Can You Trust Your Encrypted Cloud? An Assessment of SpiderOakONE's Security

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

January 2, 2018

Abstract

This paper presents an independent security review of a popular encrypted cloud storage service (ECS) *SpiderOakONE*. Contrary to previous work analyzing similar programs, we formally define a minimal security requirements for confidentiality in ECS which takes into account the possibility that the ECS actively turns against its users in an attempt to beak the confidentiality of the users' data.

Our analysis uncovered several serious issues, which either directly or indirectly damage the confidentiality of a user's files, therefore breaking the claimed *Zero-* or *No-Knowledge* property (e.g., the claim that even the ECS itself cannot access the users' data). After responsibly disclosing the issues we found to SpiderOak, most have been fixed.

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 a network eavesdropper, no mechanism prevents the storage provider itself from accessing the users' data. (In fact, the economic viability 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 deduplication techniques. However, using de-duplication might allow attackers to learn information about

[1] `https://blogs.dropbox.com/dropbox/2016/03/500-million/` (all links last retrieved on December 20th 2017)

other users' data. See [19] for a description of the problem and [18, 25] for some cryptographic solutions which allow to perform de-duplication in a secure way.)

Moreover, data that is being stored unencrypted is more vulnerable to data breaches, whether that be from a malicious outsider, malicious insider or government sponsored actor. (According to [15], of the 1792 breaches in 2016 only 75, or 4.2%, used encryption in part or full.)

All of these factors lead to an increased interest in password-encrypted cloud storage services that 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 service provider 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, 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 [27], and has received positive reviews by the EFF [14]. SpiderOak in particular marketed their product using the term *Zero-Knowledge* (now replaced by *No-Knowledge* [31]), 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 this *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 (e.g., [7]), so we will not address this threat further in this paper. What is perhaps more interesting, from a technical point of view, is to look at the service provider's choices in protocol design and software implementation to ensure that no one, not 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 to login using the web interface (which would reveal the password to the server) are required to acknowledge the following message[3]:

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

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

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

1.1 Contributions and Paper Organization

The main contribution of this paper is to present an independent security review of the *SpiderOakONE* client software with the goal of examining the *Zero-* or *No-Knowledge* claim *from the perspective of a malicious service provider*. We believe that independent security reviews of real world systems is very important towards the goal of improving user's privacy.

In Section 2 we provide a simple and formal definition of what kind of basic *confidentiality* should be provided by a password-encrypted cloud storage service. We believe that our definition represents the *minimal* guarantees that such a service should provide, and we argue for why this is the case. To the best of our knowledge, this is the first such definition, since neither SpiderOak, nor other password-encrypted cloud storage services have ever provided formal definitions of which security guarantees they provide (in particular, only intuitive, high-level and therefore ambiguous descriptions of what *Zero-* or *No-Knowledge* means could be found): therefore, we believe that our definition can be used as a benchmark in future analysis of other password-encrypted services and will have a tangible and concrete effect in practice on the design of such services. As an example of this, we note that SpiderOak published a piece on this more comprehensive threat model [34], partially inspired by our work.

In Section 3 we describe how SpiderOakONE is implemented, what systems it runs on and what version we analysed. We then describe how the application was reverse engineered and analysed. This description comprises both the implementation and how the client application was reverse engineered, as well as the choice of protocols and primitives. We show that decompilation is fairly easy and that therefore future audits can be preformed without access to the source code.

Since SpiderOak provides only a very superficial description of its application (cf. their whitepaper [33]), our description fills this hole and could serve as a stepping stone for in-depth analysis of the application's mode of operation in the future.

Our main results can be found in Section 4, where a series of attacks that can be carried out against SpiderOak users in our threat model are presented. We stress that these attacks are not possible just because of implementation "bugs", but because of the design choices made in the development of the system. Therefore, these attacks teach us what threats and pitfalls should be avoided and should serve as motivation for careful vetting of applications that make strong claims with regard to security—not just for their users but also for their developers.

We hope that the attacks we present will motivate more research into the security encrypted cloud storage application, when the ECS is considered as potentially malicious. In particular, just like SpiderOak claims[4] that only the

[4] `https://support.spideroak.com/hc/en-us/articles/115001855103-No-Knowledge-Explained`

user can read their files, so does e.g., Tresorit[5] and LastPass[6].

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 [35]. We find it commendable that SpiderOak has reacted so swiftly to the issues we found.

1.2 Related Work

Previous analysis' of SpiderOakONE has focused on its shared directory functionality (i.e., *ShareRooms*) and only in the presence of an external attacker. In [6] Bharagavan and Delignat-Lavaud demonstrate a Cross Site Request Forgery attack on the ShareRoom functionality due to a lax cross-origin policy. Bansel et al. later provide a formal modelling of this attack using ProVerif in [2]. Wilson and Ateniese show in [39] that the ShareRoom functionality reveals the shared files to the server, in addition to the intended recipient. Our work shows that, in addition to the shared files being readable by the server, some files that are *not* shared (i.e., not part of the directory being shared) might also be revealed.

Several attacks against Cloud Storage Services that use deduplication was described in [19] and [29]. However, SpiderOak does not do deduplication [33] (see also Section 3.4).

Password managers can be seen as a special case of encrypted cloud storage applications. In this direction, Li et al. conducted in [28] a security analysis of five popular web-based password managers and found in four out of five cases that attackers could learn arbitrary credentials. In [3] Belenko and Sklyarov show that mobile password managers often provide little or no protection.

In the context of ECS in general: Virvisil et al. [38] provide a brief survey of challenges and solutions for secure cloud storage. An extensive survey of some cloud storage application can be found in [8]. The description we provide of SpiderOakONE (Section 3 and the Appendix) can be seen as complementary to this previous work.

Kamara et el. provide in [23] a description of how to build ECS using standard cryptographic tools. Like us, they assume that the client is trusted, while the server is not. A similar assumption exists in [37] (so called "internal adversaries" in their work).

2 Security Model

In this section we provide a formal definition capturing a basic *confidentiality* security requirement for files stored on a *password-encrypted cloud storage*

[5] `https://tresorit.com/security`
[6] `https://lastpass.com/whylastpass_technology.php`

service (or PECS for short). The definition is designed to capture the *minimal* security guarantees that a PECS should provide, and we believe it can be used as a simple benchmark against which other PECS can be tested. The definition does not attempt to capture every possible security requirements for a PECS: in particular, the definition does not capture hiding the access pattern (which could be achieved using tools such as Oblivious RAM [17, 36]), nor does it capture *retriviability* (which could be addressed using *proofs of retrievability* [21, 36]), *integrity*, or several other properties which could be desirable in an PECS context.

The definition will follow the standard game-based approach typically used in cryptography, and will resemble the common notion of CCA security, and the way we deal with password encrypted data is inspired by [5, 40].

In our abstraction we model a PECS as a pair of interactive systems Cli, Ser which can run a number of different subprotocols (described below). We use the standard notation $(o_C; o_S) \leftarrow \pi(i_C; i_S)$ to denote an interactive protocol between Cli and Ser, where (i_C, o_C) represent the input and output of Cli, and (i_S, o_S) represent the input and output of Ser. We denote by $\perp$ an empty input/output.

Server Init: The function $\mathsf{st}_0 \leftarrow \mathsf{Init}()$ is used to initialize the server state. All other subprotocols will take as input the current state of the server st_i and output an updated state st_{i+1} (as a notational convenience, we assume that the state contains the complete view of the server in each subprotocol and that $\mathsf{st}_{i+1} \supseteq \mathsf{st}_i$ i.e., no information is ever erased from the server's state);

Password Registration: Running $(pw; \mathsf{st}_{i+1}) \leftarrow \pi_{\mathsf{reg}}(\kappa; \mathsf{st}_i)$ the client can register a password of strength κ on the server; After this step the server's state contains a hashed version of the password $h = H^c(pw)$, where c specifies the number of iterations of the hash function/random oracle H. The password registration command can be issued multiple times, to capture the fact that a user might want to change password during the lifetime of the system.

File Storage: Running $(\perp; \mathsf{st}_{i+1}) \leftarrow \pi_{\mathsf{sto}}(pw, f, id; \mathsf{st}_i)$ a client can store a file f with identifier id using the *current* password pw on the server;

Other Commands: Depending on the specific PECS system, a number of other *commands* will be implemented using sub-protocols $(o_C; o_S) \leftarrow \pi_{\mathsf{cmd}}(pw, id_1, \ldots, id_\ell; st_i)$ which take as input the current client's password and zero or more file id's. As a result the server state might be updated. We divide the commands in *non-revealing commands* π_{nrc}, capturing the fact that these commands should not reveal any information about the content of the files to the server, and in *revealing commands* π_{rev}, capturing the fact that when using these commands the server is allowed to learn the content of the involved files. Examples of common non-revealing commands include client authentication (login), retrieving a file at the client

side, and moving or deleting a file at the server side; examples of common revealing commands include public sharing or declassifying of a file.

We are now ready to present a security experiment, modeled as a game between the client and the adversary, capturing the desired security requirements. For simplicity we present the game with a single Cli, but the definition could be easily extended to deal with the more realistic setting in which multiple clients interact with the same encrypted cloud storage PECS.

***Confidentiality Experiment* for** PECS

1. $\mathsf{st}_0 \leftarrow \mathsf{Init}()$;
2. $(pw; \mathsf{st}_1) \leftarrow \pi_{\mathsf{reg}}(\kappa; \mathsf{st}_0)$;
3. The adversary $\mathcal{A}$ on input the server state st_1 adaptively chooses a series of commands to be executed (password registration, store, other revealing or non-revealing commands), specifying the input of the client in every subprotocol. More precisely, the adversary can specify a command cmd and the corresponding inputs and $(o_C; o_S) \leftarrow \pi_{\mathsf{cmd}}(pw, i_C; i_S)$ will be executed with the current password. [Optional for *forward secrecy*: every time a new password is registered, the old password leaks to $\mathcal{A}$];
4. At some point $\mathcal{A}$ outputs (f_0, f_1, id^*) where f_0, f_1 are two files of equal length;
5. One of the two files (chosen at random) is now stored on the server i.e., $b \leftarrow \{0,1\}$, and $(\perp; \mathsf{st}_{i+1}) \leftarrow \pi_{\mathsf{sto}}(pw, f_b, id^*; \mathsf{st}_i)$;
6. $\mathcal{A}$ continues issuing commands to be executed as in step (3), but is now prevented from running any *revealing command* on id^* [Optional for *forward secrecy*: when new passwords are registered, the old password *does not* leak to $\mathcal{A}$];
7. $\mathcal{A}$ outputs a guess b'.

Definition 1 (Confidentiality of Password-Encrypted Cloud Storage)**.** *We say that a* PECS *satisfies* file confidentiality *if no PPT adversary* $\mathcal{A}$ *making at most* q *queries to the random oracle can win in the above* confidentiality experiment *with probability more than*

$$\frac{1}{2} + \frac{q}{c2^{\kappa}}$$

(plus a negligible factor in the computational security parameter k*). In addition we say a* PECS *satisfies* forward secrecy *if the above holds even when old passwords are leaked to* $\mathcal{A}$ *in step 3 of the experiment.*

Some remarks about our definition are in order at this point: as already stated, our definition only attempts at capturing a minimal confidentiality requirement for the stored data, and does not capture many other desirable security properties (integrity, retrievability, hiding the access pattern, etc.). Since

our definition also allows for *revealing commands*, our definition capture the intuitive requirement that declassifying (e.g., sharing publicly) one or more files should not impact the security of the other files stored at the server. We distinguish between a *passive* $\mathcal{A}$ that only chooses the scheduling and the inputs of the commands to be executed, and an *active* $\mathcal{A}$ that in addition can specify arbitrarily corrupted behaviour for the server (but not for the client) in the subprotocols. We use the RO model to be able to better quantify the probability that the adversary can succeed in *brute forcing* the hash of the password (which is unavoidable in any password-encrypted protocol), and to allow to reason about attacks that might give a "constant" (e.g., 10^3) advantage in succeeding, which might be considered a breach in practice. In particular our definition allows for a "slack term" $\frac{q}{c2^\kappa}$ in the winning probability for the adversary that represent the base probability for an adversary to brute force a password hash $h = H^c(pw)$ with q queries to H, where the password pw has strength κ and c denotes the number of iterations of the hash function. The *forward secure* version of our definition captures the natural requirement that, in the case an old password is leaked to the server, the confidentiality of files which were uploaded under a newer password should not be compromised.

As is common in cryptographic definitions, we require that the adversary should not be able to learn *any information* about stored files, and we capture this by saying that no adversary should be able to even *distinguish* between the encryptions of two known files. However, in the upcoming sections we will distinguish between different levels to which the definition can fail. In particular we will consider:

Password recovery: This is considered a *total break* of the system, as the adversary will be able to recover every single file ever stored by the client; (As in Attacks 4.2 and 4.3)

File recovery: Here the adversary can completely recover one or more files; (As in Attack 4.4)

ρ-Password Weakening: Here the adversary can increase the *password guessing advantage*. We say that an adversary has succesfully run a ρ-password weakening attack if, after the attack, the probability that the adversary wins the distinguishing game is at least

$$\frac{1}{2} + \frac{q}{c2^{\kappa-\rho}}$$

i.e., the attack effectively removes ρ bits of security from the password. Attacks 4.1 and 4.3 are examples of this kind of attacks.

Note that a password recovery attack is (clearly) the most devastating one, whereas file recovery attacks and password weakening attacks are somehow incomparable: depending on the password strength κ and the factor ρ, a password weakening attack might have no practical impact (and not be enough to recover even a single file) or lead to a complete password recovery (thus allowing to recover every encrypted file).

Our Model vs. the Real World. One objection to our definition might be that an adversary (the server) cannot force the client to run adversarially chosen subprotocols, however, as detailed later in Section 3.1 this is justified by the fact that the client software offers an RPC interface.[7]

Another objection might be that since SpiderOak is proprietary, the adversary could simply serve a "broken" copy of the client to the user. While not unheard of (e.g., the Juniper incident [9]), such an attack could be easily detectable and could potentially ruin the reputation of a service provider. Moreover, this attack could be prevented by making the client version of the software *open source*, so that clients can verify that the software they run satisfies the specifications of the protocol. Therefore, we will not consider this threat in the paper and assume that the client is able to retrieve an "honest" copy of the client software, and that only in a second phase the cloud service provider turns on its users. (This models the natural scenario in which the service provider is coerced to attack one of its users, or the case in which a company is sold and therefore a potentially malicious actor gets full control of the server.)

Finally we note that using an encrypted communication channel (e.g., TLS) between the client and the server does not help towards achieving our security definition. Indeed, the adversary is the *intended* recipient of the client's (password-encrypted) data. On the other hand, using an authenticated and encrypted channel helps in case where a system is (as we shall see) not secure according to our definition, since in this case the use of TLS is *necessary* to prevent that third parties can impersonate the server and run the attacks against the client.

To conclude, and in the context of our analysis, our definition captures attacks that (in the real world) can be carried out by:

1. A rogue SpiderOak server (e.g., malicious insider or a new owner);
2. A rogue SpiderOak enterprise server (running a local SpiderOak server); or
3. Anyone else able to impersonate the server towards a client (e.g., by bypassing certificate pinning, exploiting possible vulnerabilities in the TLS implementation, etc.)

Feasibility of the Definition Before describing how SpiderOak *does not* satisfy our definition of security for PECS, we briefly sketch how one could build a system which which would satisfy Definition 1 (this is only meant as a "sanity check" to justify that our definition of security is indeed achievable using standard cryptographic techniques): At the key registration step, the user should send the hashed version of the password to the server. For user authentication a *Zero-Knowledge* identification protocol (in the *cryptographic*

[7] It is not unreasonable to assume that this applies to some degree in general. E.g., for applications that support multiple devices (an arguably necessary criteria for a Cloud Storage application), the server needs to be able to signal device A that a file was uploaded on device B, in order to preserve consistency.

sense e.g., such as [16]) could be used. All files should be encrypted by the client before being uploaded to the server using provably secure encryption schemes e.g., satisfying the notion of *authenticated encryption* [24]. To satisfy *forward secrecy* new files need to be encrypted using the new password. Note that our definition does not require anything of old files, therefore old files need not be downloaded and re-encrypted by the clients—a simple and efficient solution would involve storing the old password on the server encrypted under the new password.

3 SpiderOakONE

In this section we provide some high-level background information about the SpiderOakONE client application in order to allow the reader to understand the attacks presented later. We describe the offered functionalities, how the client was reverse-engineered, and the necessary technical details of its inner workings (authentication, encryption and keys, which will be treated separately). More technical details about the functioning of the client application can be found in the Appendix.

3.1 Offered Functionality

The functionality offered by SpiderOakONE is what one would expect from a PECS: Automatic synchronization of one or more directories, recovery of older version files and selective file sharing (either single files or whole directories). SpiderOakONE runs on all major operating systems (MAC OSX, Linux and Windows). Our analysis focused on the Windows and Linux versions (version 6.1.5 released July 26th, 2016) and we note that there is no discernible difference between the clients running on different OS'. Although it is possible to use the SpiderOak website to log into an account, it is not possible to register an account through their website. It is also not possible to upload files through their web interface (as opposed to e.g., Dropbox or Google Drive). The only functionality offered on their website is viewing of files that have been shared. (The reason for this, is that shared files are not end-to-end encrypted, but are at most protected by HTTP Basic Authentication. That is, the server has access to all shared files in plaintext.)

Mobile client applications for both Android and iOS are also available, although these cannot presently do any encryption or decryption of files. Their functionality is, to the best of our knowledge, limited to what their web interface offers, i.e., viewing of shared files only.

3.2 Methodology and Reverse Engineering

The SpiderOakONE client is written in Python 2.7 and comes with a collection of bundled libraries, such as OpenSSL. SpiderOakONE does not employ any kind of obfuscation, unlike e.g., Dropbox [26], making reverse engineering quite

easy using standard off-the-shelf tools. The Python bytecode files for the client is bundled together with the installer, and can be found in a zip archive named either `library.zip` or `shared.zip`, depending on the operating system (the former being the file on Linux and the latter Windows). In the end, decompilation of the core client application could be achieved by a small bash script that uses uncompyle6[8].

All communication happens over TLS and the client uses certificate pinning by default. I.e., the client checks that the incoming server certificate validates against a small set of hard-coded certificates. We also note that certificate checking is implemented in a sound way, and that it avoids pitfalls such as forgetting to check the CN field [10]. Communication (below the TLS layer) happens with either HTTP or a two-way Perspective Broker RPC interface offered by Twisted[9] (two-way since the client can call methods on the server and vice versa). HTTP is used during authentication and the RPC interface is used for essentially everything else.

Reading TLS traffic. We wrote a small patch for the client that made it output the TLS master secret of any connection it establishes to a file. Being able to read the data that is sent between the client and server was important to understand what data is being disclosed to the server and to be able to make educated guesses at the server behaviour (to which we did not have access). We found that all connections were handled by Twisted and we therefore only had to patch the code at a single location. Installation of the patch was in addition made easier by the aforementioned fact that no obfuscation is used (we could simply alter the decompiled code and put it back into the compressed archive in the installation folder). During our inspection of the application, we also discovered that certificate verification could be turned off by running the application with an environment variable `SPIDEROAKONE_SSL_VERIFY` set to `0`. This fact turned out to be very useful for validating some of our attacks later on. (Specifically, it allowed us to conduct Man-in-the-Middle attacks or run the client against a "rogue" server without performing any modifications to the clients code.)

Analysis by printing. We employed what is essentially a "debugging by printing" technique in order to trace the execution of the client. By utilizing an already existing logging framework in the application, as well as the fact that modifications are easy to make, we could e.g., make the client output the encryption keys created during the execution of the protocols. Then, given these keys, we could implement our own decryption routines, which could be tested for correctness by decrypting the data sent by the client to the server.

[8] `https://github.com/rocky/python-uncompyle6`
[9] `https://twistedmatrix.com`

3.3 Authentication

SpiderOakONE is able to execute different authentication protocols depending on the context and what the server sends. I.e., if the user is creating a new account, one kind of authentication protocol will be used, and if the user is logging in with an existing account, another type of protocol will be used. The (human) user will only be involved in the authentication process if (1) they are creating a new account, (2) logging into an existing account on a new device, or (3) if they have chosen to require a password every time the client application starts (which is non-default behaviour). This section focuses on protocols that are applicable to only points 1 and 2. Point 3 is treated briefly in the description of Attack 4.3.

The server has full control over which protocol is actually run. Concretely, the server will send a short identifier that the client then uses to determine which protocol it should engage in (that is, there is no protocol negotiation between client and server similar to what happens in e.g., TLS, SSH, etc.). So even if some of the protocols we describe were not observed during *normal* interaction between the client and server, a *malicious* server can nevertheless still make a client engage in the protocol.

We restrict ourselves to describing only two of the four possible protocols in this section—the protocols presented here are the ones that we will attack later, and as such, the presentations focus on their flaws. A full technical description of all four protocols can be found in Appendix B. Also worth mentioning is that all the authentication protocols used can be categorized as non-standard or "home-made" (even if sound, provably secure authentication protocols which allow a client to authenticate itself without revealing *any information* about the password exist in the literature).

Authentication using bcrypt. The first protocol we describe is fairly simple and involves the client deriving a bcrypt [30] hash from the user's password and a salt supplied by the server. The server then compares this hash with a hash sent previously by the user. It goes as follows:

Client: Send username usr to the server.

Server: Do a lookup for the bcrypt salt s' associated with usr and if found, return it to the user. Otherwise abort.

Client: Compute $h = \mathsf{bcrypt}(pw, s)$ and send h to the server.

Server: Lookup h' associated with usr. If $h' = h$ consider the user authenticated, otherwise abort.

During account registration, the client will generate a random bcrypt salt s (with a work factor of 12), compute $h = \mathsf{bcrypt}(pw, s)$ and send (h, s) to the server along with usr. If usr is not already registered, the server will then associate (h, s) with usr. *Immediately afterwards*, the client and server will then run the above protocol. This execution flow essentially encompasses the account registration phase.

A non-default authentication protocol (escrow login). The other authentication protocol we will look at is *non-default* in the sense that we did not observe it being used during normal interaction between the server and the client. For reference, we will call this protocol escrow login. (The format of the protocol implies that it is used for escrowing keys, and we believe it is used in SpiderOak's enterprise product "Groups".) Nevertheless, since the server gets to pick which authentication protocol to run, and because this protocol can be leveraged for a *password recovery* attack, we will present it here. The technical details of the protocol can be found in Appendix B.3. The protocol consists of two steps:

1. Computing a *fingerprint* on a list lst of public keys obtained from the server and having the user verify said fingerprint; and
2. Computing a *layered encryption* of the user's password pw using the public keys from lst and returning this encryption to the server.

The rough idea of step 1, is to hash all keys in lst and create a fingerprint using RFC1751 [11]. For step 2, first pw is encrypted using pk_1 (i.e., the first key in lst); the result is then encrypted with pk_2 and so on. The exact details can be found in the appendix. The protocol in its entirety proceeds roughly as follows:

Client: Send username usr to the server.

Server: Do a lookup for a list lst containing some number of RSA public keys pk_i. Also do a lookup for a string chl. Send (lst, chl) to the client.

User: Compute a fingerprint of lst and show fp to the user. If the user accepts fp, proceed. Otherwise abort.

Client: If still running. Compute $c = E_{pk_n}(\ldots E_{pk_1}(pw \mid\mid chl))$ using the keys pk_i from lst. Send c back to the server.

Our description stops here since, as mentioned, we did not observe the protocol during normal interaction between the client and server and thus cannot say what is supposed to happen on the server side after the it receives c. However, the description above will be sufficient for the purpose of demonstrating the attack against the client.

3.4 Keys and Secrets

SpiderOakONE creates and maintains several different secrets used for IV generation[10], key generation, encryption, authentication and as KDF salts. Some of these values will be public, in the sense that both the client and server know them; the rest will be private, i.e., the server knows only an encryption of those values. From a high-level point of view, it is possible to divide these secrets into three groups, depending on when they are created:

[10] SpiderOakONE uses a SIV scheme [32]. Concretely, IVs are generated as $H(id \mid\mid miv)$ where id is e.g., the unique id for the content and miv is a per-account random string (long-term secret).

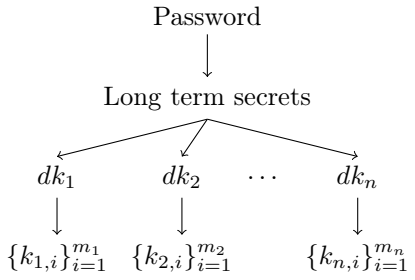

Figure 1: Relationship between secrets (the notation $a \rightarrow b$ indicates that the value b can only be recovered using the value a).

Long-term: Secrets that are created at the same time as the account are called *long-term*. In a nutshell, each long-term secret is protected by first deriving a key from the user's password and then using this key with a symmetric encryption scheme to encrypt the secret. (Details can be found in Appendix C.) We note that all non-key material (seeds, KDF salts, and so on) are in this category.

Directory keys: Each directory stored on the client will have an associated key, which we dub *directory key*. Each directory key is encrypted with a single specific long-term key.

File encryption keys: Each file stored in a given directory is encrypted with its own fresh key. Interestingly, this key is derived as $k = H(F \mid\mid mk)$ where F is the content being encrypted, mk is a long-term secret ($mk =$*"master key"*) and H a hash function. The encrypted file is stored as $E_{dk}(k) \mid\mid E_k(F)$ where dk is the directory key and E is a symmetric cipher. This scheme is reminiscent of a convergent encryption scheme as described in [4, 12], although the introduction of mk means that deduplication is not possible.

The relationship between long-term secrets, directory keys and file encryption keys is illustrated in Figure 1.

Password Changes. The relationship between the different secrets used in the application (i.e., the user's password protects the long-term secrets, which protect directory keys, which protect file encryption keys) means care has to be taken when a password change occurs. In our analysis we discovered that when a user starts a password change in SpiderOakONE the only effect is to re-encrypt the long-term secrets that are directly affected by the password change (i.e., secrets which were encrypted using $k = KDF(pw)$). In particular, the long-term secrets themselves are not rotated, nor are the directory and file keys.

3.5 Encryption

File encryption is handled in two slightly different ways, depending on the data being encrypted. One method is used for encrypting metadata (such as directory structure, settings and even old passwords); another method is used for encrypting a user's files. That said, these differences are small and we only summarize how each method works here. (Details can be found in Appendix D.)

Metadata encryption. SpiderOakONE uses a different long-term key for each particular type of metadata. For example, all information concerning directory structure is encrypted with a long-term key tree.key; all information concerning application settings is encrypted with a long-term key conf.key, and so on. Metadata encryption is then conceptually simple: For a piece of metadata m, find the corresponding long-term key k and encrypt m using k.

User file encryption. From a high-level point of view, encryption of a file F in SpiderOakONE is performed as follows:

1. Split F into blocks b_i, and assign identifiers to each block;
2. Derive a key for each block as $k_i = H(b_i \mid\mid mk)$ and encrypt b_i under k_i as $E_{dk}(k_i) \mid\mid E_{k_i}(b_i)$ (where dk is the directory key for the directory F was added to, and mk is the master secret key);
3. Derive a key using the whole file $vk = H(F \mid\mid mk)$ and use it to encrypt the identifiers of all the blocks that form F;
4. Upload the encryption of each block and the encrypted list of identifiers to the server.

The encryption done in step 3 corresponds to a so-called *version file* for F. Each time a file is added or updated, a version file is also generated. As the name implies, these files are used for versioning. However, they are also used in file sharing and file recovery, in that they describe which encrypted blocks of data makes up a particular file.

File sharing. File sharing—of either an individual file or whole directories—is performed in the following way:

Single File: Use the version information of the file being shared to determine which blocks are needed. From these blocks, each key is recovered and uploaded to the server (together with information about which blocks the server needs to decrypt). A shared file lives for three days at which point the server removes access to it.

Directory (*ShareRooms*): Decrypt the particular directory key used for the directory that is being shared, and give it to the server. The server can then itself recover all files. A shared directory lives until the user explicitly revokes it.

Note that this affirms the results of [39], i.e., that the server can read files that are being shared.

4 Attacks on SpiderOakONE

The following section presents four different attacks that show how the analysed version of SpiderOakONE does not satisfy Definition 1. All attacks are critical and lead to either a total breach of confidentiality (i.e., *password recovery*), *file recovery* or *password weakening*. For each attack, we describe the underlying cause, how the attack was experimentally validated and the practical impact.

4.1 Password weakening in bcrypt login

Recall the bcrypt authentication protocol: the Server sends a salt s, the client computes $h = \mathsf{bcrypt}(pw, s)$ and returns h to the server, who then accepts or rejects.

The issue that will be exploited in this protocol, is (1) that the format of the salt s used by the bcrypt KDF also specifies the work factor, (2) that the server essentially gets to pick s, and (3) that the client does not check that s actually matches the value it itself created earlier. As a result, a malicious server can obtain a drastically weakened password hash (i.e., execute a ρ-*password-weakening* attack for $\rho = 8$).

More in detail, the format of the salt s follows the *modular crypt format*, i.e., is of the form

```
$2a$cost$salt
```

where `2a` designates the format (bcrypt salt), `cost` is the cost factor and `salt` a random string (base64 encoded). Now an *active* adversary (playing the role of the server) can attack the client by sending a value s' such that $s'.\texttt{cost} < s.\texttt{cost}$, and in this way, obtain a much weaker hash h'. The question is now: how low can `cost` go, or put differently, what is the greatest ρ that the adversary can obtain? In a normal execution `cost` is set to 12 meaning bcrypt does 2^{12} iterations during the key derivation [30]. By inspecting the source code[11] of the bcrypt implementation used by SpiderOakONE we find that the lowest value of `cost` allowed is 4. Thus an 8-*password weakening* attack can be achieved by asking the client to run the bcrypt login protocol with the salt s' equal to

```
$2a$04$AAAAAAAAAAAAAAAAAAAAAA
```

which let the adversary obtain the weakened hash $h' = \mathsf{bcrypt}(pw, s')$.

$$\Pr[b' = b] = \frac{1}{2} + \frac{q}{c2^{\kappa-8}}$$

[11] `https://github.com/grnet/python-bcrypt/blob/master/bcrypt/bcrypt.c`

```
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: base64 decoding function used. `csalt` will contain uninitialized memory if either `c1` or `c2` (content of `data`) is not valid base64.

Remember that in a regular execution `cost` is set to $c = 2^{12}$ and in our attack this is downgraded to 2^4. Thus $\mathcal{A}$ has the following probability of computing pw from h' in q queries:

$$\Pr[\mathcal{A} \text{ guesses } pw] = \frac{q}{2^4 2^\kappa} = \frac{q}{c2^{\kappa-8}}.$$

which implies the claim.

Upon inspection of the code for the bcrypt library we also discovered a trivial memory leak (relevant parts shown in Listing 1). If the `salt` part of s is invalid base64, then up to 16 bytes of memory will be leaked through h'. We can therefore augment the attack from before, by instead using the salt

```
$2a$04$0x01AAAAAAAAAAAAAAAAAAAA
```

where `0x01` is the byte 00000001.

Experimental verification. We verified this attack by writing our own login server. As described in Section 3.2 there are essentially two different servers: One which "talks" HTTP and one which talks RPC (using Twisted PB). Moreover, the HTTP server is only used during authentication (which is relevant for this attack), so we only had to implement the HTTP part of the server to verify this attack. Our server works as one would imagine: upon a login request, the server constructs the salt as specified and sends it back. Upon obtaining h we can then verify that the salt we sent was indeed the one used (as we of course also know the password used in the test run).

Practical impact. As described, this attack effectively removes 8 bits of entropy from the user's password. Whether this leads to a breach of confidentiality or not ultimately depends on the original strength of the password. For reference, [7] estimates that typical passwords only provide between 10 and 20 bits of security. We note that this attack is not detectable from the user point of view.

4.2 Password recovery via escrow login

We move on to the attack on the escrow login protocol described briefly in Section 3.3 (cf. B.3 for the details). Recall that this protocol is in two steps and revolves around a list $lst = \{pk_i\}_{i\in[n]}$ essentially chosen by the server:

1. First, the client computes a fingerprint on lst and shows it to the user;
2. Second, assuming the user accepts the fingerprint, an encryption $c = E_{pk_n}(\ldots E_{pk_1}(pw \mid\mid chl))$ is computed and returned to the server.

Looking at the protocol, we can see that the server is the one who picks the keys pk_i. Therefore, the attack exploits the fact that the server can (maliciously) pick keys for which it knows the corresponding private keys and thus decrypt c to obtain pw when the client sends back c in the last step. Note that the attack can only happen if the user accepts the shown fingerprint fp' that is computed on the (malicious) keys in lst. However, as we shall see, due to the wording of the message to the user, it is not unlikely that a user might accept a malicious set of keys.

Having the user check the fingerprint is obviously done in order to ensure that she does not produce an encryption under maliciously chosen keys. But what if she does not have anything to check the fingerprint against? More precisely, since the adversary can run any login protocol he wants, he can run this particular protocol which is *never* run in the single user settings. Thus, when the protocol is run (in the single user setting) it can be assumed that the user *does not have a "valid" fingerprint to verify the malicious fingerprint against!* Of course, the attack would be thwarted if the client instructed the user to reject a fingerprint if there is nothing to check it against. Unfortunately, SpiderOakONE takes a TOFU (trust on first use) approach with regards to these fingerprints. The message shown to the user is presented here, emphasis ours:

> 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.

So assuming the user behaves according to the instructions given by the application, our attack will succeed with significant probability.

Formally, the attack proceeds as follows: the attacker $\mathcal{A}$ generates an RSA keypair (pk, sk) and then requests the client to execute the escrow login protocol with $lst = \{pk\}$ and $chl = 0$, thus receiving an encryption of the password pw using pk which can be decrypted using sk leading to *password recovery* and a total breach of confidentiality

An interesting quirk. A missing length check in the client means that a similar attack is possible even in a passive setting (albeit in the enterprise product). Suppose the escrow login protocol is used as follows: A company uploads lst to SpiderOak and gives fp (the fingerprint computed on lst) to each of its employees. Whenever an employee wants to use SpiderOak (for work) they use the escrow protocol to login. SpiderOak authenticates the employee by passing c back to the company who can decrypt it and check the challenge stored alongside the password (and thus determine if the employee should be authenticated or not). However, if the company *misconfigures* lst as $lst = \emptyset$ then *no* encryption is done in the client! That is, $c = pw \mid\mid chl$ and thus pw is leaked to SpiderOak.

Experimental verification. Verification of this attack was performed in a similar way as what was described for the attack on the bcrypt login protocol. We verified that both $lst = \{pk\}$ for a pk we control (i.e., know sk) and $lst = \emptyset$ leads to a full password recovery.

Practical impact. While the effect of this attack are more devastating than the previous one (as it leads to full password recovery), this attack can easily be detected since it requires the user being prompted and accepting a dialog box. Unfortunately, as we shall see in the next attack, there is another (undetectable) way that allows a rogue server to recover the user's password.

4.3 Password recovery via RPC methods

The third attack we present also leads to full recovery of the user's password. Even more, it does so *completely* silently and *at any point the client is online* (as opposite to the first two which can only be executed during the login/authentication phase), and in addition requires no interaction from the user. We also show that, even if the user takes extra steps to thwart the full password recovery attack, the attack would still lead to a significant *password weakening* attack.

We first note that the SpiderOakONE the client writes the user's password (unencrypted) to a file after the first login (which is done as part of the account registration or device registration phase), in order to avoid having the user type in their password on every startup. Therefore, the user's password constantly resides in plaintext on the client. We can exploit this by making use of specific RPC methods the client makes available to the server. Concretely, the client has three methods available that allow for file retrieval. Such methods implement

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

Listing 2: Regular expression used by SpiderOakONE to check if a file retrieval request should be allowed

some security checks to prevent the server from retrieving *any* file stored on the client. Unfortunately we discovered that the checks in place do not prevent the server from requesting the file with the user's password. Each of the three methods works in essentially the same way: They take as input a filename, check if this filename satisfies a certain regular expression (shown in Listing 2) and if so, return the content of the file to the server. Two of the three methods are available by default and the last is available if the user has enabled remote diagnostics. The file that stores the user's password is located at the following two locations[12]

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

By inspecting the regular expression in Listing 2 we see that, while the first file does not match (due to the `.` before the `/`), the second file does.

Therefore, an attacker $\mathcal{A}$ can simply request the client to execute one these insecure RPC and immediately recover the password.

As mentioned in Section 3.3, users can opt-out from the automatic login functionality and instead choose to input the password at every startup. When this option is enabled, the plaintext password pw is not stored on the client. In its place, the client will store a hashed version of the password which is used by the client to verify the password input by the user. Unfortunately the stored hash is quite weak, and is calculated as in (1) where u is a value picked by the server during account registration.

$$\begin{aligned} tmp &= \mathsf{MD5}(\text{"password\_verify"} \mid\mid u \mid\mid pw), \\ h &= \mathsf{MD5}(tmp \mid\mid \text{"password\_verify"} \mid\mid u \mid\mid pw), \end{aligned} \tag{1}$$

Therefore, even more "paranoid" users choosing the stronger security settings are not immune from the attack, since even in this case the adversary can run the attack and learn the weakened hash of the password thus leading to a ρ-*password weakening attack* with $\rho = \lg(c) - 1 = 11$.

Experimental verification. Implementing this attack required more work than the the previous one, since relatively little documentation exists for the

[12] Locations are relative to the SpiderOakONE configuration directory. E.g., `$HOME/.config` on Linux.

used RPC interface (as opposed to the simple HTTP communication used during authentication). In addition, as many different RPCs are called on the client, it was not straightforward to inject extra calls to the insecure remote methods, as this would require our Man-in-the-Middle application to keep track of every call made after the injected call, and adjust sequence numbers accordingly to avoid having the client crashing after completion of the attack. Therefore, instead of injecting new calls between the real SpiderOak server and our client we changed an *existing* call originating at the real server to one which requests the file containing the user's password. Then, from the clients response, we verified that the password could be successfully extracted.

Practical impact. As already stated, this attack is extremely dangerous as it is completely undetectable and it allows to recover the password from users running with standard settings *at any time*. Even in the case where users chose the more conservative settings, the attacks is still effective in weakening the password hash.

4.4 File recovery in directory sharing

The last attack we describe can be run by a *passive* adversary e.g., it only requires the adversary to be able to observe the server (as opposed to the previous three that all require $\mathcal{A}$ to make the server deviate from normal behaviour in some way).

In a nutshell, we discovered that a shared directory cannot be securely un-shared. That is, files added to a directory *after* it has been un-shared can *still* be read by the server; the same goes for files moved out of a directory *before* it is shared. The following two scenarios illustrate these observations:

Scenario 1: Suppose Alice and Bob begin a new relationship, and therefore Alice decides to share a directory of photos with Bob. After some time, Alice and Bob break up and therefore Alice stops sharing this directory with Bob. Afterwards, she uploads new photos to the same directory assuming that Bob will be unable to see them. However, Bob, who has read access to the PECS, can also see her new photos since they are protected by the same directory key which he learned while they were in a relationship.

Scenario 2: Suppose Alice was on a vacation where she took a lot of photos, all stored in her PECS directory Dir. She would really like to share these photos with her colleagues. Alas, a few of the photos are a bit too private in their nature and thus cannot be shared. To solve this dilemma, Alice simply moves the few private pictures to a different directory Dir′ and then shares Dir. Now everyone can see her cool vacation photos. However, Bob, who has read access to the PECS, can also see her private photos, as these are not securely detached from Dir!

Scenario 2 arise because no re-encryption happens when a file is moved from one directory to another. In particular, a file that was encrypted with the

directory key k of Dir, is *still* encrypted with k even after it has been moved to another directory Dir'. Thus, when Dir is shared, the file that was moved inadvertently gets shared as well (recall that sharing reveals the directory key to the server cf. Section 3.5). The same idea applies in Scenario 1: When a directory is un-shared, the previously revealed directory key k is *not* invalidated, thus making new files added at a later time readable to the server.

Both scenarios can be phrased in terms Definition 1 and lead to *file recovery* attacks: In Scenario 1 $\mathcal{A}$ makes the client upload some file in a directory, share the directory (thus learning the directory key) and unshare the directory. Finally $\mathcal{A}$ makes the client upload an unknown file to that directory and decrypts it using the directory key; In Scenario 2 $\mathcal{A}$ makes the client upload an (unknown) file to a directory, then moves the file to a different directory and finally shares the first directory, thus learning the directory key which allows to recover the file. Note that since no *revealing* commands were performed on the retrieved file both are "valid" attacks.

Experimental validation. Validation of these attacks was carried out by executing the steps described above on the client. E.g., uploading a file to some directory, moving the file to another directory and then sharing the first directory. We then recorded all traffic generated by the client and used it to extract the file.

Practical impact. As shown, the attacks presented here lead to file recovery for files that, one way or another, users *actively* choose to prevent from being shared and is therefore critical.

4.5 On forward secrecy

As mentioned in Section 3.4, very little is changed when a SpiderOakONE user changes their password which implies that SpiderOakONE does not satisfy our notion of *forward security*. This is quite critical since, as shown by the previous attacks, there are several ways in which a user's password might leak.

The attack can be easily described: when an account is created a number of long-term secrets are created and encrypted using the password. The long-term secrets are in turn used to encrypt the individual directories and files. Thus, if the password pw is leaked, the adversary $\mathcal{A}$ will be able to retrieve the long-term secrets (and in turn the content of all existing files and directory). Now when the user changes their password the long-term secrets are not replaced by new ones, thus from $\mathcal{A}$'s point of view the password change has no effect at all! Now, when the user uploads a new file, $\mathcal{A}$ can simply retrieve it using the (unchanged) long-term secrets, thus breaking *forward secrecy*.

5 Mitigations

We reported our findings on April 5th, 2017. As of September 28th, 2017, all reported issues have been fixed by SpiderOak [35]. The fixes implemented by SpiderOak are summarized in this section:

4.1: A check was implemented that (1) ensures `cost` is at least 12 (thus preventing a downgrade), and (2), that the salt part of $\mathsf{bcrypt}(pw, s)$ is equal to s (thus preventing the memory leak). It is worth noting that the memory leak is still present, it is just not exploitable anymore.

4.2: The client now ensures that lst is non-empty. In addition, the message displayed to the user has been reworded to be less ambiguous.

4.3: SpiderOak claimed that the functions allowing for (almost) arbitrary file retrieval was part of an API that never got implemented. As a result, their fix was simply to remove them.

4.4: Keys are now properly rotated, ensuring that files are not encrypted under a key that has been previously revealed.

4.5: Forward-secrecy is still not supported by SpiderOakOne.

6 Conclusion

In this paper we described a number of vulnerabilities which (might have) allowed a rogue SpiderOak server (or anyone able to bypass certificate pinning and manage to interact with the client software) to break 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 out 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 Section 4.5, 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 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.

References

[1] Modular crypt format: A explanation about a standard that isn't. `http://passlib.readthedocs.io/en/stable/modular_crypt_format.html`.

[2] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Keys to the cloud: formal analysis and concrete attacks on encrypted web storage. In *International Conference on Principles of Security and Trust*, pages 126–146. Springer, 2013.

[3] Andrey Belenko and Dmitry Sklyarov. "secure password managers" and "military-grade encryption" on smartphones: Oh, really? *Blackhat Europe*, 2012.

[4] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. Cryptology ePrint Archive, Report 2012/631, 2012. `http://eprint.iacr.org/2012/631`.

[5] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 312–329, 2012.

[6] 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.

[7] Joseph Bonneau. The science of guessing: analyzing an anonymized corpus of 70 million passwords. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 538–552. IEEE, 2012.

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

[9] Stephen Checkoway, Jacob Maskiewicz, Christina Garman, Joshua Fried, Shaanan Cohney, Matthew Green, Nadia Heninger, Ralf-Philipp Weinmann, Eric Rescorla, and Hovav Shacham. A systematic analysis of the juniper dual EC incident. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 468–479, 2016.

[10] 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.

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

[12] John R Douceur, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 617–624. IEEE, 2002.

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

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

[15] Gemalto. 2016 mining for database gold. `http://breachlevelindex.com/assets/Breach-Level-Index-Report-2016-Gemalto.pdf`, 2016.

[16] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 1069–1083, 2016.

[17] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[18] 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.

[19] 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.

[20] 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`.

[21] Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 584–597, 2007.

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

[23] Seny Kamara, Kristin E Lauter, et al. Cryptographic cloud storage. In *Financial Cryptography Workshops*, volume 6054, pages 136–149. Springer, 2010.

[24] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition.* CRC Press, 2014.

[25] 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.

[26] 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.

[27] 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`, 07 2014.

[28] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. The emperor's new password manager: Security analysis of web-based password managers. In *USENIX Security Symposium*, pages 465–479, 2014.

[29] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar R. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.

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

[31] 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`, 02 2017.

[32] P Rogaway and T Shrimpton. Deterministic authenticated-encryption. In *Advances in Cryptology–EUROCRYPT*, volume 6, 2007.

[33] SpiderOak. Encryption white paper. `https://spideroak.com/resources/encryption-white-paper`.

[34] SpiderOak. Building for new threat models in a post-snowden era. `https://spideroak.com/articles/building-for-new-threat-models-in-a-postsnowden-era`, 6 2017.

[35] SpiderOak. security update for spideroak groups & one; bugs reported & resolved. `https://spideroak.com/articles/security-update-for-spideroak-groups-one-bugs-reported-resolved/`, 9 2017.

[36] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310, 2013.

[37] Mark W. Storer, Kevin Greenan, Darrell D.E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, StorageSS '08, pages 1–10, New York, NY, USA, 2008. ACM.

[38] 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.

[39] 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.

[40] F. F. Yao and Y. L. Yin. Design and analysis of password-based key derivation functions. *IEEE Transactions on Information Theory*, 51(9):3292–3297, Sept 2005.

A Notation

We explicitly denote the KDFs that will be used: $\mathsf{bcrypt}(pw, s)$ [30] where pw is a password and s a modular crypt formatted salt (see e.g., [1]); $\mathsf{PBKDF2}(pw, s, c)$ [22], where pw is a password, s a random string and c an integer denoting an iteration count. By writing $\mathsf{Enc}^i_k(iv, m)$ (resp. $\mathsf{Dec}^i_k(iv, m)$) we mean an AES-CFB encryption (resp. decryption) of message m under key k, using segment size i (cf. [13]) and initialization vector iv. Similar notation is used for RSA encryption, and we use $|pk|$ to denote the bit-length of the modulus. Assignment is written as $x := y$ and random sampling as $x \xleftarrow{\$} X$. Finally, $x_{i,j}$ denotes the bit sub-string $x_i, \ldots, x_{j-1}$ of x; and by $|x|$ we mean x's length in bits.

B Authentication Protocols

Protocol names (e.g., "pandora/zk") reflect the value of the short identifier the server uses to instruct the client which protocol is to be run.

We will always use pw to denote the user's password and usr to denote their username.

B.1 pandora/zk (login)

Define ck as $ck := \mathsf{PBKDF2}(\mathsf{pwd}, s_1, 16384)$ where $s_1 \xleftarrow{\$} \{0,1\}^{256}$. Assume both client and server knows ck. (The server learns ck after an account registration where the client sends it.)

Client: Send usr to the server.

Server: Let $k \in \{0,1\}^{256}$, $iv \in \{0,1\}^{128}$ and let $tv \in \{0,1\}^{32}$ be the current server time. Using usr, find values s_1 and ck. Compute $c := \mathsf{Enc}^8_{ck}(iv, k)$ and send (tv, iv, s_1, c) to the client.

Client: Compute $k' = \mathsf{Dec}^8_{ck}(iv, c)$, $a := \mathsf{Enc}^8_{k'}(iv, tv)$ and send (iv, tv, c, s_1, a) back to the server.

Server: Accept if $tv = \mathsf{Dec}^8_k(iv, a)$.

B.2 bcrypt (account registration)

We repeat the description from Section 3.3 here:

Client: Send usr to the server.

Server: Using usr, find a salt s and bcrypt hash h. Send s' to the client.

Client: Compute $h' := \mathsf{bcrypt}(pw, s)$ and send h' to the server.

Server: Reject if $h' \neq h$.

B.3 escrow/challenge (i.e., escrow login)

We first describe the two procedures FP (**f**inger**p**rint) and LE (**l**ayered-**e**ncryption):

$\mathsf{FP}(lst)$

lst is a list of keys and ids. We use a counter n to denote its length. Define lst as

$$lst := \begin{cases} \{(id_1, pk_1), \ldots, (id_n, pk_n)\}, & \text{if } n > 0 \\ \emptyset. & else. \end{cases}$$

A hash h of l is computed as

$$\begin{aligned} &l' := [\,], \\ &l' := l' \,||\, id_i \,||\, \mathsf{E}(pk_i), \text{ for } (id_i, pk_i) \in lst \wedge i = 1, \ldots n \\ &h := \mathtt{sha256}(l'), \end{aligned}$$

(Step 2 is skipped if $n = 0$.) $\mathsf{E}(x)$ performs a DER encoding [20] of x. Use the key2eng procedure from [11] to obtain 24 words $w_0, \ldots, w_{23}$. Output the fingerprint $fp = w_0 \,||\, w_2 \,||\, \ldots \,||\, w_{22}$ (i.e., only words at even indexes are used).

$\mathsf{LE}(pw, lst, chl)$

lst is defined as before and $chl \in \{0,1\}^*$. Let $auth = chl \,||\, \mathsf{pwd}$ and do for every pair $(id_i, pk_i) \in lst$:

1. Pick $x_i \xleftarrow{\$} \{0,1\}^{|pk_i|-8}$, let tv denote the current system time and define $iv_i := \mathtt{sha256}(tv)_{0,16}$.
2. Compute

$$\begin{aligned} A &:= \mathsf{Enc}^8_{\mathtt{sha256}(x_i)}(iv_i, auth) \\ B &:= \mathsf{RSAEnc}_{pk_i}(x_i), \\ auth &:= id_i \,||\, A \,||\, B \,||\, iv_i. \end{aligned}$$

Return $auth$.

The protocol in its entirety then goes as described in Section 3.3:

Client: Send usr to the server.

Server: Retrieve lst and chl associated with usr.

Client: Compute $fp = \mathsf{FP}(lst)$ and show fp to the user (i.e., the human). If the user accepts the fingerprint, continue. Otherwise the client aborts the protocol.

Client: (If the user accepted fp) Compute $auth = \mathsf{LE}(pw, lst, chl)$ and send $auth$ to the server.

Note that, as we did not observe this protocol being used during normal interaction with the server, we cannot say what criteria has to be satisfied, for the server to authenticate the client.

B.4 pandora/zk/sha256

We do not know when or where this protocol is used.

Client: Retrieve value s_1 from local storage and compute ck as in B.1. Send ck to the server.

For the same reasons as in the previous protocol, we do not know how the server should react to the client's message.

C Keys

Figure 2 shows the relationship between various secrets in SpiderOakONE. Key encapsulation is done in different ways, depending on the "layer" in Figure 2:

1. The user's password protects an RSA keypair kp;
2. kp protects a special long term secret, k_{sym};
3. k_{sym} protects all other long term secrets;
4. jk (a long term key) protects directory keys; and
5. Each directory key protects a set of file encryption keys corresponding to files stored in the corresponding directory.

A technical description of all but the last step follows (the last step is treated in Section D).

C.1 Step 1

Let $kp := (sk, pk)$ be a 3072-bit RSA keypair, and $s_2 \xleftarrow{\$} \{0,1\}^{256}$. Compute

$$\begin{aligned} k &:= \mathsf{PBKDF2}(pw, s_2, 16834), \\ iv &:= \mathtt{sha256}(\text{"keypair"} \mid\mid s_2)_{0,16}, \\ c_{\text{keypair}} &:= \mathsf{Enc}^8_k(iv, kp). \end{aligned}$$

C.2 Step 2

Write $(sk, kp) = kp$, let $k_{sym} \xleftarrow{\$} \{0,1\}^{3064}$ and compute

$$\begin{aligned} c &:= \mathsf{RSAEnc}_{pk}(k_{sym}), \\ s &:= \mathsf{RSASign}_{sk}(\mathtt{sha256}(c)), \\ c_{sym} &:= (c, s). \end{aligned}$$

(Note: "textbook RSA" is used, which explains the number 3064 as it is exactly 1 byte smaller than the size of the modulus.)

C.3 Step 3

Let $k \xleftarrow{\$} \{0,1\}^{\ell}$ where ℓ is the length of this particular long term key; let id be its name (in step 1, $id =$ "keypair") and let $miv \xleftarrow{\$} \{0,1\}^{2048}$. Compute:

$$\begin{aligned} kk &:= \mathtt{sha256}(k_{sym}) \\ iv &:= \mathtt{sha256}(miv)_{0,16}, \\ c_{id} &:= \mathsf{Enc}^8_{kk}(iv, k). \end{aligned}$$

C.4 Step 4

For $k = jk$, we have $\ell = 256$ and $id =$ "journalkey". Let $dk_i \xleftarrow{\$} \{0,1\}^{256}$ and id be a unique ID for this directory. (In fact, this ID will be unique across *all* accounts in the system.) Compute:

$$\begin{aligned} iv &:= \mathtt{sha256}(miv \mid\mid \text{"journal"} \mid\mid id \mid\mid \text{".key"})_{0,16} \\ c_{id} &:= \mathsf{Enc}^8_{jk}(iv, dk_i). \end{aligned}$$

C.5 Remark on password change

As noted, a password change does not effectively prevent an old password from being useful in the future. To see why, we note that, upon a password change, the client only recomputes c_{kp} but otherwise leaves everything as is. I.e., the secrets recoverable with the old password, will *still* be the same secrets in use with the new password.

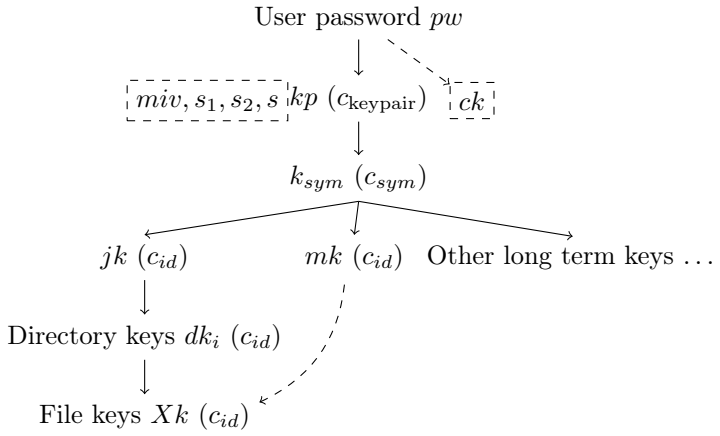

Figure 2: Key tree. A solid arrow from A to B means that A is used to "protect" B in some way (e.g., B is encrypted under A), while a dashed arrow means A is used to derive B in some way. Values in a dashed box are the public values (i.e., client and server both possess them). Values inside parenthesis are possessed by the server and id's are distinct.

D File Encryption

We describe in technical detail the different encryption methods mentioned in Section 3.5.

D.1 Metadata Encryption

An encrypted metadata file can be seen as a bit-string $rn \,||\, rs \,||\, c$ where

$$|rn| = |rs| = 32 \quad \text{and} \quad |c| = \frac{rs}{8}.$$

That is, both rn and rs are 4-byte integers and rs describes the size of c (the ciphertext). rn describes a record number and is used in IV creation and encryption in the following way. Suppose m is the piece of metadata to be encrypted:

1. Find the highest rn^* among all stored encryptions. For the new encryption, set $rn := rn^* + 1$;
2. Compute $iv := \texttt{sha256}(miv \,||\, rn)_{0,16}$;
3. Let k be an encryption key (this k is always one of the long-term keys);
4. Compute $c := \mathsf{Enc}^8_k(iv, m)$, $rs = |c|/8$ and define the new encryption as $rn \,||\, rs \,||\, c$.

D.2 User file encryption

Let F be a file uploaded by the user to a directory with directory key dk. Encryption of F proceeds as follows:

1. partition F into n blocks $b_0, \ldots, b_{n-1}$ each of some (not necessarily equal) size. The client treats block as a separate file, so let $b_i.id$ denote the id of block b_i. For each block b_i, do

   (a) Compute

   $$iv_i := \texttt{sha256}(\text{"block"} \,||\, b_i.id \,||\, miv)_{0,16},$$
   $$bk_i := \texttt{sha256}(b_i \,||\, mk),$$

   where $mk \xleftarrow{\$} \{0,1\}^{2048}$ is a long-term secret (the previously mentioned "master key").

   (b) Encryption of b_i:

   $$c_i := \mathsf{Enc}^{128}_{bk_i}(iv_i, \mathsf{pad}(b_i)),$$
   $$ebk_i := \mathsf{Enc}^{8}_{dk}(iv_i, bk_i),$$

   where $\mathsf{pad}(x)$ applies an ANSI X.932 padding to x. Define the encryption of b_i as $cb_i := ebk_i \,||\, c_i$.

   Having so obtained an encryption for each b_i, compute

   $$vk := \texttt{sha256}(F \,||\, mk),$$
   $$iv := \texttt{sha256}(\text{"version"} \,||\, F.id,\, miv)_{0,16}.$$

   Let $bl := [b_0.id, \ldots, b_{n-1}.id]$, compute $c_F := \mathsf{Enc}^{8}_{vk}(iv, bl)$ and $evk_F := \mathsf{Enc}^{8}_{dk}(iv, vk)$. Output c_i for $i = 0, \ldots, n-1$ and $cv_F := evk_F \,||\, c_F$.

D.3 File sharing

Observe that cv_F describes the exact blocks making up the file F. Thus, a file sharing of F has to include also cv_F. File sharing then proceeds in the way described in 3.5, namely:

Single files: The client first recovers each $b_i.id$ from cv_F. From $b_i.id$, the corresponding cb_i can be found, and from cb_i, the client extracts bk_i from ebk_i and sends $\{bk_i\}_{i=0}^{n-1}$ as well as evk_F to the server.

Directory: Sharing a whole directory works in much the same way as with single files. However, instead of recovering each individual file encryption key, the directory key dk is shared instead.