public keys, everybody can legitimately create new, validly encrypted RSA packages for the said keys (which is necessary for sharing). This means that even if Boxcryptor were to enforce integrity, valid RSA encrypted packages could still be forged.

KS-SCBX-O-06: Boxcryptor is not ensuring obliviousness

As it currently stands, Boxcryptor allows users to access files stored on an untrusted server in such a way that the server does not learn the contents of the files, however the server can still monitor the access patterns of which files were accessed at which times. The study of "Oblivious RAM" (ORAM) protocols is an active field of research in cryptography whose goal is to protect both the content and the access patterns from the monitoring of an untrusted server.

While the loss of privacy caused by the leakage of the access patterns of given files (especially if files have encrypted file names) is arguably concerning, we do recommend that Secomba monitors closely the evolution of existing ORAM schemes and the emergence of new solutions, as these are particularly well suited to store data on untrusted servers and are not necessarily unpractical.

Status

Secomba will monitor the evolution of ORAM as suggested and consider it for future versions of Boxcryptor.

6 About

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com or https://kudelski-blockchain.com/.

Kudelski Security
Route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland