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

Anders Dalskov Claudio Orlandi

Aarhus University

RWC 2018

Agenda

- A Threat Model for Encrypted Cloud Storage (ECS).
- A high-level look into a modern ECS service SpiderOakONE.
- Attacks on SpiderOakONE and what we can learn from them.

Agenda

- A Threat Model for Encrypted Cloud Storage (ECS).
- A high-level look into a modern ECS service SpiderOakONE.
- Attacks on SpiderOakONE and what we can learn from them.

Disclaimer: All issues were reported on June 5th 2017 responsibly, and are fixed in version 6.4.0 of SpiderOakONE.

(Password) Encrypted Cloud Storage

Traditional Cloud Storage raises some privacy concerns:

(Password) Encrypted Cloud Storage

Traditional Cloud Storage raises some privacy concerns:

- Besides us, who can read our files?

(Password) Encrypted Cloud Storage

Traditional Cloud Storage raises some privacy concerns:

- Besides us, who can read our files?
- What happens to the files we delete? Or when we close our account?

(Password) Encrypted Cloud Storage

Traditional Cloud Storage raises some privacy concerns:

- Besides us, who can read our files?
- What happens to the files we delete? Or when we close our account?
- What if the Cloud Storage company is sold?

(Password) Encrypted Cloud Storage

Traditional Cloud Storage raises some privacy concerns:

- Besides us, who can read our files?
- What happens to the files we delete? Or when we close our account?
- What if the Cloud Storage company is sold?

Solution: Encrypt files on the client before sending them to the server.

Threat Model

ECS should provide *more security* than Traditional Cloud Storage:
We want our files to stay secure **even if the server turns malicious.**

Threat Model

ECS should provide *more security* than Traditional Cloud Storage:
We want our files to stay secure **even if the server turns malicious.**

ECS providers seem to agree:

- Tresorit: *We believe you should never have to 'trust' a cloud service*
- LastPass: *No one at LastPass can ever access your sensitive data.*
- sync: *We can't read your files and no one else can either*
- pCloud: *No one, even pCloud's administrators, will have access to your content*
- SpiderOak: *No Knowledge means we know nothing about the encrypted data you store on our servers*
- ...

Threat Model

But is a “malicious server” threat model actually used?

Threat Model

But is a “malicious server” threat model actually used? For example, SpiderOak wrote (after we’d disclosed the issues we found):

> *When we started building SpiderOak in 2006, the threat model was an attacker who would want to compromise SpiderOak and steal customer data [...] Because this was a legacy mindset, the SpiderOak ONE backup code base is not robust against a different kind of threat model: SpiderOak, the company, as the active attacker*

Threat Model

But is a "malicious server" threat model actually used? For example, SpiderOak wrote (after we'd disclosed the issues we found):

> *When we started building SpiderOak in 2006, the threat model was an attacker who would want to compromise SpiderOak and steal customer data [...] Because this was a legacy mindset, the SpiderOak ONE backup code base is not robust against a different kind of threat model: SpiderOak, the company, as the active attacker*

Previous work that has examined ECS (SpiderOakONE in particular):

- *Bhargavan et al (2012)*: External adversary. CSRF in web interface that could be used to learn location of shared files.
- *Wilson & Ateniese (2014)*: Only considers file sharing. Found that the server can read files shared by the user.

Threat Model—Our attempt

Assume an honest client (client software obtained *before* server turns malicious).

Threat Model—Our attempt

Assume an honest client (client software obtained *before* server turns malicious).

Informally, we try to answer the questions:

Threat Model—Our attempt

Assume an honest client (client software obtained *before* server turns malicious).

Informally, we try to answer the questions:

1. Are we secure against a **passive** adversary? I.e. is the client's default behaviour secure?

Threat Model—Our attempt

Assume an honest client (client software obtained *before* server turns malicious).

Informally, we try to answer the questions:

1. Are we secure against a **passive** adversary? I.e. is the client's default behaviour secure?
2. Are we secure against an **active** adversary? Is the protocols secure against misuse? What about the client implementation?

Threat Model—Our attempt

Assume an honest client (client software obtained *before* server turns malicious).

Informally, we try to answer the questions:

1. Are we secure against a **passive** adversary? I.e. is the client's default behaviour secure?
2. Are we secure against an **active** adversary? Is the protocols secure against misuse? What about the client implementation?

Formally: Indistinguishability experiment between an oracle (client) and adversary (server).
Our definition only considers confidentiality. Refer to our paper for the details:
`https://eprint.iacr.org/2017/570`

SpiderOakONE—Quick facts

SpiderOakONE is an ECS with praise/endorsements from both Edward Snowden and the EFF.

Uses "*No Knowledge*" (and "*Zero Knowledge*" before that) to describe their encryption routines.

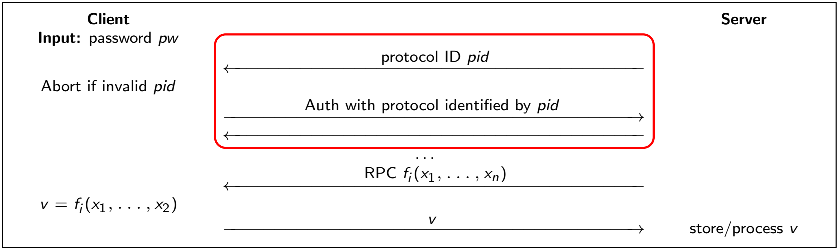
- Supports Windows, Mac and Linux (partial support for Android and iOS),
- File sharing (single files and whole directories),
- Written in Python $\implies$ decompilation is easy,
- Certificate Pinning + TLS $\implies$ limits scope of attacks.

Our review focused on version 6.1.5, released July 2016.

SpiderOakONE—Communication

Client		**Server**
Input: password *pw*		
	protocol ID *pid* ←	
Abort if invalid *pid*		
	Auth with protocol identified by *pid* →	
	←	
	...	
	RPC $f_i(x_1, \ldots, x_n)$ ←	
$v = f_i(x_1, \ldots, x_2)$		
	v →	store/process v

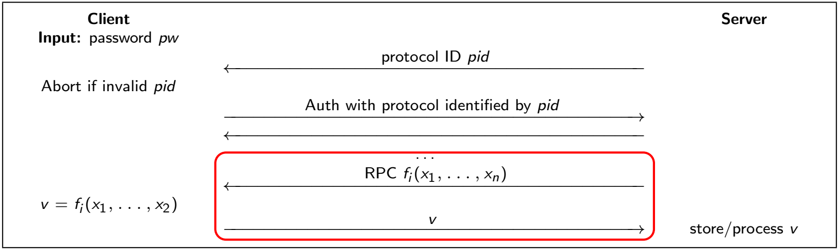
SpiderOakONE—Communication



Authentication:

- Only run on first install.
- Server picks what protocol to run. (4 possible, but only 2 were observed.)
- All protocols are non-standard (i.e. "home-made").

SpiderOakONE—Communication



Authentication:

- Only run on first install.
- Server picks what protocol to run. (4 possible, but only 2 were observed.)
- All protocols are non-standard (i.e. "home-made").

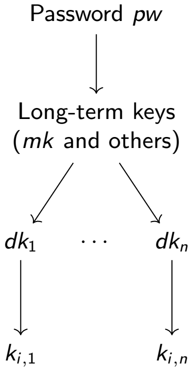
RPC:

- Everything else (data transfer, device stats, etc.)
- Comprehensive: Server can call ≈ 90 different procedures on the client.

SpiderOakONE—Encryption

User files:

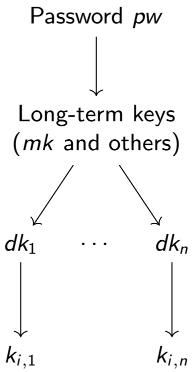
- File F is encrypted with $k_F = H(F \mid\mid mk)$;
- k_F is encrypted with a per-directory key dk_i;
- dk_i is encrypted with a per-account long-term key;
- long-term keys are encrypted with $k = KDF(pw)$.



SpiderOakONE—Encryption

User files:

- File F is encrypted with $k_F = H(F \parallel mk)$;
- k_F is encrypted with a per-directory key dk_i;
- dk_i is encrypted with a per-account long-term key;
- long-term keys are encrypted with $k = KDF(pw)$.



Password changes: A password change only triggers re-encryption of the long-term secrets. I.e. no “future secrecy”.

Attacks

We found 4 different issues that can be leveraged for attacks on the client:

- 1 attack weakens the security of a hash derived from the user's password (we'll skip this);
- 2 attacks recover the user's password—one is completely silently!
- 1 attack can in some situations recover files that are not supposed to be shared (during sharing of other files).

All but the last attack is active.

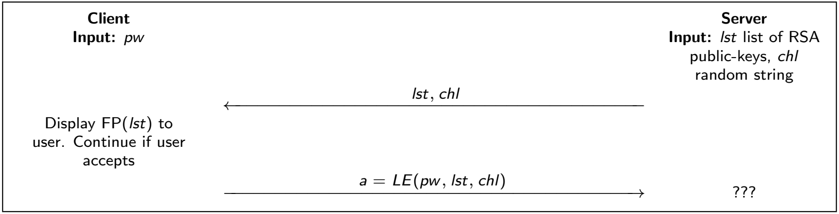
Verification: All attacks was implemented and verified to work against version 6.1.5 of SpiderOakONE.

Password recovery

Recall: 2 authentication protocols were seen, yet 4 can be run.

Password recovery

Recall: 2 authentication protocols were seen, yet 4 can be run.



- FP(*lst*) computes a "fingerprint" on *lst* using RFC1751;
- LE(*pw*, *lst*, *chl*) computes a "layered encryption" of *pw* and *lst* using keys from *lst*. I.e.

$$a = \mathrm{Enc}_{pk_n}(\mathrm{Enc}_{pk_{n-1}} \ldots (\mathrm{Enc}_{pk_1}(pw \mid\mid chl))).$$

Password recovery

Recall: 2 authentication protocols were seen, yet 4 can be run.

Client		Server
Input: *pw*		**Input:** *lst* list of RSA public-keys, *chl* random string
	← lst, chl	
Display FP(*lst*) to user. Continue if user accepts		
	$a = LE(pw, lst, chl)$ →	???

- FP(*lst*) computes a "fingerprint" on *lst* using RFC1751;
- LE(*pw*, *lst*, *chl*) computes a "layered encryption" of *pw* and *lst* using keys from *lst*. I.e.

$$a = \mathsf{Enc}_{pk_n}(\mathsf{Enc}_{pk_{n-1}} \ldots (\mathsf{Enc}_{pk_1}(pw \mid\mid chl))).$$

Issue: Server can pick pk_i s.t. it knows sk_i, which lets it recover *pw* from *a*.

Password recovery

$FP(lst)$ changes when lst changes. But what should the user compare the fingerprint to?

Password recovery

$FP(lst)$ changes when *lst* changes. But what should the user compare the fingerprint to? TOFU:

> *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.*

I.e. if the user does not have anything to compare $FP(lst)$ against, then they should just accept.

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server;

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another;

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another; and (3) directory keys are never rotated.

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another; and (3) directory keys are never rotated.

Scenario 1:

1. Directory D with files $F_1, F_2, \ldots, F_n$;
2. Move F_i to D' and then share D;

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another; and (3) directory keys are never rotated.

Scenario 1:

1. Directory D with files $F_1, F_2, \ldots, F_n$;
2. Move F_i to D' and then share D;
3. But, F_i is encrypted with dk_D (obs. 2), which server knows (obs. 1);
4. Server can recover F_i.

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another; and (3) directory keys are never rotated.

Scenario 1:

1. Directory D with files $F_1, F_2, \ldots, F_n$;
2. Move F_i to D' and then share D;
3. But, F_i is encrypted with dk_D (obs. 2), which server knows (obs. 1);
4. Server can recover F_i.

Scenario 2:

1. Directory D with files $F_1, \ldots, F_n$ is shared (server knows dk_D);
2. Sharing of D ceases;
3. File F_{n+1} is added to D;

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another; and (3) directory keys are never rotated.

Scenario 1:

1. Directory D with files $F_1, F_2, \ldots, F_n$;
2. Move F_i to D' and then share D;
3. But, F_i is encrypted with dk_D (obs. 2), which server knows (obs. 1);
4. Server can recover F_i.

Scenario 2:

1. Directory D with files $F_1, \ldots, F_n$ is shared (server knows dk_D);
2. Sharing of D ceases;
3. File F_{n+1} is added to D;
4. But, dk_D was not invalidated in step 2 (obs. 3) $\implies$ F_{n+1} is also encrypted under dk_D;
5. Server can recover F_{n+1}.

File recovery via. directory sharing (ShareRooms)

Observations: (1) sharing directory D happens by revealing dk_D (the directory key) to the server; (2) file encryptions are not updated when moving a file from one directory to another; and (3) directory keys are never rotated.

Scenario 1:

1. Directory D with files $F_1, F_2, \ldots, F_n$;
2. Move F_i to D' and then share D;
3. But, F_i is encrypted with dk_D (obs. 2), which server knows (obs. 1);
4. Server can recover F_i.

Scenario 2:

1. Directory D with files $F_1, \ldots, F_n$ is shared (server knows dk_D);
2. Sharing of D ceases;
3. File F_{n+1} is added to D;
4. But, dk_D was not invalidated in step 2 (obs. 3) $\implies$ F_{n+1} is also encrypted under dk_D;
5. Server can recover F_{n+1}.

In both scenarios, files are recovered that the user took specific steps to *avoid* sharing.

Password recovery (again)

After installation, the user's password is stored *in plaintext* on the user's machine. (This avoids the user having to input it on every boot.)

Password recovery (again)

After installation, the user's password is stored *in plaintext* on the user's machine. (This avoids the user having to input it on every boot.)

RPC methods exist that, on input a file path, will return the file's content **if** the file path matches a regular expression.

Password recovery (again)

After installation, the user's password is stored *in plaintext* on the user's machine. (This avoids the user having to input it on every boot.)

RPC methods exist that, on input a file path, will return the file's content **if** the file path matches a regular expression.

The file path for the file containing the user's password matches this regular expression.

Password recovery (again)

After installation, the user's password is stored *in plaintext* on the user's machine. (This avoids the user having to input it on every boot.)

RPC methods exist that, on input a file path, will return the file's content **if** the file path matches a regular expression.

The file path for the file containing the user's password matches this regular expression.

Attack: The server can just "ask" the client to send the user's password.

My 5 cents on secure application design

- Complexity. Many RPC methods and different authentication protocols create a large attack surface.

My 5 cents on secure application design

- Complexity. Many RPC methods and different authentication protocols create a large attack surface.
- Same secret for both authentication and encryption. All active attacks we found were the result of using the same secret (the password) for both encryption and authentication.

My 5 cents on secure application design

- Complexity. Many RPC methods and different authentication protocols create a large attack surface.
- Same secret for both authentication and encryption. All active attacks we found were the result of using the same secret (the password) for both encryption and authentication.
- Different execution contexts. The client should avoid making assumptions about the user.

Wrapping up

Talk Summary:

- Motivation for Encrypted Cloud Storage and its security requirements;
- A Threat Model for ECS. Specifically, security in the presence of an either *passive* or *active* malicious server;
- Examples of how security in a real ECS (SpiderOakONE) breaks down when the server turns malicious.

Wrapping up

Talk Summary:

- Motivation for Encrypted Cloud Storage and its security requirements;
- A Threat Model for ECS. Specifically, security in the presence of an either *passive* or *active* malicious server;
- Examples of how security in a real ECS (SpiderOakONE) breaks down when the server turns malicious.

Concluding remark:
ECS is intended to provide more, in terms of security, than traditional Cloud Storage, and the Threat Model should reflect this fact.