Assessing the *No-Knowledge* Property of SpiderOak ONE

Anders P. K. Dalskov, Claudio Orlandi
Aarhus University, Aarhus, Denmark

June 12, 2017

Abstract

This paper presents the findings of an independent security review of *SpiderOak ONE*, a popular encrypted cloud storage application. In this application, the storage provider claims that, since all the users' data is password encrypted and the password never leaves the client, even the storage provider cannot learn any information about the users' data. After providing a formal description of the key design choices in the reviewed application (e.g., how user's accounts are registered, how new devices are registered, how and what cryptographic keys are used, how file encryption is handled, etc.), we present a number of vulnerabilities that can be exploited by a malicious storage server to break, to different degrees, the confidentiality of the users' password and therefore the users' data.

Our findings have been communicated to SpiderOak in April 2017. The vendor promptly replied to our concerns by releasing an updated version of the application (v. 6.3.0, June 2017) which resolves most of the issues described in this paper.

1 Introduction

More and more users worldwide choose to store their data using cloud storage services such as Dropbox, Google Drive, Microsoft Azure, etc. (Dropbox alone recently celebrated reaching half a billion users[1]). These services give users a transparent way to share their data between multiple devices, they allow to share files between users, and provide a relatively cheap way for keeping personal backups in the cloud.

Unfortunately, "classic" cloud storage solutions provide little or no guarantee about the confidentiality of the data that the users' choose to store in the cloud. While most of these services guarantee that the data is encrypted in transit to protect against network eavesdropper, no mechanism prevents the storage provider itself from accessing the users' data. (In fact, the economic viability

[1] `https://blogs.dropbox.com/dropbox/2016/03/500-million/` (all links last retrieved on June 9th 2017)

of some of these systems relies on being able to identify multiple copies of the same data being stored, and thus being able to implement de-duplication techniques. However, using de-duplication might allow external attackers to learn information about other users's data. See [9] for a description of the problem and [8, 12] for some cryptographic solutions which allow to perform de-duplication in a secure way).

As a consequence of the Snowden revelations on the existence of mass-surveillance program many users and companies have started questioning the choice of storing their (potentially sensitive) data on unencrypted cloud storage, especially on foreign soil (the legal controversy around the status of the EU-US Privacy Shield is still ongoing).

All of these factors lead to an increased interest in encrypted cloud storages e.g., services which offer to store the user's data in an encrypted format, in such a way that even the service provider themselves cannot access the users' data. This is a very useful property, which has an important impact against several interesting threat models: if the cloud storage is *technically unable* to access the user's data, the provider cannot be coerced (e.g., by law enforcement) to reveal the content of the user's encrypted storage; also, since the users' data is only stored in encrypted format (and the password is unknown to the server), even if someone could gain access to the cloud storage system (e.g., either a malicious employee or an external attacker), this would not help in compromising the users' data.

SpiderOak[2] is among the most popular encrypted cloud storage services offering end-to-end encrypted cloud storage. SpiderOak received popular attention after being endorsed by Edward Snowden as a secure alternative to Dropbox[3], and has received positive reviews by the EFF [6]. SpiderOak in particular marketed their product using the term *zero-knowledge* (now replaced by *no-knowledge*[4]), capturing the property that even SpiderOak themselves have no way of accessing the content of the encrypted users' storage.

In a nutshell, virtually every encrypted cloud storage, including SpiderOak, stores the users' data encrypted under a user chosen password. Therefore, whether the *no-knowledge* property holds ultimately relies on two factors:

1. The user must choose a strong password; and
2. The user's password must never leave the client's software;

Much has been written about the (in)ability of users to choose strong passwords, so we will not address this threat further in this paper. What is perhaps most interesting, from a technical point of view, is to look at the service provider choices in protocol design and software implementation to ensure that no one, even the service provider itself, can extract the user's password from the client software. SpiderOak is very explicit about this: for instance, users attempting

[2] `https://spideroak.com/`

[3] `https://techcrunch.com/2014/10/11/edward-snowden-new-yorker-festival/`

[4] `https://spideroak.com/articles/why-we-will-no-longer-use-the-phrase-zero-knowledge-to-describe-our-software`

to login using the web interface (which would reveal the password to the server) are required to acknowledge the following message[5]:

> I understand that for complete 'Zero-Knowledge' privacy, I should only access data through the SpiderOak desktop application.

Our Contributions. In this paper we present an independent security review of the *SpiderOak ONE* client software with the goal of assessing to which degree SpiderOak satisfies the *no-knowledge* property. We choose to start by analysing SpiderOak due to its popularity, and we believe that it would be interesting, in the future, to perform similar studies of other encrypted cloud storage apps. In particular, we are not claiming that SpiderOak is less secure than any of its competitors (in fact, since SpiderOak has already fixed most of the problems described in this paper it is entirely possible that the current version of SpiderOak is *more* secure than its competitors who have not gone through an independent security review yet).

Since SpiderOak is not an open source application, the first contribution of this paper is to present (in section 3) a high-level overview of how SpiderOak handles the user's password in some critical phases (such as user registration, authentication, file encryption etc.). In section 2, we describe our methodology and how we have inferred the behaviour of the client software. Finally, in section 4, we present the main contribution of this paper i.e., four possible attacks (three active, one passive) that would allow a rogue SpiderOak server (or a man-in-the-middle who can circumvent the TLS layer) to break to different degrees the *no-knowledge* property.

To understand the significance of these attacks, we need to take a step back and describe the threat-model that we consider in this paper: it is evident that, since SpiderOak is a closed source application and since users retrieve the client software from SpiderOak itself, a malicious SpiderOak server could always serve a targeted user a different copy of the client software that, on purpose, leaks the user password to the server.[6]

We ignore this threat in the remainder of the paper and we will instead assume that the user can trust SpiderOak to deliver a benign version of the client software. After this initial phase our threat model includes any attacker that is able to interact with the client-software. Note that, due to the use of TLS and certificate pinning, it is not easy to convince the client software to interact with anyone else but the real SpiderOak server. In particular, we conclude that the only entities which will be able to run the attacks described here are:

1. A rouge SpiderOak server (which could become compromised due to external hacking, insider attacks, etc.);

[5] `https://spideroak.com/browse/login/storage`

[6] SpiderOak, partially inspired by our research, has recently published a blog post discussing these threats, see `https://spideroak.com/articles/building-for-new-threat-models-in-a-postsnowden-era`.

2. A rogue SpiderOak enterprise server (in enterprise mode a company can internally run a SpiderOak server); or
3. Anyone else able to bypass certificate pinning or the TLS encryption layer (e.g., certificate pinning can be turned off in order to inspect traffic, which is not uncommon in corporate settings);

Responsible Disclosure. We have communicated our findings to the security team of SpiderOak on April 5th, 2017. On June 5th 2017 SpiderOak released a new version of the software which resolves most of the issues described in this paper. SpiderOak notified their users by email and released a blog post about this[7]. We find it commendable that SpiderOak has reacted so swiftly to the issues we found.

Other Related Work. There seem to be relatively little previous work that analyses the security of encrypted cloud storage solutions. Some notable examples include: Kholia and Węgrzyn [13] analyzed Dropbox and described several security vulnerabilities in a threat-model very different from ours (since Dropbox never claimed any "no-knowledge" property). Grothe et al. [7] analyzed Microsoft Azure and its interaction with Tresorit (a solution supporting client-side encryption), showing that when a users shares a file with another user, Tresorit could decrypt this file as well. In an extensive technical report Botgmann et al. [2] examined the security mechanisms of several cloud storage services (but not of SpiderOak). Virvisil et al. [16] provide a good survey of challenges and solutions for secure cloud storage. Bhargavan and Delignat-Lavaud [1] presented attacks against the web-based interface offered by SpiderOak (concretely, the interface related to shared directories). Finally, Wilson and Ateniese [17] have also analyzed the security of client-encrypted cloud storage, including SpiderOak, however they only focus on the ways in which the applications could learn information about files shared by the user.

Therefore, to the best of our knowledge, ours is the *first* work uncovering problems with encrypted data storage of *any* encrypted cloud storage solution which involves only data at rest.

2 Analyzing the Application

Our analysis was performed on the SpiderOak ONE desktop client version 6.1.5 (released 26–07–2016) as can be downloaded from their website.[8] The client was run in a Windows XP virtual machine.

In section 4 we attack a client (also version 6.1.5) running on a GNU/Linux virtual machine, however, the differences between operating systems are minimal and will be explicitly mentioned where appropriate.

[7] `https://spideroak.com/articles/security-update-for-spideroak-groups--one-bugs-reported--resolved`

[8] `https://spideroak.com/opendownload`. A 32-bit version (which we used) can be retrieved from `https://spideroak.com/getbuild?platform=win32`

2.1 Introduction to SpiderOak

SpiderOak ONE runs on both Windows, GNU/Linux and Mac OS X, and SpiderOak provides builds for both `i386` and `x86_64` (i.e., 32 and 64 bit variants), as well as various packaging formats (e.g., `rpm` and `deb`).

Registration must happen through the desktop client, and SpiderOak warns against using their website for logging in (as mentioned in the introduction).

Once the user has acquired an account, the application creates a directory named *SpiderOak HIVE*, where files herein gets automatically backed up. In addition, SpiderOak ONE uses the notion of a *Sync* folder (the aforementioned *SpiderOak HIVE* being the default) which is a folder that is kept synchronized across the user's devices. The user can select other folders that should be backed up and turn them into Syncs as well. These Syncs allows the user, for example, to have a directory be synchronized on two of his devices, but not on a third. Finally, the user can select single files or whole directories, and share them. Sharing a file or folder makes it available through SpiderOak's website so others (who do not necessarily have SpiderOak ONE installed) can access them. SpiderOak ONE is able to handle multiple versions of a single file, making it possible to revert to historical versions of a file, or restore deleted files.

By default, the application only requires the user to input their password the first time they login. I.e., when they create their account or register a new device. It is possible to set the application to require a password on every startup, however.

2.2 Under the Hood

SpiderOak ONE gets installed at `C:\Program Files\SpiderOakONE` (or at `/opt/SpiderOakONE` on GNU/Linux). After the first execution, the application furthermore creates a directory for run-time specific files (in `Local Settings\Application Data` on Windows and `$HOME/.config` on GNU/Linux).

Libraries and Files. Browsing the install directory reveals that the application makes use of various mature open source libraries. Examples include OpenSSL version 1.0.1t, released 3th May 2016; libsodium version 1.0.0, released 30 September 2014; py-bcrypt version 0.4 released 25 August 2013; and Twisted Matrix version 10.2.0 released 29 November 2010. An interesting observation is the apparent age of some of these libraries. For example, the version of py-bcrypt used is fairly old and not maintained anymore (and we show later that it contains an exploitable bug). Similarly for the version of Twisted used (For example, it is still contains the *httpoxy*[9] vulnerability, although it is not an issue in SpiderOak ONE as it does not use CGI scripts).

The run-time directory contains various files used to persist settings across reboots, as well as diagnostic data from the application. For example, in order to not require a password on every startup, the application will write the user's

[9] `https://httpoxy.org/`. Fixed in Twisted version 16.3.1

password un-encrypted to file. If the user has set the application to require a password, the user's password will not be stored in cleartext. A very weak password hash, however, will (and we return to this issue in subsection 4.3). In addition, the run-time directory also contains the output of logging statements from the application, providing a very convenient way of tracking the behaviour of the application when it is running.

Reverse Engineering. SpiderOak ONE, being proprietary[10], provides no readily available source code. The first step in analyzing the application was therefore to obtain a readable copy of the application's code. Browsing the application files after installation, revealed that the application is written in Python, bundled as zipped archive containing Python bytecode. Unlike e.g., Dropbox which uses a modified interpreter and obfuscates its bytecode [13], SpiderOak ONE does no such thing, making the reverse engineering relatively straightforward. Using the open source tool *uncompyle6*, a Python decompiler[11], the application's bytecode was converted into normal Python code. Since there is no obfuscation, and due to the nature of Python bytecode (in particular, it does not aim to optimize the code or minimize its size) the obtained Python code has a lot of information, such as proper variable names, documentation strings and so on.

The client application communicates with the server in two different ways: Using HTTP over TLS (HTTPs) and using a Perspective Broker[12] implementation, also over TLS. Both types of connections use certificate pinning, although with different certificates.

2.2.1 HTTPs

We observed essentially three different situations in which the application will use HTTPs for communicating with the server: Account registration, device registration and sharing of single files. The TLS version used is 1.0 using the cipher suit `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA`. The certificate presented by SpiderOak is issued by GeoTrust and uses the signature algorithm `sha256WithRSAEncryption`. When received the certificate is checked against a pinned (i.e., hard-coded) root certificate in the client. SpiderOak ONE checks that the Common Name (CN) field in the certificate presented by the server, matches either `spideroak.com`, `*.spideroak.com` or `*.backupsyncshare.com`. Although the last name seems out of place, its DNS entries point to SpiderOak servers, so we assume that it is also owned by SpiderOak. Checking the CN is necessary to prevent a Man-in-the-Middle attack where the certificate used is signed by the correct CA, but where the CN is unrelated to the requested

[10] See *Open Source* section at `https://spideroak.com/features/private-by-design`

[11] `https://github.com/rocky/python-uncompyle6`

[12] A serialization and Remote Procedure Call abstraction from the Twisted library: See e.g., `https://twistedmatrix.com/documents/12.1.0/core/howto/pb-intro.html`

site, as described in [3]. Validation of the certificate signature is handled by OpenSSL.

2.2.2 Perspective Broker (PB)

All other communication — synchronizing stored files, settings etc. — is handled by a Perspective Broker (PB) class. Roughly speaking, PB is a Remote Procedure Call interface that can handle transmission of both simple and complex data types. When the client has performed a successful login, it will send a copy of its *avatar* (a reference to a class with methods that can be called remotely) to the server. Once received by the server, the server then sends back its *avatar*. Thus allowing the client to call remote procedures on the server, and vice versa.

Communication done in this fashion is protected by TLS, also version 1.0, although `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA` is used as the cipher suit instead. The SpiderOak server presents in this context a self-cipher certificate which is checked against an identical certificate that is hard-coded in the application. We note that SpiderOak ONE does not use the certificate described in the previous section here. Since the certificate presented by the server is also pinned in the client, there is no need to validate the CN field (in fact, it is not even present). A final, somewhat interesting observation here, is the fact that the signature algorithm is `sha1WithRSAEncryption`, which has been deprecated everywhere else, especially after the recent reveal that collisions for SHA1 have been found [15].

2.2.3 Patching of the application

In order to better understand and test SpiderOak ONE, we performed small patches. As mentioned, the application does not make use of obfuscation techniques which made the decompilation and patching easier. Indeed, in order to introduce a patch in the application, we simply had to write it in python, compile the new file and put it back into the correct directory.

We created two types of modifications. One for helping us analyze the network traffic generated by the application and one which helped us understand the flow of the program (which was particularly helpful for the sake of understanding what files gets encrypted with which keys).

Network Analysis. In a nutshell, we patched the application to simply write to the disk the master secret for any TLS connection it established. This modification was easy to implement, as all TLS communication was handled by a single class (from the Twisted library), but also because the version of Twisted used was old enough, that it exposes some particularly low level OpenSSL objects. Specifically, a connection object which has a method to retrieve a master secret for the connection in question. (In the code snippet, `cheating` is a PyOpenSSL Connection object[13]).

[13] `https://pyopenssl.readthedocs.io/en/latest/api/ssl.html`

```
if self.cheating:
  _k = self.cheating.master_key()
  if _k:
    _cr = str(self.cheating.client_random())
    _cr = _cr.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
```

Analyzing the network was then simply a matter of recording all traffic (which we did by routing network traffic from our test machine through another virtual machine, running *tcpdump*) and then have *wireshark* decrypt it.

Program Flow. We took advantage of the already fairly comprehensive logging framework present in the application, in order to better understand the program flow. Concretely, we employed what is essentially "debugging by printing" by inserting more logging statements in places where we needed to understand what happened. The output of these logging statements could then be found in the run-time directory (as mentioned earlier). By using some prefix not used elsewhere in the application (e.g., "`!!!`") grepping for the relevant outputs became easy.

3 Protocols

From the decompiled application code we extracted a formal description of some core aspects of SpiderOak ONE, namely

1. how SpiderOak ONE handles authentication in the context of a new user account, or a new device registration for an existing device;
2. how it handles and creates the various cryptographic keys needed; and
3. how it handles encryption of a users personal files.

Notation. For a bit-string x, let $|x|$ denote its length in bits and $|x|_8$ its length in bytes. $a \mid\mid b$ is the concatenation of bit-strings a and b; and by $x_{i:j}$ we mean the sub-string composed of the bits $x_i, \ldots, x_{j-1}$. Let $\mathrm{Enc}_k(iv, m)$, respectively $\mathrm{Dec}_k(iv, m)$, denote an AES-CFB encryption, respectively decryption, using initialization vector iv, key k on message m. For a description of CFB refer to [5]. Unless otherwise stated, the segment size is 8 and as a consequence, $|m|$ must be a multiple of 8 (which is always the case if m is a byte-string). Let $\mathrm{RSAEnc}_{pk}(m) = m^e \mod n$ denote a textbook RSA encryption on $m \in \mathbb{Z}_n$

using key $pk = (e, n)$, decryption is $\text{RSADec}_{sk}(m)$ and an RSA signature is $\text{Sign}_{sk}(h)$ where $h = H(m)$ for some hash function H and m is the message being signed.

Finally, for some password p and salt s, let $\text{bcrypt}(p, s)$ denote a bcrypt hash [14] and let $\text{PBKDF2}(p, s, n)$ denote a PBKDF2 hash [11] (n being the iteration count).

3.1 Registration protocols

SpiderOak ONE handles registration of both new user accounts, as well as new devices to existing accounts. In either case, the client and server execute a challenge-response authentication protocol, the concrete format of which depends on what type of registration (i.e., account or device), is taking place.

In total, the client can be made to engage in four different authentication protocols: **bcrypt**, **pandora/zk**, **escrow/challenge** and **pandora/zk/sha256**.[14] We will present all but the last one in this paper. At the end of the section, we will give a brief description of the whole account and device registration process.

Note that our analysis is based only on the client-side version of the software. We therefore resort to some conjectures on how the server-side version of the application is supposed to run. The accuracy of these conjectures is irrelevant to our analysis, since we want the *no-knowledge* property to hold even against a corrupt SpiderOak server.

3.1.1 bcrypt

The following conceptually simple challenge-response protocol is executed whenever a user registers a new account through the client.

Server Send bcrypt salt s to client.

Client Compute $h = \text{bcrypt}(p, s)$, p being a password input by the user and send h to the server.

Server Compare h to a stored hash and accept if they are equal. Otherwise reject.[15]

During account registration, before this protocol is used, the client will have transmitted h to the server. Thus, at the point where this protocol gets executed, the server already possesses h.

When the server is allowed to act maliciously, the protocol is not without issue (which is somewhat surprising given its simple description). We explore this further in subsection 4.1.

[14] The names reflect those used internally in the application.

[15] Concretely, the server will reply with an HTTP code 200 (accept) or 403 (reject). This holds for all authentication protocols in the application.

3.1.2 pandora/zk

This next protocol is executed during device registration. After the client has computed all the user's cryptographic keys, these are transmitted to the server (a process we will detail in sections 3.1.4 and 3.2). However, for the sake of this protocol, we jump and define two of these values here, namely $ck = \text{PBKDF2}(p, s_1, 16384)$, a *challenge key*; and s_1 a 256-bit salt (p is the password input by the user during account registration). The server and client then execute the following protocol.

Server Let k be a 256-bit value and iv a 128-bit value. Let tv be a 32-bit value denoting the current server time. Send iv, tv, s_1 and $c = \text{Enc}_{ck}(iv, k)$ to the client.

Client Compute $ck^* = \text{PBKDF2}(p^*, s_1, 16834)$, $k^* = \text{Dec}_{ck^*}(iv, c)$ (p^* being a password input by the user) and reply with $a = \text{Enc}_{k^*}(iv, tv)$ to the server.

Server Abort if $\text{Enc}_k(iv, tv) \neq a$. Otherwise accept.

It is clear that the client can compute the same ck as used by the server, if the user input the same password in this protocol, as during the account registration. Without knowing the description of the server-side of the protocol, it is not possible to give a full analysis of the soundness of this protocol (e.g., whether it is possible to make the server accept without knowing ck). We note however that the knowledge of ck is always enough to make the server accept, and that therefore the protocol is not a cryptographic *proof of knowledge* of the password p.

3.1.3 escrow/challenge

In contrast to the protocols presented so far, we have never observed the **escrow/challenge** protocol being invoked during our analysis of the application and it appears that the protocol is only used in the enterprise setting. Nevertheless, the client software can be made to engage this protocol and we later show (in subsection 4.2) that this protocol can be used by a malicious server in order to fool the client into revealing the user password.

We assume the server has a (possibly empty) list l of pairs (pk_i, id_i), where pk_i as an RSA public key and id_i is an arbitrary ID. For the sake of presentation we first describe two subroutines *Fingerprinting* and *Layered Encryption*

Fingerprinting. The client will compute a fingerprint (concretely a string of 1 to 4 letter words) from l using [4], pick every second word and present the resulting string to the user. Let $\texttt{E}(x)$ be a function that returns x encoded according to the *Distinguished Encoding Rules* scheme (defined in ITU-T X.690) and $\texttt{key2eng}(h)$ be the function that converts a 256-bit binary string h into a list of 24 words according to [4]. The process for creating a fingerprint is then

1. Compute $h = \text{sha256}(id_0 \mid\mid \texttt{E}(pk_0) \mid\mid \ldots \mid\mid id_n \mid\mid \texttt{E}(pk_n))$;
2. Compute $\texttt{key2eng}(h) = x_1 \mid\mid y_1 \mid\mid \ldots \mid\mid x_{12} \mid\mid y_{12}$; and
3. Output the fingerprint $fp = y_1 \mid\mid \ldots \mid\mid y_{12}$.

We speculate that the choice of only using every other words is a question of usability; 12 words are easier to recognize than 24. An example of fingerprint derived in this fashion can be seen in Figure 1.

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

Figure 1: Example fingerprint. Computed using $h = \text{sha256}()$, i.e., l is an empty list.

We use $\textsc{Fingerprint}(l)$ to denote a function that outputs a fingerprint computed in the described way, on a list l of keys and ids.

Layered Encryption. The following process describes an approach for creating a layered encryption of the user's password p, using the contents of l. Let c some arbitrary value obtained from the server.

1. Let $auth$ = "{"*challenge*" : c, "*password*" : p}", i.e., a JSON string. Then, for all id, pk in l do
   (a) Let k be a random bit-string s.t. $|k| = |pk| - 1$ and let $iv = \text{sha256}(tv)_{0:16}$ where tv is the current system time.
   (b) Compute $A = \text{Enc}_{\text{sha256}(k)}(iv, auth)$, $B = \text{RSAEnc}_{pk}(k)$
   (c) Re-assign $auth = id_i \mid\mid A \mid\mid B \mid\mid iv$
2. Output $auth$

As with the $\textsc{Fingerprint}$ function, we also define a function, $\textsc{LayerEnc}(p, l, c)$ to denote the above process, using inputs p, user's password; l, a list of keys and ids; and c some arbitrary value.

The actual protocol can then be summarized quite nicely as follows (note, we still assume the server to own some list l of keys and ids)

Server Let c be an arbitrary bit-string and send c, l to the client.

Client Compute $fp = \textsc{Fingerprint}(l)$ and prompt the user to either accept or reject the fingerprint. If the user rejects fp, the client aborts the protocol. Otherwise it computes $auth = \textsc{LayerEnc}(p, l, c)$ and sends $auth$ to the server.

We leave out how the server acts when given $auth$. As mentioned, we have not witnessed this protocol during interaction with a real SpiderOak server and will in this context refrain from speculating on what the server does in a real setting.

3.1.4 Account Registration

When users register a new account, they input an email *email*, password p, name *name* and optional password hint, after which the client and server perform the following protocol

Client Compute a random bcrypt salt s with a cost factor of 12. Send $h = \text{bcrypt}(p, s)$, *email* and *name* to the server.

Server If *email* is already registered, abort. Otherwise pick a username $u =$ `u_spideroak_auto_`n (n being an integer chosen such that u is unique among SpiderOak users) and send u back to the client.

Both Execute the **bcrypt** protocol from subsubsection 3.1.1 and continue if it succeeds.

Client User inputs a *device name dname* and sends it to the server.

Server Let rt be a 256-bit *reinstall token*, $did = 1$ the *device ID* and send did, u, rt to the client.

Client Compute a list of keys kl according to subsection 3.2 and send it to the server.

An interesting note is the fact that there are two usernames associated with an account: The *email* input by the user and u picked by the server. Same goes for the device name, with the user chosen *dname* and server chosen *did*. (Part of) u and *did* play a role in the naming of files internally in the application. Additionally, u is used by the application if the user has chosen to require the password to be input on all startups.

3.1.5 Device Registration

At the point where the user registers a new account, the server already knows some information about the user. In particular, the list of keys kl, which includes the values ck and s_1. In order to register another device for the same user, the client does the following

Both Run the protocol **pandora/zk** from subsubsection 3.1.2 and continue if it succeeds.

Server Compute a list of the user's registered devices *dlist* and send *dlist* to the user.

Client User inputs a new device name *dname* and sends it to the server (the *dlist* is needed so the user can see a list of other owned devices)

Server Compute did_{new} as the maximum of device ID's from *dlist* plus 1. Send did_{new}, kl to the client.

Note how at account registration, the client sends the user's keys to the server. Conversely, during device registration, the server sends the user's keys to the client.

3.2 Keys

SpiderOak ONE uses several different cryptographic keys. Almost all are computed by the client at the point of account creation and remain static thereafter. As all keys are also stored on the server to allow support of multiple devices, these keys need to be encrypted. To that end, the application encrypts all but one key (the exception being *ck* from the previous section) in a hierarchical manner, with a key derived directly from the user's password at the top. A rough sketch of this procedure is as follows

- An RSA keypair `keypair.key` is encrypted with a key derived from the users password
- A symmetric key `symkey.key` is encrypted with `keypair.key`
- All other keys are encrypted with `symkey.key`.

The only key not encrypted in this manner is *ck* from the **pandora/zk** authentication protocol.

A more detailed description of the key encryption process follows. To that end, let p denote the user password, s_2 be a random 256-bit salt and *miv* is a 2048-bit random value used as a *master iv*. All keys are assumed to have a distinct name and size parameter. The latter is 256 in all but two cases, in which case it will be 4096.

keypair.key Let $(sk, pk = (e, n)) \leftarrow \text{RSAGen}(3072)$ be an RSA keypair with $e = 2^{16} + 1$.[16]. Compute a key and a *synthetic IV* as

$$\begin{aligned} k &\leftarrow \text{PBKDF2}(p, s_2, 16384) \\ iv &\leftarrow \text{sha256}(\text{"}keypair\text{"} \mid\mid s_2)_{0:16} \end{aligned} \tag{1}$$

and define `keypair.key` $:= \text{Enc}_k(iv, (sk, pk))$.

symkey.key Let k be a 3064-bit[17] random value s.t. the most significant byte is not 0 and let pk, sk be the values from `keypair.key`. Compute

$$\begin{aligned} c &\leftarrow \text{RSAEnc}_{pk}(k) \\ s &\leftarrow \text{Sign}_{sk}(\text{sha256}(c)) \end{aligned}$$

and define `symkey.key` $:= (c, s)$.[18]

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

[17] Note that this is 1 byte less than $|pk|$

[18] Clearly this KEM fails to satisfy CCA security (as opposed to more standard KEM based on RSA and OAEP) but, since CPA security seems enough in this application and k is chosen close to uniform in the domain this does not appear to be a problem.

Everything else. Let $k_{sym} \leftarrow \text{sha256}(\text{RSADec}_{sk}(c))$ and suppose we want to encrypt a key with name *name* and size parameter ℓ. To that end, let k be a random ℓ-bit string and compute a synthetic IV as

$$iv \leftarrow \text{sha256}(miv \mid\mid name)_{0:16}.$$

Finally, define $name := \text{Enc}_{k_{sym}}(iv, k)$.

When the user creates an account, all keys are transmitted to the server. In addition, the values s_2, miv and ck are sent unencrypted. Note that, in order for the client to recover their encryption keys, all that is needed is the user password and the values s_2 and miv. When the user registers a new device the server will then transmit all the encrypted keys and the values s_2 and miv, at which point the client can recover the actual keys.

An interesting observation is that the RSA keypair acts more as a "master key" than the user's password. Indeed, access to the content of `keypair.key` would give access to all other keys. This has some rather severe consequences with regard to password changes (as we will describe in subsection 3.6). The same can in principle be said about `symkey.key`.

For the rest of the paper we will call values encrypted by k_{sym} as *symkey encrypted* values. Also, the value miv is used throughout the encryption process (as a seed for IV generation), so whenever we write "miv" it is implied that the same value as above is used.

3.3 File Encryption

We describe two different schemes for file encryption used by SpiderOak ONE. The first is used for metadata regarding both user files and the application (settings etc.). The second is used only in the context of the files the user store. For the sake of terseness we assume all symkey encrypted values to be decrypted.

3.3.1 Filenames

The naming scheme of files used in application plays an important role in determining the keys used and how IVs are computed. The general format can be seen in Figure 2. `directory` determines the key. In some cases, `name` is used for

```
directory/name.extension
```

Figure 2: General filename format inside the application.

creating IVs, something we have already seen with e.g., `keypair.key`. Likewise for `directory`. The `extension` only plays a role with *journalfiles* (a specific kind of metadata file), and is not present in all files.

Of particular note is the fact that these filenames and directory names do *not* correspond to an actual location on the filesystem; they are only used by the application and the server. We shall label the actual filesystem directory and

filenames as *physical*. E.g., a user stores a file `foo.txt` at the *physical* location `foo/bar.txt` on the filesystem. However, internally in the application, the file might be stored at `block/1234-4-1001`.[19] We also remark that these names are not secret.

3.4 Metadata Files

The format for encrypted files holding metadata can be seen in (2), where rn is the *record number*, c is the encrypted content and $rs = |c|_8$ is the *record size*. We will refer to this construction as an *AppendFile*. In addition if f is an AppendFile, we write f_{rn} to denote its record number.

$$rn \,||\, rs \,||\, c \tag{2}$$

Suppose the application is to store some data d at a location described by the format in Figure 2. Compute the new record number as

$$rn = \max(\{f_{rn} \mid f \text{ stored AppendFile}\}) + 1 \tag{3}$$

Retrieve the appropriate symkey encrypted key k according to `directory` and compute a synthetic IV as

$$iv = \text{sha256}(miv \,||\, rn)_{0:16} \tag{4}$$

and the encryption

$$c = \text{Enc}_k(iv, d) \tag{5}$$

Finally, the new AppendFile g gets defined, using the values in (3) and (5), according to (2). That is

$$g = rn \,||\, |c| \,||\, c$$

A remark: if $|d|_8 > 32768$ then d will be split into chunks of at most 32768 bytes. Each chunk is then treated as a separate piece of data to be encrypted according to the process above. Decryption is straightforward: extract rn from the AppendFile, k according to `directory`, compute the IV as in (4) and decrypt the first rs bytes of c.

3.4.1 journalfiles

An important type of metadata files are *journalfiles*, which are used to keep track of all actions regarding other files: removal, moving, adding, deleting and so on. For our purposes, they are of interest because they add an additional level to the key hierarchy. Each (physical) directory that is backed up in SpiderOak ONE has its own journal and an associated key. We will refer to the latter as a *directory key*.

[19] The format *a*-*b*-*c*, is the naming scheme most commonly used inside the application. a is the n part of the username u from subsubsection 3.1.4, and is unique across all accounts; b corresponds to *did* from subsubsection 3.1.4 and c is a sequence number starting at 1001.

Suppose the user adds a physical directory to be stored in the cloud by SpiderOak. The application then creates a new journal with name *name*.`jrn` and a new directory key with name *name*.`key` in the following way

1. Compute a synthetic IV as

$$iv = \text{sha256}(miv \;||\; \text{“}journal\text{”} \;||\; name.\texttt{key})_{0:16}$$

2. Let dk, the *directory key*, be a 256-bit random string and define *name*.`key` as

$$name.\texttt{key} := \text{Enc}_{jk}(iv, dk)$$

where jk is a symkey encrypted key associated with journalfiles. The new journal *name*.`jrn` is then encrypted as an AppendFile using *name*.`key` as the key. This construction gives the application a key per directory, a key that is also used in the encryption of user files. Roughly speaking, when the user adds a file to some directory, the journal for that directory key is retrieved, then the directory is retrieved (by simply exchanging the `.jrn` extension for `.key`). The file is then encrypted using a construction that relies on the directory key. One important note is, the directory key used for a file, does *not* change when the file is moved. If the user moves a file, all the application does is create two new journal entries: one in the old directory, stating the file was moved out (a “move out” entry); and one in the new directory stating the file was moved in (a “move in” entry).

3.5 User Files

The client creates two different types of files, whenever the user uploads a new (physical) file. The first type will contain the content of the actual file, while the second contains information about the first. We will call the former *blockfiles* and the latter *versionfiles*.

The structure of both blockfiles and versionfiles are the same and can be seen in (6). eXk is an encrypted key derived from the data being encrypted and a 4096-bit *master secret key* mk (which is another symkey encrypted value), and c is the encrypted data.

$$eXk \;||\; c \tag{6}$$

How c is encrypted depends on whether the file is a blockfile or a versionfile. That said, the way eXk is derived from a piece of data m, with name *name* in a physical directory with key dk, is the same. This derivation can be seen in (8) (the key used to encrypt c being (7)).

$$Xk = \text{sha256}(m \;||\; mk) \tag{7}$$

$$eXk = \text{Enc}_{dk}(iv, Xk). \tag{8}$$

The synthetic IV used is computed according to (9).

$$iv = \text{sha256}(\texttt{directory} \;||\; name \;||\; miv)_{0:16} \tag{9}$$

where `directory` is either "block" or "version".

We now give a more complete description. That is, suppose the user stores some physical file f in a physical directory with the directory key dk. In addition, let Enc^* denote a full-width AES-CFB encryption (128-bit segment size), let $\text{pad}(x)$ denote a function that returns x padded according to ANSI X.923 and suppose $f.\texttt{name}$ is the name given to f in the application. Then

1. Partition f into n blocks $b_0, \ldots, b_n$ of some size. Each b_i has a distinct name $b_1.\texttt{name}$.

2. For each b_i, compute iv_i as in (9), using "block" as the `directory` and $b_i.\texttt{name}$ as *name*. Derive bk_i — the encryption key — according to (7) (using b_i in place of m). Compute

$$c_i = \text{Enc}^*_{bk_i}(iv, \text{pad}(b_i))$$
$$ebk_i = \text{Enc}_{dk}(iv, bk_i)$$

   and define the blockfile for block b_i according to (6) (i.e., as $ebk_i \mid\mid c_i$).

3. Derive, according to (7), a key fk using f; and compute, according to (9), an IV iv with `directory` "version" and *name* as $f.\texttt{name}$. Let $bl = [b_0.\texttt{name}, \ldots, b_n.\texttt{name}]$ be a list containing the names of the blocks containing f. Compute

$$c = \text{Enc}_{fk}(iv, bl)$$
$$efk = \text{Enc}_{dk}(iv, fk)$$

   and define the versionfile for f as $efk \mid\mid c$.

Some observations: Suppose f is small (e.g., a small text file), then it is likely that only one block is needed. Note that, in this case we will derive the same encryption key for both the (single) blockfile and versionfile, i.e., $fk = bk$. However, the IVs derived will be different since the value of `directory` differs (the string "block" is used in the first and "version" in the latter) and the IVs are generated using a collision-resistant hash function. A similar argument can be made in case f has two blocks b_i and b_j that are identical: in this case, $b_i.\texttt{name} \neq b_j.\texttt{name}$ and thus $iv_i \neq iv_j$. And similarly if f shares a block with some other file f'.

The way f is partitioned depends on what data f stores. For example, if f is an MP3 or JPEG file, then $|b_0|_8 = 2048$ and all subsequent blocks will be 102400 bytes. This arguably leaks some information about f, although we will not consider it further.

Finally, in reality, the list bl contains (in addition to the names of blocks) also an MD5 hash and adler32 checksum of each block, as well as its size.

Note that the construction makes sharing of a file efficient: If the user wants to share some file f, then it is enough that the client simply sends fk and all bk_i (for the relevant blocks) to the server. The server can then itself decrypt f. This is, more or less, also how file sharing is handled.

3.5.1 Single File Sharing

Let V and $B_0, \ldots, B_n$ be the versionfile respectively blockfiles corresponding to some file f the user wants to share. For V the client decrypts efk to obtain fk; for all B_i the client decrypts ebk_i to obtain bk_i. The client then constructs a Basic HTTP Authentication [10] header with a username consisting of an ID assigned by the server at account registration, a device ID and optionally rt from subsubsection 3.1.4. As password, the client will use the *encrypted* content of a symkey value.[20] The client sends $fk, bk_0, \ldots, bk_n$ to the server, using HTTP Basic Authentication and HTTPs. The server then responds with a URL at which the file f (which can now be reconstructed by the server, as it has all the necessary keys) can be downloaded. Note that, by using the fact that the encryption keys are part of a file, the application can share even large files very efficiently.

3.5.2 Shared Directories

The approach for sharing a whole directory is conceptually similar to the single file case. However, instead of sharing the keys embedded in the encrypted files (blockfiles and versionfiles), the application simply shares the *directory key*.

Since files are not reencrypted when they are moved, this approach inadvertently shares "too much" in some scenarios. We return to this issue in subsection 4.4.

3.6 Password Change and Key Upgrade

The hierarchical relationship between the various encryption keys in the application makes possible to support password change or RSA key upgrade at very little cost. SpiderOak ONE, not surprisingly, allows the user to change their password. In addition, the server can instruct the client to upgrade the strength of the RSA keypair inside `keypair.key`.

Password change. The application only contains two values that depend directly on the user's password: `keypair.key` as it is encrypted with a key k derived from the users password, and the *challenge key* ck. Thus, when the user changes their password, the application simply recomputes these two values. Note that the RSA keypair stored in `keypair.key` is *not* updated, and the system essentially just re-encrypts the same key using the new password. The details of the process were already described in (1).

RSA keypair Upgrade. A similar approach is taken when the RSA keypair from `keypair.key` has to be upgraded. However, since (the key inside) `symkey.key` depends on the size of this keypair, this will have to be recomputed as well. Roughly speaking

[20] This is the other 4096-bit symkey encrypted value (the first being mk). It is interesting that this is the only place it is used. In other words, this key is encrypted but never decrypted.

1. Compute new RSA keypair of size n_{new},
2. pick new k'_{sym} as $(n_{new} - 8)$ random bits,
3. encrypt k'_{sym} using the new RSA keypair, and
4. extract and re-encrypt all the *symkey encrypted* keys (i.e., decrypt them using the old k_{sym} and re-encrypt them using k'_{sym});

As in the password change case, this process does not sample fresh keys i.e., upgrading the RSA keypair does not produce new symkeys. The value of n_{new} (which the server sends) must be greater than the size of old keypair, in order to avoid trivial downgrade attack on the client by the client.

Consequences. Consider the case where a user wants to change their password (due to fear of the password being compromised) or where the server instructs a client to upgrade the keylength (due to fear on a possible attack on the current parameters). We note that, if the attack *has already taken place*, then neither the password change procedure, nor the RSA key upgrade procedure will guarantee any additional security *even for files which will be uploaded in the future!*

To see why, note that knowledge of the password (resp. the RSA secret key) allows to learn the symkey used to encrypt the files (concretely, the journalkey). Since these keys are not being rotated, the next uploaded file will be encrypted with a compromised key.

4 Attacks

We now present some concrete issues we found in the application, and show how a malicious server can abuse them in order to obtain a copy of the user password and thereby decrypt their stored files.

The attacks we will present can be abused by a malicious server acting as either an *active* or *passive* adversary. More precisely, we consider the following two types of malicious servers

Active A malicious server is considered *Active* if it will tamper with an ongoing connection between the server and client.

Passive A malicious server is considered *Passive* if it will only inspect that data being transmitted by the client.

We present three issues that can be exploited by an actively malicious server: A downgrade and memory leak in the protocol from subsubsection 3.1.1; a password retrieval in the protocol from subsubsection 3.1.2, where we in addition argue that the FINGERPRINT procedure is insufficient to guard against the attack; and lastly, an attack which abuses weak path string validation in some of the remote procedures available to the server. In particular, this allows us to request the file mentioned in subsection 2.2 that stores the user's password in

plaintext. We also present one passive attack, which was mentioned in passing when we described directory sharing: In a nutshell, sharing a directory has consequences for the secrecy of other files that was once in the shared directory, or that become part of the shared directory after it stops being shared.

Attacker Setup. For issues exploitable by an active adversary, we used a client running in a Debian 8.7 virtual machine. The SpiderOak ONE version was 6.1.5. We made no modifications to the client, although we did modify the environment it was running in. That is, by starting the client with the command in Figure 3

```
$ SPIDEROAKONE_SSL_VERIFY=0 SpiderOakONE
```

Figure 3: Command for running SpiderOak ONE without any kind of certificate validation enabled (GNU/Linux).

we can turn off certificate pinning, allowing us to perform active Man-in-the-Middle attacks.

The single passive attack was verified by using data generated by our analysis client. I.e., the client described in subsection 2.2 running on a Windows XP virtual machine.

4.1 Active Attack 1: bcrypt downgrade and memory leak

When executing the protocol in subsubsection 3.1.1, the client performs very little verification of the salt s it receives from the server. In fact, if one considers only the core application, no verification *at all* is done and the salt is passed directly to the underlying bcrypt library. This results in two distinct issues that can be combined by a malicious server to

1. obtain a weak password hash derived from s and user's password; and
2. leak a small amount of client memory.

The bcrypt KDF expects a salt of the *Modular Crypt Format*[21] i.e., of the form `$id$cost$salt`. In a nutshell, the first issue arises because a malicious server can lower the value of `cost`; the second issue is a result of a bug in the bcrypt implementation used, which leaks memory when `salt` is not valid base64.

Downgrade. During normal execution of the protocol the server will use a salt computed earlier by the client. In particular, `cost` will be 12 (as described in subsubsection 3.1.4). The "strength" of the downgrade the server can perform, depends on what values are allowed by the underlying bcrypt library. Inspecting

[21] See e.g., `http://man7.org/linux/man-pages/man3/crypt.3.html`

the implementation[22] reveals that `cost` must be between 4 (16 iterations) and 31 (2147483648 iterations). This implies that a downgrade by a factor of 8 (i.e., from 4096 to 16 iterations) is possible.

```
static void
decode_base64(u_int8_t *buffer, u_int16_t len,
              u_int8_t *data){
// snip
    while (bp < buffer + len) {
        c1 = CHAR64(*p);
        c2 = CHAR64(*(p + 1));
        /* Invalid data */
        if (c1 == 255 || c2 == 255)
            break;
// snip
int
pybc_bcrypt(const char *key, const char *salt,
            char *result, size_t result_len){
// snip
    u_int8_t csalt[BCRYPT_MAXSALT];
// snip
    decode_base64(csalt, BCRYPT_MAXSALT,
                  (u_int8_t *) salt);
```

Listing 1: Memory leak

Memory Leak. Further inspection of the bcrypt implementation revealed a memory disclosure bug. Consider the snippet in Listing 1, and suppose the server supplied a salt with invalid base64 data in the `salt` part. The code allocates 16 bytes of uninitialized memory in `csalt`, then tries to decode `salt`; storing the result in `csalt`. However, if the first byte of `salt` is *not* valid base64, the `decode_base64` function will exit immediately, leaving `csalt` uninitialized.

Constructing a "bad" salt for use by a malicious server is straightforward. For example, using the salt in Figure 4 results in the client returning password

`$2a$04$0x01AAAAAAAAAAAAAAAAAAAA`

Figure 4: Bad salt (`0x01` denotes the byte 00000001)

hashes shown in Figure 5.

In particular, the returned salts contain 16 bytes of memory (the 21 base64 encoded characters after the last `$`) as well as a cost factor of 4 indicating only 16 iterations was used in deriving any of the hashes. The impact of this attack (on top of the described memory leakage) mostly depends on the strength of

[22] concretely, the code in `bcrypt.c`, see `https://github.com/grnet/python-bcrypt/blob/master/bcrypt/bcrypt.c`

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

Figure 5: bcrypt hashes derived by the client when the user input the password `asd` and using the salt in Figure 4.

the user password. In any case, by downgrading the cost factor, the server can speed-up any brute force (or dictionary attack) against the hashed password.

Validation. We validated the attack by writing a small Python Flask[23] application that could act as a malicious login server.

4.2 Active Attack 2: escrow/challenge password retrieval

Recall the LayerEnc function from subsubsection 3.1.2 and note that the holder of the private keys sk_i, corresponding to the pk_i in l can decrypt *auth* and thus learn the user password. More precisely, suppose the server instead sends $l^* = [(id_0^*, pk_0^*), \ldots, (id_n^*, pk_n^*)]$, such that the server knows the secret keys sk_i^* corresponding to pk_i^* for $i = 1 \ldots n$ (and suppose for the time being that the client also accepts the fingerprint output by Fingerprint(l^*))[24]. When the client returns a response, the server can simply reverse the encryption done by LayerEnd as the server knows the corresponding secret keys to the public keys the client used.

Moreover, the if the server sends $l^* = [\,]$ (i.e., an empty list), the client will skip the encryption step altogether (since there is no values to loop over) and send back the user password un-encrypted.

```
eyJjaGFsbGVuZ2UiOiAiZGVhZGJlZWYiLCAicGFzc3dvcmQiOi
Aic2VjcmV0MTIzIn0=

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

Figure 6: Auth string returned by the client (before and after decoding) in the case where no keys are sent.

Since the client has to accept the fingerprint created from the list of keys the server sends, it is naturally to ask how effective it is. The fingerprint computed is shown as a popup containing a message and the fingerprint itself, along with two buttons: "yes" to accept the presented fingerprint and "no" to reject it. The message included is the interesting part and is quoted below (emphasis ours):

> If your SpiderOakONE Administrator has given you a fingerprint phrase and it matches the fingerprint below, **or if you have not**

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

[24] The fingerprint is shown even in the case of an empty list.

been given a fingerprint, please click "Yes" below. Otherwise click "No" and contact your SpiderOakONE Administrator.

Consider the following scenario. A user connects to a malicious server and tries to perform an action that requires some form of authentication towards the server (i.e., either account or device registration). The server, being malicious, then performs the attack described above, resulting in a fingerprint along with the aforementioned message being presented to the user. It is entirely possible the user has never seen this message before and is thus unsure of how to act. Now, common sense would dictate that you simply press "no" (do not accept strange proposals from your computer). However, the message suggests the opposite. In other words, if the user followed the instructions given by the application, she would end up revealing her password to the server, thus giving full access to all her files.

One could argue that it would be risky for a rogue SpiderOak to run this attack: regardless of whether the user clicks 'yes' or 'no', the user will detect some anomaly in the functioning of the application. However, as we shall see in the next section, there exist a way of stealing the user password which is completely undetectable.

Validation. The attack was verified using the same malicious login server as in the previous section (although with a different configuration).

4.3 Active Attack 3: Unsafe file retrieval

The next issue we describe centers on three remote procedures the server can invoke remotely on the client. Two of these procedures are available by default while the last is only available if the client has enabled the "remote diagnostics" flag. All three procedures share the same basic functionality, namely that of allowing the server to retrieve physical files stored locally on the client.

```
1 _safe_user_file_regexp = re.compile('''
2    ^([a-zA-Z0-9_-]{1,240})
3     ([\\\\/])
4     ((?:[@a-zA-Z0-9_-]|\\.(?!\\.)){1,240})$''',
5     re.VERBOSE)
```

Listing 2: safe user file regular expression. Formatted a bit for readability.

Of course, simply letting the server retrieve any physical file it wants to is insecure. The client therefore checks the path requested against the regular expression shown in Listing 2.
Roughly speaking, the regular expression allows only paths of the form `dir/file.ext`, where `dir` cannot contain any dots and `file` cannot more than one consecutive dots and no slashes.[25] If the path requested matches the regular

[25]The last requirement prevents relative directory traversing, e.g., `foo/bar/../../../outside.txt`

expression, the client appends it to the local directory SpiderOak ONE uses and retrieves the file. For example, if the server asks for `foo/bar.txt`, the client will reply with the content of the physical file stored at (if it exists)

```
$HOME/.config/SpiderOakONE/foo/bar.txt
```

By default the client will store a file with the user password in plaintext at two different locations:

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

The first directory does not match the regular expression (as the directory name contains a dot); the second does, however. Abusing this issue is then simple: When the client connects, simply issue a remote procedure call asking for the second location above. Since the client considers the path requested as "safe", the content of the file is returned. An example of the content can be seen in Figure 7.

```
{'reinstall_token':
   'xGqQnxG8+7wfvvE+EREZtcNSgSi2bsBP5z/RC24cHLs',
 'password_verify':
   '*\x84f;\xae\xd4\xb2\xcf\xb4w\tm\x91\x17\xecb',
 'user_name': 'u_spideroak_auto_205015',
 'password_plain': '________',
 'email': '__________________________',
 'device_id': 1}
```

Figure 7: Example content of a retrieved file after deserialization with password and email blurred. `reinstall_token` is the rt value from subsubsection 3.1.4. `password_verify` is a hash of the user's password

Since all file encryption ultimately depends on the security of the user password, the impact is clear: possessing the user password allows one to recover all their files.

Non-default behaviour. The setting described so far is the default behaviour in SpiderOak ONE in which the user password is stored unencrypted on the user client. Thanks to this, the user does not have to type the password at every login.

More paranoid users could have enabled the flag that requires the password to be typed at every login. When this flag is enabled, the `password` field in Figure 7 will be missing. However, `password_verify` will still be present, which is computed as

$$\begin{aligned} h_1 &= \text{MD5}(\text{"password\_verify"} \mid\mid \textit{username} \mid\mid \textit{password}) \\ h_2 &= \text{MD5}(h_1 \mid\mid \text{"password\_verify"} \mid\mid \textit{username} \mid\mid \textit{password}) \\ &= \texttt{password\_verify} \end{aligned}$$

where *username* is the value in the `user_name` field in Figure 7 and `password` is the user's password. Now the attack has the same flavour of the one described in subsection 4.1 i.e., the server has now hold of a *weak hash* of the password, which is much easier to brute-force.

Validation. We wrote a Man-in-the-Middle program that could parse and tamper with the remote procedure calls constructed by the PB interface. We then used it to intercept the connection between our client and a legitimate SpiderOak server. Finally, the actual attack (crafting the procedure call) was done by substituting a benign call from the legitimate server (e.g., a query about the current client time) with a malicious call, after which we could parse the clients response and verify that it was possible to extract the user's password.

4.4 Passive Attack 1: Missing key rotation for shared directories

The last issue we present is connected to the process of releasing the directory key when sharing a directory. More specifically, the client does not ensure that other files also encrypted under the shared directory key — but which are *not* currently part of the physical directory — are re-encrypted under a new directory key. We present two scenarios, in order to illustrate this issue

Scenario 1. The user has stored some physical directory D containing a file F on their machine. At some points the user decides to share D and therefore instructs their client to do so. The client then decrypts dk and sends it to SpiderOak, who can now decrypt F and publish it on their website. Later, the user instructs their client to stop sharing D. The client forwards the instruction to SpiderOak who removes F from their website. Now, later again, the client adds a new file F' to D. As the directory key associated with D is still dk, F' will be encrypted using dk (as explained in subsection 3.3). In particular, when the client uploads the encryption of F', the server can also decrypt this file, as it already possesses dk.

Scenario 2. The user has stored some physical directory D containing files F_{pub} and F_{sec}. The user wants share D, but not F_{sec}. Thus, before the user asks their client to share D, the user moves F_{sec} into some other directory D'. The user then asks their client to share D (now containing only F_{pub}) at which point the client does as before: Find the correct dk, decrypt it and send it to SpiderOak. However, the act of moving a physical file simply means adding the "move in" and "move out" journal entries. In particular, the encryption key used will still be dk. Thus, when the client uploads dk, the server can recover both F_{pub} and F_{sec}.

It is easy to imagine both scenarios happening in the real world. The first scenario shows that sharing a directory "taints" files into the future, while the

second scenario shows that sharing also taints files into the past (so to speak).

Validation. We used our analysis client and executed the two scenarios above. In order to verify that we could decrypt some block b, we need miv, b.`name` and dk (as described in subsection 3.5). The first value could be extracted by observing a device registration (miv is part of kl in subsubsection 3.1.5); b.`name` is sent when (the blockfile for) b gets uploaded; and the directory key dk is transmitted un-encrypted when the corresponding directory gets shared.

5 Conclusion

In this paper we described a number of vulnerabilities which allow a rogue SpiderOak server (or anyone able to bypass certificate pinning and manage to interact with the client software) to break the *no-knowledge* property i.e., compromise the confidentiality of the user's file. While most of the problems have already been fixed by SpiderOak, the fact that the attacks have been possible so far has serious consequences: since the attacks are easy to carry and undetectable at the client side, there is no way to be completely sure that attacks have not been already run.

We would recommend all SpiderOak users to change their password (as this could have been stolen). Unfortunately, as described in subsection 3.6, changing the password simply re-encrypts the long term secret key under a new password. Therefore if an attacker has already obtained this long-term secret key, changing the password will not even help in ensuring the confidentiality of the files uploaded by the users in the future (and clearly nothing can restore the eventual loss of confidentiality which could have already occurred).

Our analysis can also be used to draw some general conclusions about the design of encrypted cloud storage systems. We believe that "the root of all evil" in the case of SpiderOak relies in the choice of using the same secret (the password) both for authentication and confidentiality purposes. We understand that, from a user experience point of view, it is hard to have to generate, store and type two strong passwords. However we have also observed how this choice (combined with authentication protocols which are not *zero-knowledge* in a strong, cryptographic sense) leads to a complete loss of confidentiality.

Acknowledgments. Research funded by the Danish Council of Independent Research. We are grateful to DDIS for useful sparring and technical dialogues.

References

[1] Karthikeyan Bhargavan and Antoine Delignat-Lavaud. Web-based attacks on host-proof encrypted storage. In *6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings*, pages 97–104, 2012.

[2] Moritz Borgmann, Tobias Hahn, Michael Herfert, Thomas Kunz, Marcel Richter, Ursula Viebeg, and Sven Vowe. On the security of cloud storage services. 2012.

[3] Tom Chothia, Flavio D Garcia, Chris Heppel, and Chris McMahon Stone. Why banker bob (still) can't get tls right: A security analysis of tls in leading uk banking apps. 2017.

[4] McDonald D. A convention for human-readable 128-bit keys. RFC 1751, RFC Editor, 12 1994.

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

[6] EFF. Who has your back? government data requests 2014. `https://www.eff.org/who-has-your-back-2014#spideroak`, 2014.

[7] M. Grothe, C. Mainka, P. Rösler, J. Jupke, J. Kaiser, and J. Schwenk. Your cloud in my company: Modern rights management services revisited. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*, pages 217–222, Aug 2016.

[8] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 491–500, 2011.

[9] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[10] Reschke J. The 'basic' http authentication scheme. RFC 7617, RFC Editor, 9 2015.

[11] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

[12] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. Dupless: Server-aided encryption for deduplicated storage. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 179–194, 2013.

[13] 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., 2013. USENIX.

[14] Niels Provos and David Mazières. A future-adaptable password scheme. In *USENIX Annual Technical conference*, 1999.

[15] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. Cryptology ePrint Archive, Report 2017/190, 2017. `http://eprint.iacr.org/2017/190`.

[16] Nikos Virvilis, Stelios Dritsas, and Dimitris Gritzalis. *Secure Cloud Storage: Available Infrastructures and Architectures Review and Evaluation*, pages 74–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

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