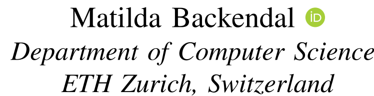
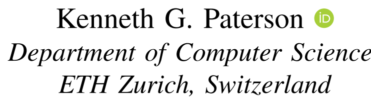
Share with Care:
Breaking E2EE in Nextcloud

Martin R. Albrecht
King's College London
London, UK

Matilda Backendal

Department of Computer Science
ETH Zurich, Switzerland

Daniele Coppola
Department of Computer Science
ETH Zurich, Switzerland

Kenneth G. Paterson

Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—Nextcloud is a leading cloud storage platform with more than 20 million users. Nextcloud offers an end-to-end encryption (E2EE) feature that is claimed to be able "to keep extremely sensitive data fully secure even in case of a full server breach". They also claim that the Nextcloud server "has Zero Knowledge, that is, never has access to any of the data or keys in unencrypted form". This is achieved by having encryption and decryption operations that are done using file keys that are only available to Nextcloud clients, with those file keys being protected by a key hierarchy that ultimately relies on long passphrases known exclusively to the users.**

We provide the first detailed documentation and security analysis of Nextcloud's E2EE feature. Nextcloud's strong security claims motivate conducting the analysis in the setting where the server itself is considered malicious. We present three distinct attacks against the feature in this setting. Each one enables the confidentiality and integrity of all user files to be compromised. All three attacks are fully practical and we have built proof-of-concept implementations for each. The vulnerabilities make it trivial for a malicious Nextcloud server to access and manipulate users' data.

We have responsibly disclosed the three vulnerabilities to Nextcloud. The second and third vulnerabilities have been remediated. The first was addressed by temporarily removing the file sharing feature from the E2EE feature until a redesign of the feature can be made. We reflect on broader lessons that can be learned for designers of E2EE systems.

1. Introduction

Cloud storage enables users to store data and files online using the storage infrastructure of a cloud computing service provider. The advantages for the users are many: outsourced files are safely backed up, can be accessed from any device, and are easily shared with other users. Additionally, features such as real-time collaborative editing make it possible to work with data stored in the cloud in a way that is not possible for local files. For these reasons, and more, the use of cloud storage has become ubiquitous among individuals and businesses alike, and estimates indicate that it is likely to account for half of all global data storage by 2025 [22].

However, the advantages come at a cost. When data leaves the safety of a user's local device, it may become exposed to a range of additional security threats. Users must place their trust in the selected cloud storage provider to not deny them access to their files. In the absence of cryptographic mechanisms, users must also trust the provider to not examine or modify their files. Additionally, users are no longer in control of how their data is stored, and must also trust the provider to protect their files from malicious third parties. The sheer amount of data stored by cloud providers, and their large user-bases, make them attractive targets for attackers; a successful attack on a cloud storage service can potentially compromise data of numerous users simultaneously.

These threats are not just hypothetical. Cloud storage breaches due to misconfigurations or lack of appropriate access management are ubiquitous and lead to exposure of potentially sensitive data. For example, in 2021, personal information about US citizens was left accessible to the public due to misconfiguration of the Amazon S3 buckets used to store the data of over 80 US municipalities [40]. Furthermore, beyond privacy issues, lack of *integrity* of data stored in cloud,[1] leads to another set of attacks, such as ransomware attacks, e-skimming and cryptojacking. These types of attacks can be performed directly by a malicious cloud storage provider, but a more likely adversary is an external actor who compromises the cloud or makes use of its lack of security to launch an attack.

Given that cloud storage systems are highly attractive targets for attackers and that they are not immune to breaches, it makes sense to minimize the amount of trust that needs to be placed in the service provider, in an attempt to decrease the negative consequences of a breach. Additionally, privacy-minded users might not want to entrust their sensitive data to a company, even if it would be perfectly secure against outsiders. Luckily, with appropriate use of cryptographic mechanisms and key management techniques, most of the required trust assumptions can be dispensed with.[2]

To meet the demand for privacy-preserving outsourced storage, multiple services offering cloud storage with client-side encryption are available, including MEGA [21], Nextcloud [28], Preveil [37], Proton Drive [38] and Tresorit [41]. These systems aim to provide end-to-end

1. By design, or due to misconfiguration.
2. Trusting the service provider with availability, that is, to not deny users access to their outsourced files, remains a necessary assumption.

encryption (E2EE), which guarantees both confidentiality and integrity of data stored in the cloud, even if the cloud provider is compromised or malicious. In addition, all of the above-mentioned service providers aim to offer file-sharing features, enabling users to securely grant, and possibly later remove, access to their files for other users.

There is still no standardized approach for E2EE cloud storage, and as recent analyses [1], [2], [17] show, *ad hoc* approaches such as that taken by MEGA can spectacularly fail to achieve the advertised security claims. In the case of MEGA, up to 1000 Petabytes of user data was left vulnerable to attacks that could be mounted in practice by a malicious or compromised service provider.

In the continued absence of a standardization effort, we are forced to rely on security audits of the existing systems in order to evaluate their security. However, the standard industry approach to performing security audits typically involves a time-limited approach that checks for standard vulnerabilities and that is usually focussed on testing for standard software vulnerabilities rather than conducting a deep investigation of the cryptographic design. This has limited value. As supporting evidence for this statement, we remark that Nextcloud and its customers had commissioned at least two different security audits[3] and yet we were able to find three different and severe cryptographic vulnerabilities in the E2EE feature.

1.1. Nextcloud

Nextcloud provides open-source software that lets individuals and businesses create and host their own cloud storage platforms. In 2017, Nextcloud estimated that it had "well over 20 million users" [25]. Nextcloud has reported having over 400,000 separate deployments in 2022, also suggesting a large number of users. However, estimating the true number of Nextcloud users is challenging, as they are spread across a range of self-hosted server instances.

Nextcloud's primary goal is to give users control over their data and to safeguard the privacy of sensitive information. Nextcloud advertises an enterprise-grade, seamlessly integrated solution for end-to-end encryption [27]. They claim "to keep extremely sensitive data fully secure even in case of a full server breach" because the server "has Zero Knowledge, that is, never has access to any of the data or keys in unencrypted form" [27]. Given these claims, it makes sense to study the security of Nextcloud's E2EE feature in the setting where the adversary has control over the server.

The platform has gained popularity among users who require advanced security features. For example, organisations such as Amnesty International and the German Federal Government [24], [29] use Nextcloud to safeguard the privacy of their sensitive data.

1.2. E2EE in Nextcloud

Here we briefly describe Nextcloud's approach to E2EE. More details can be found in Section 2.2.

Notably, E2EE is not deployed by default in Nextcloud. Instead, it is a feature that users have to enable at the folder level by marking the folder as E2EE. The Nextcloud client encrypts the files in all E2EE folders using AES-GCM (an authenticated encryption with additional data (AEAD) mode of operation of AES) with a separate key for each file. Each E2EE folder is assigned a so called "metadata key". The metadata key encrypts the file keys as well as other metadata associated with the folder, again using AES-GCM.

In order to support key rotation, the system allows each folder to have an array of associated metadata keys and only uses the one with the highest index to encrypt the folder metadata. Each time the client syncs with the server, the metadata (including all file keys) are re-encrypted with the highest indexed metadata key in the array.

Each Nextcloud user additionally holds a unique RSA key pair. The metadata keys of all folders to which the user has access are encrypted to the user's public key using RSA-OAEP. To support access from multiple devices, storage of the RSA key pairs is outsourced to the Nextcloud server. The public key is stored in clear, and the private key is encrypted using AES-GCM under a master key that is derived from a so-called mnemonic. This is a secret passphrase consisting of 12 randomly sampled words. The mnemonic is generated for the user by its client when E2EE is first enabled.

In totality, then, Nextcloud uses a key hierarchy consisting of: file keys (at the lowest level), metadata keys, RSA key pair, master key, user-memorable mnemonic (at the highest level). See Figure 1 for a pictorial view.

When a user wants to access E2EE files, the user enters their mnemonic into the client. The client then re-derives the master key, uses the master key to decrypt the private RSA key, uses the RSA private key to decrypt the metadata keys, uses the metadata keys to decrypt the file keys, and finally uses the file keys to decrypt E2EE files fetched from the server.

When a folder is shared between multiple users, its metadata key is encrypted for all of those users individually using RSA-OAEP. A user can revoke access to a folder for another user by generating a new metadata key for the folder and encrypting it using RSA-OAEP to the public keys of all the users who should still have access, omitting the public key of the revoked user.

1.3. Contributions

In this work, we conduct a detailed analysis of the cryptographic security of Nextcloud's E2EE feature, assessing the achieved security compared to the advertised guarantees. Our analysis unveiled three distinct vulnerabilities in the E2EE module of Nextcloud. As we will explain, an adversary in control of the Nextcloud server can leverage the presented vulnerabilities to completely break confidentiality and integrity of users' data. We briefly present each attack next.

1.3.1. Key Insertion Attack. The first attack arises from a basic misunderstanding of the security offered by public key encryption (PKE) in tandem with Nextcloud's desire to allow access to files to be granted and possibly later revoked for other users as part of the E2EE feature. The attack exploits the fact that the malicious server can create an RSA-OAEP ciphertext containing a chosen metadata key and place it in the relevant directory of the victim

3. See https://nextcloud.com/secure/.

user; the server can then trigger the client to perform a key rotation operation so that the chosen key is taken into use. Hence, a malicious server can substitute a legitimate metadata key for a folder with one that they chose, encrypt it for the relevant user(s), and thereby make the client encrypt the keys of all future files added to the folder using a metadata key that the server knows. Knowledge of the metadata key also allows the server to trivially insert new files and modify existing ones.

At its core, the attack exploits the fact that PKE provides confidentiality but not data origin authentication: without some form of the latter, the victim user cannot be sure from whence the key came.

1.3.2. Ghost Key Attack. Nextcloud clients fetch encrypted key material from the server. The file structure containing this information is not thoroughly checked by the client. This allows a malicious server to craft a file that tricks clients into accessing the map of metadata keys at an index that is not present in the map. If accessed at an uninitialized index, the map allocates a default all-zero key. The all-zero key is later used by the client to encrypt file keys. This obviously leads to a complete loss of both confidentiality and integrity for user data.

1.3.3. IV Reuse in File Encryption. Files are encrypted using AES-GCM with a random 128-bit key and a random 96-bit initialization vector (IV). However, when a file is modified, it is re-encrypted using the same key and the same IV. It is well known that the security of AES-GCM depends critically on using a fresh IV for each encryption under a given key, with disastrous consequences otherwise. Specifically, because AES-GCM uses a stream cipher mode of AES as its encryption component, a repeated IV leads to a repeated keystream. This can result in a loss of confidentiality. In the context of Nextcloud, we show how to exploit this weakness to recover plaintext files in the situation where the adversary can see encrypted versions of a file and of a modified version of the file (more exactly, a version of the file in which a single character has been inserted). A repeated IV in AES-GCM also enables recovery of the AES-GCM authentication key, making it possible for an attacker to violate integrity [5], [18]. In the context of Nextcloud, this allows the adversary to mount a framing attack in which it injects validly encrypted files into E2EE folders.

1.4. Methodology

Our security analysis was performed in three main steps:

1) *Definition of the Threat Model.* We outlined the threat model and associated security claims by referring to Nextcloud's documentation on the E2EE module [23], [30] and on their advertised claims [27].
2) *System Modeling.* We modeled Nextcloud's E2EE based on their white paper [23], [26] and on the client source code.[4] In particular, only the in-depth analysis of the client source code revealed the ghost key vulnerability and the IV reuse.
3) *Proof-of-Concepts.* We used a self-hosted Nextcloud server to test all attacks against real (self-owned) desktop clients.

1.5. Ethical Considerations

All the experiments and proof of concepts were developed on a self-hosted Nextcloud server under our full control; no other server instance was targeted.

We contacted Nextcloud to inform them of the vulnerabilities in their system and to suggest mitigations on January 2, 2023. We suggested a 90-day disclosure period. Nextcloud acknowledged the attacks on January 12, 2023, confirming that the system is vulnerable and needs patching.

Nextcloud released patches for all three vulnerabilities on March 29, 2023. Each vulnerability was accompanied by a CVE.[5] The first vulnerability was temporarily addressed by disabling the sharing feature and implementing a checksum on the metadata keys [**?**]. This represents a significant downgrade to the E2EE feature set. A significant revision of Nextcloud's cryptographic design is required to fully restore the sharing feature. Nextcloud has announced that it plans to reintroduce the feature with a new version of E2EE. At the time of writing, this has not yet happened. The other two vulnerabilities were fixed by addressing the implementation pitfalls that caused them [**?**], [**?**].

1.6. Related Work

The only previous published work on cryptography in Nextcloud that we are aware of is that of Niehage [35], who analyzed server-side encryption in Nextcloud and discovered four vulnerabilities. The first exploits the lack of authenticity of public keys to break confidentiality, essentially by performing a key substitution attack. The others break file integrity. The vulnerabilities were patched by Nextcloud in version 20 of their server.

As noted above, the systematic study of deployed E2EE cloud storage systems only began recently, with the analysis of MEGA [1], [2], [17]. Backendal, Haller and Paterson [2] performed the first in-depth analysis of MEGA. By leveraging a lack of key separation, poor primitive choices, and a lack of integrity of users' private keys, they managed to completely break both the confidentiality and integrity of MEGA's cloud storage. In a follow-up work, Ryan and Heninger [17] improved the central RSA private-key recovery attack of [2] that required 512 logins (and user interactions) down to just six logins. Albrecht, Haller, Mareková and Paterson [1] showed that the patches to the attacks of [2] deployed by MEGA were ineffective. They presented two distinct attacks each leading to RSA private-key recovery and needing roughly the same number of logins as the original attack in [2].

Our work can be seen as a natural successor to this line of work on MEGA, focussed on the next most prominent E2EE cloud storage provider. The attacks we present on Nextcloud are significantly less complex (both from a technical perspective and in terms of the attack

4. https://github.com/nextcloud/desktop

5. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-28997, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-28998, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-28999.

requirements). But they are as equally devastating when considering the core security properties that Nextcloud's E2EE feature purports to achieve.

Already five years ago, Dalskov and Orlandi [13] found significant and numerous vulnerabilities in the cryptographic design of SpiderOak One, an E2EE backup solution. This work can be seen as an early indicator that implementations of E2EE cloud storage systems could be badly designed. Other prominent E2EE cloud storage solutions include PreVeil [37], Proton Drive [38] and Tresorit [41]. Analyses of these systems may yield interesting results.

One of our attacks exploits the well-known IV (or nonce) re-use vulnerability in AES-GCM. This was first highlighted by Joux [18]. Böck, Zauner, Devlin, Somorovsky and Jovanovic [5] showed how this vulnerability arose in some SSL/TLS implementations and performed Internet-wide scans to identify vulnerable servers. The field of Misuse-Resistant Authenticated Encryption (MRAE), cf. [39], can be seen as attempting to design AE schemes that are more resilient to mishandling of IVs by developers. RSA-OAEP is generally considered to be a secure PKE scheme, with a detailed analysis of its IND-CCA security being available in the literature [16]. However, some early implementations of RSA-OAEP were vulnerable to Manger's attack [19] which exploited analysis of different error conditions that could arise during decryption. Our attack does not rely on any implementation pitfall in RSA-OAEP itself, but rather on a misunderstanding on the part of Nextcloud's developers concerning what security services a PKE scheme can provide.

There is a huge academic literature on cloud storage with E2EE and other, more advanced security features, e.g. user anonymity, metadata hiding, and obliviousness of access and operations. While scientifically sound and technically interesting, this work seems to have had little influence on the *practice* of E2EE for cloud storage, as evinced by the above-mentioned work on MEGA and our work on Nextcloud.

1.7. Paper Structure

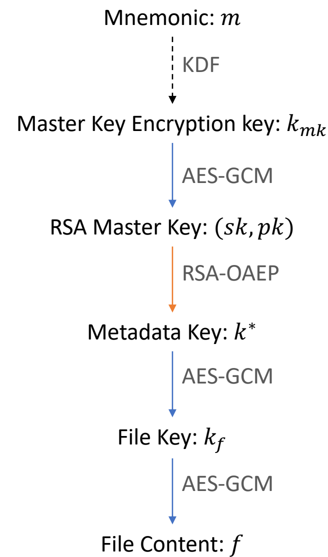
We give a detailed description of cryptographic aspects of Nextcloud's E2EE feature in Section 2. We provide more details on each attack in Section 3. In Section 4, we briefly describe our proof of concept implementations of the three attacks. We discuss mitigations for the attacks in Section 5. We discuss general takeaways for the design of secure cloud storage systems in Section 6. In particular, we believe that the first vulnerability reveals a common misconception PKE schemes, namely that they provide some form of (data origin) authenticity.

2. Background

2.1. Notation

By $[ptxt]_k$ we denote the encryption of a plaintext $ptxt$ with the key k. The encryption algorithm can be derived from the context. We use maps to collect identifier-value pairs. A map T is initialized as $\mathsf{T} \leftarrow \{\}$. $\mathsf{T}.\textsc{put}(id, v)$ represents the insertion of the identifier-value pair (id, v). $\mathsf{T}.\textsc{get}(id)$ returns the value v corresponding to the identifier id, and $\mathsf{T}.\textsc{keys}()$ returns the set of all identifiers for which a value has been inserted. Finally, the Encode/Decode procedures serialize/deserialize a list of variables. In practice, Nextcloud collects the values in a JSON-encoded byte string.

Figure 1: Nextcloud's key hierarchy



2.2. Cryptography in Nextcloud

Key Hierarchy. Figure 1 illustrates the key hierarchy in Nextcloud's E2EE module.

At the bottom of the hierarchy are the file keys. File keys are 128-bit symmetric keys used to encrypt files with AES-GCM, and are freshly generated by the client for every new file that is uploaded.

Files are organized in folders, and every folder is associated with a metadata key k that the client generates by sampling $k \leftarrow\$ \{0, 1\}^{128}$ when a folder is marked as E2EE. Each file key is encrypted separately with the metadata key of the corresponding folder using AES-GCM with a randomly sampled IV.

To enable sharing of encrypted folders, Nextcloud uses public-key encryption (RSA-OAEP) to encrypt the metadata keys. When a user enables E2EE, an RSA master key pair (sk, pk) is generated by their client and the public key is signed by the server to form an X.509 certificate in a public key infrastructure (PKI) in which the Nextcloud server acts as the root certificate authority. The first time a user wants to share a folder with another user, they establish a trust relationship with the recipient by downloading and verifying the certificate for their public key [26]. Following a trust on first use (TOFU) approach, clients store the downloaded certificates locally for future use. We discuss this design choice in Section 3. The metadata key of an E2EE folder is initially encrypted for the owner, and can then additionally be encrypted with the public key of any user to give them shared access to the folder. To allow a user to fetch their keys from any device, the RSA key pair is stored on the server, in partially encrypted form; the public key is stored in the clear, whereas the private

key sk is encrypted using AES-GCM with an encryption key k_{mk} derived from a so-called mnemonic.

The mnemonic is a 12-word long passphrase, which is generated on the client by random sampling from a set of 2048 words. It stands at the top of the key hierarchy and thereby forms the root of security for E2EE data. Note that in other E2EE systems, it is common to have a user-chosen password as the root of the key hierarchy. In Nextcloud, in contrast, passwords are only used to authenticate users, and the mnemonic (for which the client enforces relatively high entropy) is used to achieve confidentiality and integrity. As a consequence, user authentication is kept separate from cryptographic access to E2EE data. This is a laudable design choice which likely improves the E2E security of Nextcloud compared to password-only systems, since users unfortunately often pick weak passwords having much lower entropy than the mnemonics used by Nextcloud.

To summarize, all of the keys in the hierarchy depicted in Figure 1 – except for the mnemonic and the public part of the RSA master key – are stored encrypted on the server to support access from multiple devices. The public key is stored in plaintext on the server, and the mnemonic is the root of the key hierarchy that the user needs to remember or securely store in order to access their files.[6] A user on a new device, after authenticating to the server, can enter the mnemonic, fetch the encrypted key material from the server, decrypt it, and validate it.[7] Once the file keys have been recovered, the encrypted files can be fetched from the server and decrypted.

Folder Metadata. The metadata of an E2EE encrypted folder consists of two maps, $\mathsf{T_k}$ and $\mathsf{T_f}$, containing the metadata keys associated with the folder and the file metadata (i.e. the file keys and file names), respectively.

The metadata key map $\mathsf{T_k}$ maps integers i to metadata keys k. Here, the index i represents the order in which metadata keys were generated. (The initial metadata key has index $i = 0$, the second one index 1, and so on.)

In the file metadata map $\mathsf{T_f}$, a filename fn is mapped to a tuple consisting of the *obfuscated* file name ofn, the file key k_f, the IV iv_f used for the encryption of the file, as well as the resulting message authentication code (MAC) tag τ_f. The obfuscated file name is a random bitstring used to identify an encrypted file on the server.

Together, $\mathsf{T_k}$ and $\mathsf{T_f}$ form the folder metadata, which is stored encrypted on the server. That is, corresponding to each of $\mathsf{T_k}$ and $\mathsf{T_f}$, there is an "encrypted" map $\lceil \mathsf{T_k} \rfloor$ and $\lceil \mathsf{T_f} \rfloor$, which instead maps the identifiers to the *encryption* of the values in the original map. The encryption and decryption procedures for the folder metadata are described later in this section.

6. Note that, as a consequence of the E2EE security guarantees, the server cannot support a user who has lost their mnemonic in recovering their keys.

7. The validation of the RSA key pair is done by retrieving the public key from the server, reconstructing a version of the RSA modulus from the private key, and finally performing a trial encryption and decryption operation. This operation uses the retrieved public key for encryption and the private key and reconstructed modulus during decryption. Because of the integrity provided by AES-GCM and the use of a reconstructed modulus during decryption, this approach appears to prevent key overwriting attacks like those in [6].

Folder Creation. When an E2EE folder is created, the client initializes the folder metadata ($\mathsf{T_k}$, $\mathsf{T_f}$), samples a metadata key $k \leftarrow_\$ \{0,1\}^{128}$ and puts it in the map of metadata keys $\mathsf{T_k}$ via an operation $\mathsf{T_k}.\textsc{put}(0, k)$. The client concludes the creation of the folder by encrypting the folder metadata and uploading ($\lceil \mathsf{T_k} \rfloor$, $\lceil \mathsf{T_f} \rfloor$) to the server. At this point, $\lceil \mathsf{T_k} \rfloor$ contains the encryption of a single metadata key, and $\lceil \mathsf{T_f} \rfloor$ is empty because the folder still does not contain any files.

Folder Synchronization. Nextcloud desktop clients are synchronization clients that keep a local folder synchronized with the one stored remotely on the server. Synchronization can be triggered by various events, including a modification to the local folder, a user-initiated synchronization request, or an automatic activation upon opening the client.

The synchronization process consists of three steps:

1) *Folder Metadata Download.* The client downloads the encrypted folder metadata and decrypts it to retrieve the list of filenames contained in the E2EE folder and the corresponding file keys.
2) *File Synchronization.* The local and remote timestamps of each file are compared to determine which copy (if the file is present both remotely and locally) is newer. Each timestamp reported by the server is associated with the obfuscated file name. If the client determines that the more recent version of a file is the remote one, it requests the encrypted file identified by the obfuscated name ofn from the server. Similarly, new local files are encrypted and uploaded to the server under the name ofn.
3) *Folder Metadata Upload.* At the end of the synchronization, the folder metadata is encrypted and uploaded to the server. The folder metadata is re-uploaded each time as it may have been modified in the previous step. For example, if a new file is added locally, its filename and corresponding file key are added to $\mathsf{T_f}$.

The encryption and decryption procedure of folder metadata, as well as the file encryption procedure, are described next.

Folder Metadata Encryption. The client executes the ENCRYPTFOLDERMETADATA procedure in Figure 2 to encrypt the folder metadata before uploading it to the server.

During the encryption, the metadata keys are encrypted with RSA-OAEP under the user's public key pk and are stored in the map $\lceil \mathsf{T_k} \rfloor$ under their index from $\mathsf{T_k}$. For each file, the protected metadata $prot$, encoding the filename fn and the file key k_f, is encrypted with AES-GCM under the metadata key k^* associated with the highest index i^* in $\mathsf{T_k}$, and a randomly sampled IV iv. In the encrypted file metadata map $\lceil \mathsf{T_f} \rfloor$, the *obfuscated* file name ofn is mapped to $([prot]_{k^*}, \tau, iv, i^*, \tau_f, iv_f)$, where τ, iv and τ_f, iv_f are the tags and IVs resulting from the encryption of $prot$ and of the file itself, respectively.

The inverse of the folder metadata encryption is the DECRYPTFOLDERMETADATA procedure shown in Figure 3. This procedure is executed by the client at the beginning of synchronization to retrieve the file metadata.

```
encryptFolderMetadata(pk, T_k, T_f):
    Given: the user's public key pk, the map of metadata keys T_k,
    and the map of file metadata T_f
    Returns: the map of encrypted metadata keys [T_k], and the
    map of encrypted file metadata [T_f]
1   [T_k] ← {}
2   for i, k ∈ T_k
3     [k]_pk ← RSA.Enc(pk, k)
4     [T_k].put(i, [k]_pk)
5   i* ← max(T_k.keys())
6   k* ← T_k.get(i*)
7   [T_f] ← {}
8   for fn, (ofn, k_f, τ_f, iv_f) ∈ T_f
9     iv ←$ {0,1}^96
10    prot ← Encode((fn, k_f))
11    [prot]_k*, τ ← AES-GCM.Enc(k*, prot, iv, ε)
12    [T_f].put(ofn, ([prot]_k*, τ, iv, i*, τ_f, iv_f))
13  return ([T_k], [T_f])
```

Figure 2: Encryption of folder metadata. Note that file metadata is encrypted with the latest metadata key k^* corresponding to the highest index i^* in $\mathsf{T_k}$.

```
decryptFolderMetadata(sk, [T_k], [T_f]):
    Given: the user's secret key sk, the map of encrypted metadata
    keys [T_k], and the map of encrypted file metadata [T_f]
    Returns: the map of metadata keys T_k, and the map of file
    metadata T_f
1   T_k ← {}
2   for i, [k]_pk ∈ [T_k]
3     k ← RSA.Dec(sk, [k]_pk)
4     T_k.put(i, k)
5   T_f ← {}
6   for ofn, ([prot]_k*, τ, iv, i, τ_f, iv_f) ∈ [T_f]
7     k ← T_k.get(i)
8     if k == ⊥
9       k ← {0}^128
10      T_k.put(i, k)
11    prot ← AES-GCM.Dec(k, [prot]_k*, τ, iv)
12    if prot == ⊥
13      continue  // No error is reported.
14    (fn, k_f) ← Encode(prot)
15    T_f.put(fn, (ofn, k_f, τ_f, iv_f))
16  return (T_k, T_f)
```

Figure 3: Decryption of folder metadata. Note that the map $\mathsf{T_k}$ is accessed at the index i specified in the file metadata. This is relevant for the ghost key attack.

In the procedure, the protected metadata $[prot]_k$ is decrypted using the metadata key corresponding to the index i specified in the tuple (which is not necessarily the highest in $\mathsf{T_k}$). We believe this behavior is to allow clients to rotate the metadata key while postponing the re-encryption of the file metadata to only happen when/if the file itself is modified.

File Encryption. Files are encrypted with the encryptFile procedure in Figure 4. If a new file is created locally, a file key k_f and IV iv_f are sampled randomly and used to encrypt the file content using AES-GCM. According to Nextcloud's white paper and E2EE RFC [23], [26], file keys should be re-sampled each time a file is modified and re-encrypted. However, during our analysis, we found that file keys were only generated when a file is first uploaded. That is, if an already existing file is updated, k_f and iv_f are retrieved from $\mathsf{T_f}$ and used to re-encrypt the file content. At the end of the procedure, $\mathsf{T_f}$ is updated with the key and IV used in the encryption.

```
encryptFile(T_f, fn, f):
    Given: the map of file metadata T_f, the name of the file fn,
    and the file content f
    Returns: the updated map of file metadata T_f, and the encrypted
    file [f]_kf
1   if fn ∈ T_f
2     (ofn, k_f, τ_f, iv_f) ← T_f.get(fn)
3   else
4     k_f ←$ {0,1}^128
5     iv_f ←$ {0,1}^96
6     ofn ←$ {0,1}^128
7   ([f]_kf, τ_f) ← AES-GCM.Enc(k_f, f, iv_f, ε)
8   T_f.put(fn, (ofn, k_f, τ_f, iv_f))
9   return (T_f, [f]_kf)
```

Figure 4: File encryption. Note that if the filename is already in the map $\mathsf{T_f}$, then both the file encryption key and the IV are reused.

```
decryptFile(T_f, ofn, [f]_kf):
    Given: the map of file metadata T_f, the obfuscated name ofn,
    and the encrypted file [f]_kf
    Returns: the filename fn, and the decrypted file f
1   for fn, (õfn, k_f, τ_f, iv_f) ∈ T_f
2     if õfn == ofn
3       (ofn, k_f, τ_f, iv_f) ← T_f.get(fn)
4       f ← AES-GCM.Dec(k_f, [f]_kf, τ_f, iv_f)
5       return (f, fn)
6   return (⊥, ⊥)  // The client reports an error.
```

Figure 5: File decryption. If there is no entry in $\mathsf{T_f}$ for the file identified by *ofn*, then the client reports an error to the user during the synchronization step.

The decryptFile procedure is shown in Figure 5. On input the obfuscated file name *ofn*, it first looks for an entry in the file metadata $\mathsf{T_f}$ which contains *ofn*. If such an entry is found, the associated file key is used to decrypt the file, which is returned together with the (unobfuscated) filename. If no entry contains the obfuscated name, then an error is raised by the client.

Sharing. In Nextcloud, file sharing is implemented on a folder level using the PKI of RSA keys associated to users. To share a folder, the relevant metadata key is encrypted under the RSA public key of the intended recipient. Direct sharing of individual E2EE files is not supported.

As described above, each folder has an associated metadata key map $\mathsf{T_k}$, which may contain multiple metadata keys. According to Nextcloud documentation [23], the reason that each folder can have multiple metadata keys is to allow the removal of users from shared access to the folder. That is, if user A wants to revoke access to a folder from user B, A can add a new metadata key to $\mathsf{T_k}$ and not encrypt the new key with the public key of user B. Since only the highest indexed metadata key k^* in $\mathsf{T_k}$ is used to encrypt the folder metadata when the folder is next synchronized (see the encryptFolderMetadata procedure in Figure 2), user B will no longer be able to decrypt the folder metadata.

Note, however, that since file keys are not updated

when files are re-encrypted (contradicting the specification in Nextcloud's white paper [23]), the removed user could in theory still access the files by storing the individual file keys themselves. That is, the re-encryption of file keys with a new metadata key is insufficient to provide forward security for updated files from removed users.

Finally, we note that, in the Nextcloud client deployed at the time of our analysis (v3.6), the sharing feature is not fully implemented, meaning that folder sharing is actually not possible in practice. However, clients already support multiple metadata keys and key rotation as described above.

3. Attacks

Threat Model. According to Nextcloud's documentation, "end-to-end encryption in Nextcloud protects user data against any attack scenario between user devices, even in case of an undetected, long-term security breach or against untrusted server administrators" [30]. As this quote highlights, the E2EE module is supposed to remain secure even if the server running Nextcloud's protocol is itself compromised or malicious.

In this setting, the adversary has full access to all encrypted data outsourced by the clients to the server, and can also interact with clients via legitimate channels during operations like authentication and file upload. Furthermore, a malicious server – or an adversary with control of the server – can change the behavior of the server and deviate arbitrarily from the normal protocol. Consequently, the security of the system relies entirely on the protective measures and cryptography employed by the clients.

In the following sections, we describe three attacks on the end-to-end security guarantees of Nextcloud in this setting. First, however, we make a short detour to critique Nextcloud's trust assumptions concerning authentication of users' public keys.

Trust Assumptions for Public Key Authentication. While a client can authenticate its own RSA key pair because it is authenticated by the key k_{mk} derived from the user's mnemonic, the authentication of other users' public key relies on a PKI having the server as the root of trust.

Nextcloud employs a TOFU approach to justify the use of an untrusted entity as the root of trust: when a client downloads a certificate from the server, it is stored locally for future use; it is assumed that the server is honest and provides a certificate for the legitimate requested public key at this point. However, this approach leaves much to be desired.

First, given the setting, the server should be considered to be an untrusted entity and so cannot be relied upon as a root of trust. A malicious server can mount certificate substitution attacks that fool clients into accepting the wrong public key for any other client; coupled with the file sharing mechanism described in Section 2.2 this enables trivial attacks allowing the server to recover file contents of E2EE shared folders.

Second, if TOFU is the chosen security model, then exchanging certificates instead of just public keys does not add significant security to the system. This is because an adversary in control of the server would presumably also have access to the server's private key and hence be able to issue fake certificates.

Third, if TOFU is used, it should not be possible for the server to remove already cached keys or certificates on the clients. However, Nextcloud allows servers to issue a so-called "remote wipe" which induces a client to delete all the stored key material (including the certificates). The feature is intended to allow the server to delete important key material from the client in case the device on which it is running is lost or stolen. However, this feature voids the TOFU approach because it allows a compromised server to erase older certificates and substitute them with new ones when the user next logs in.

Nextcloud is aware that TOFU relies on a strong security assumption and aspires to integrate a certificate transparency log and support for hardware security modules [23] in the future. However, these features are not currently implemented.

None of the attacks that we present below exploit weaknesses in the TOFU assumption; they all work even if public keys are properly authenticated.

3.1. Key Insertion Attack

In this attack, the malicious or compromised server exploits the lack of authenticity of encrypted metadata keys to replace an honest metadata key with one generated by and known to the server. This gives the adversary access to all the file keys which are encrypted with the malicious metadata key.

The attack relies on the fact that metadata keys are encrypted under the user's master key using RSA-OAEP [4]. While RSA-OAEP has been proven secure against adaptive chosen-ciperhtext attacks (IND-CCA2) [15] – which also implies that ciphertexts are non-malleable [3] – it does not provide any data origin authentication (or integrity) guarantees for the generated ciphertexts. Consequently, anyone who has access to the public key of a Nextcloud user can generate a valid encryption of a metadata key of their choosing. This allows a malicious server to deceive a client into using a metadata key known to the server, by replacing the highest-indexed metadata key of a folder with the encryption of a chosen key. We provide the pseudocode of the attack in Figure 6.

Once the attack has been executed, the folder metadata contains the encryption of a metadata key $\widehat{k}$ chosen by the adversary which decrypts correctly on the client. Any subsequent synchronization process will then cause the file metadata – including file keys – to be encrypted with the rogue metadata key, as shown in Figure 2.

Note that users' public keys are stored in the clear on the server, and hence known to the adversary, enabling the attack. Additionally, according to the white paper [23] public keys should be auditable. That is, it is explicitly required that the keys can be made public. This means that the attack is in theory also possible by a TLS machine-in-the-middle (MiTM) attacker (i.e. a MiTM attacker that can break TLS), since such an attacker can intercept the client to server traffic and replace the metadata key map with a modified version containing $\widehat{k}$. This highlights the system's reliance on the security provided by TLS. Furthermore, the attack is entirely surreptitious: since RSA-OAEP does not provide authentication, the client cannot distinguish

the maliciously inserted metadata key from a metadata key added by another legitimate client (for example to perform a key rotation).

Key Overwriting Variant. The presented key insertion attack uses the fact that clients support the decryption of protected metadata with different metadata keys, while always encrypting these fields with the metadata key associated with the highest index in the key map $\mathsf{T_k}$. Therefore, the insertion of $\widehat{\mathrm{k}}$ does not disrupt the decryption of file metadata encrypted with other legitimate keys. Rather, only the encryption step at the end of the synchronization process changes, as the client will encrypt all the file metadata with the new $\widehat{\mathrm{k}}$.

Here, we present a variant of the attack which does not rely on clients supporting the use of multiple metadata keys, but still gives the adversary access to files that are added after the attack. We include this variant to demonstrate that the simple fix of removing multiple metadata keys would not be sufficient to address the vulnerability. It works as follows. Instead of inserting $\widehat{\mathrm{k}}$ with identifier $\widehat{i} > i^*$ into $\mathsf{T_k}$, the adversary overwrites the legitimate metadata key at i^* with $\widehat{\mathrm{k}}$. This means that all file metadata of files added to the E2EE folder after the overwriting takes place will be encrypted with the malicious key $\widehat{\mathrm{k}}$. Therefore the adversary can recover the file keys and get full access to any newly added files.

This attack variant is weaker than the original key insertion version for two reasons. First, it does not allow the recovery of files added to the E2EE folder before the adversary was active. Second, it is less stealthy: without support for multiple metadata keys, the client cannot use the previous legitimate metadata keys to decrypt the metadata of files added before the attack. As a result, the decryption of file metadata (and the related files) fails because the client would try to decrypt it using $\widehat{\mathrm{k}}$ instead of the overwritten previous key. This raises an error when procedure DECRYPTFILE in Figure 5 is executed with one of the older files as input, because there is no corresponding entry in the file metadata. (The decryption of the protected metadata on line 11 in Figure 3 will fail, leading to a $\perp$ entry in $\mathsf{T_f}$.)

To avoid raising errors, the adversary can delete the files on the server, causing the files to also be deleted locally. Both options may raise the suspicion of users. However, if the adversary is active when the folder is created and still empty, then the attack stealthiness would be preserved. In conclusion, although weaker, the attack variant highlights the root cause of the vulnerability: PKE-encrypted metadata keys are not authenticated.

Consequences. The consequences of the key insertion attack are severe, as it provides an attacker with complete control over the E2EE folder. The confidentiality and integrity of the folder are entirely compromised, allowing the adversary to access files, modify existing ones, and insert new ones at will – and all of this in a way that is completely undetectable to the client.

The overwriting variant of the attack enables an adversary to gain full control of folders created after the adversary becomes active. Performing the attack on existing, non-empty E2EE folders either leads to errors or

```
KEYINSERTIONATTACK([T_k], pk, k̂):
  Given: the encrypted map of metadata keys [T_k], the victim's
  public key pk, a metadata key chosen by the adversary k̂
  Returns: the map of encrypted metadata keys [T_k]
1 [k̂]_pk ← RSA.ENC(pk, k̂)
2 i* ← MAX([T_k].KEYS())
3 î ← i* + 1 // Key Overwriting: î ← i*
4 [T_k].PUT(i, [k̂]_pk)
5 return [T_k]
```

Figure 6: Key insertion attack. A malicious or compromised server can add the encryption of a rogue metadata key $\widehat{\mathrm{k}}$ to the map of metadata kes.

to content modifications which are detectable to victim. It also does not allow the recovery of the already existing files.

3.2. Ghost Key Attack

In this attack, the adversary exploits two implementation pitfalls to insert an all-zero metadata key at the highest index in the metadata key map $\mathsf{T_k}$ of an E2EE folder. Similar to the key insertion attack, this results in the client using a metadata key which is known to the adversary to encrypt the folder metadata, thereby giving the attacker complete access to the files in the folder.

At the core of the vulnerability is the fact that the metadata key map $\mathsf{T_k}$ allocates a default value – namely, an all-zero entry – when accessed at an index that is not already in the map. Nextcloud maps are implemented using the QMap object [8] from the Qt library [10], and this default allocation is clearly specified in the map documentation [9]. Hence the issue does not stem from a bug in the library. Rather, the problem is that this behavior is not well-suited for use in a security-critical system, unless precautions are taken to avoid it. This leads to the second implementation pitfall: Nextcloud clients do not perform sufficient sanity checks on the inputs provided by the server. In particular, the client does not check for "out-of-bound" indices, and error messages from metadata decryption failures are ignored. Together, this creates sufficient conditions for an attack which we call the "ghost key attack". The attack is shown in Figure 7.

To perform the attack, the adversary first inserts a dummy entry into the encrypted file metadata map $[\mathsf{T_f}]$. The dummy entry is empty, except for where it specifies the index of the metadata key that should be used to decrypt it. There, the adversary chooses $\widehat{i} \leftarrow \max([\mathsf{T_k}].\text{KEYS}()) + 1$, such that the malicious index $\widehat{i}$ is higher than the highest index in the metadata key map $[\mathsf{T_k}]$. As a consequence, the index in the dummy entry points to a non-existing metadata key in $[\mathsf{T_k}]$. When the client performs the next synchronization, it will try to decrypt the file metadata of the dummy entry (as shown in Figure 3), thereby accessing $\mathsf{T_k}$ at the adversarially chosen index $\widehat{i}$. Due to the default behavior of the Qmap object, this creates an entry with the all-zero key $\widehat{k_0} = \{0\}^{128}$ in $\mathsf{T_k}$ at $\widehat{i}$. At the end of the synchronization, $\widehat{i}$ is the highest identifier in $\mathsf{T_k}$ and therefore all file metadata – including all file keys – are re-encrypted with $\widehat{k_0}$ before being re-uploaded to the server.

```
GhostKeyAttack([T_k], [T_f]):
  Given: the encrypted map of file metadata [T_k], the encrypted
  map of metadata keys [T_k]
  Returns: the modified encrypted file metadata [T_k]
1 i* ← MAX([T_k].KEYS())
2 î ← i* + 1 // Key Overwriting: î ← i*
3 [T_f].PUT("dummy", (0, 0, 0, î, 0, 0))
4 return [T_k]
```

Figure 7: Ghost key attack. A malicious or compromised server can add a dummy entry to the file metadata which leads the client to access $\mathsf{T_k}$ at a non-existing index. By default behavior of the map object used, this access creates an all-zero key in $\mathsf{T_k}$ at the index $\widehat{i}$.

This attack can be performed by a malicious or compromised server, or by a TLS MiTM attacker. After the attack has been performed, the adversary has access to the folder metadata encrypted under $\widehat{k_0}$ and therefore full control over the whole E2EE folder.

Attack Stealthiness. As shown in procedure decryptFolderMetadata in Figure 3, no error is raised when decryption of the protected fields fails. Instead, the entry is simply skipped. For this reason, the attacker can set all values in the dummy entry (except $\widehat{i}$) to zero and let the decryption fail. However, it is also possible to be even smarter; since the adversary knows that the client will try to decrypt the protected fields using $\widehat{k_0}$, this decryption failure can be avoided.

Specifically, the adversary can set the protected field of the dummy entry to $[(\widehat{k_f}, \widehat{fn})]_{\widehat{k_0}}$ where $\widehat{k_f}$ and $\widehat{fn}$ are a file key and a filename chosen by the adversary, and additionally insert a dummy file in the remote folder encrypted with $\widehat{k_f}$. As a result, the client would successfully decrypt both the protected fields and the file content of the dummy file. This strategy avoids the decryption failure but forces the adversary to insert a file into the victim's folder. On the one hand, this prevents the decryption failure, and also directly breaks the integrity guarantees of the end-to-end encryption. On the other hand, the new file is detectable to the user and may hence make the attack less stealthy.

Key Overwriting Variant. Similarly to the key insertion attack, the ghost key attack relies on clients supporting multiple metadata keys, but can be modified to work without this feature. If the adversary is active when the folder is first created, then they can substitute the encrypted folder metadata $[\mathsf{T_k}]$ with an empty map. This causes the client to replace the honest metadata key map $\mathsf{T_k}$ with one which contains $(0, \widehat{k_0})$ as the only entry. The replacement will not raise any errors since the legitimate metadata key was never used, and after the replacement the metadata of all files added to the folder will be encrypted with $\widehat{k_0}$.

Performing this attack for an existing folder, rather than a new one, leads to the overwriting of a metadata key that already encrypts some data. This is possible but results in the same limitations as discussed in the overwriting version of the key insertion attack in Section 3.1. That is, the decryption of the existing metadata would fail on the client and additionally the adversary would not be able to recover files that are present in the E2EE folder before the attack was performed.

Consequences. The ghost key attack allows an adversary in control of the server or a TLS MiTM attacker to gain full access to an E2EE folder. Because the adversary knows the metadata key used to encrypt the folder metadata it can access files, modify existing ones, as well as insert new ones at will.

The key overwriting variant of the attack allows the adversary to gain full control over newly created folders. For existing folders, the adversary can gain full access to the files added after the overwriting attack takes place.

Despite the consequences of the ghost key attack being the same as the key insertion attack, we stress that the two vulnerabilities are independent. In the ghost key attack, the key known to the adversary ($\widehat{k_0}$) is generated by the client itself and therefore this attack would work even if metadata keys were authenticated.

3.3. IV Reuse in File Encryption

This vulnerability is caused by the reuse of IVs when E2EE files are updated. As shown in Figure 4, no new IV is sampled when an existing file is re-encrypted; rather, the existing IV is reused. This is problematic, since the encryption scheme used for file encryption in Nextcloud's E2EE module is AES-GCM, and it is a well-known fact that IV reuse in AES-GCM can lead to a complete loss of confidentiality and integrity. Below, we briefly explain why this is the case, as well as how it can be turned into an attack on the E2EE security of Nextcloud.

AES-GCM is an AEAD scheme obtained by combining AES-CTR with a Carter-Wegman MAC. That is, the encryption component of AES-GCM uses a keystream to mask the plaintext using an XOR operation. The keystream is generated by applying AES block cipher encryption to an increasing counter. The counter is initialized by the IV. Reusing the IV leads to a repeated sequence of counters and hence a repeated keystream. Consequently, taking the XOR of two ciphertexts c_1, c_2 created using the same IV will yield the XOR of the corresponding plaintexts $p_1 \oplus p_2$.

The problem of recovering the individual plaintexts p_1 and p_2 given their XOR $p_1 \oplus p_2$ was carefully studied in [20]. There, the authors show how to separate p_1 and p_2 with a high success rate if a suitable language model for the underlying plaintext is available. The core idea of the attack is to use an n-gram model of the language in combination with dynamic programming to approximate the likelihoods of pairs of candidates (p_1, p_2) satisfying the constraint on their XOR.

In the context of Nextcloud, we can mount an even simpler attack breaking confidentiality under a mild assumption on the nature of the files being encrypted. Recall that in Nextcloud, IV reuse occurs when different versions of the same file are encrypted. Imagine a scenario where a user edits a text file, perhaps adding and deleting a few characters on each update. Then many characters in the updated file will appear in shifted positions relative to the original file. This can be used to recover part of the original plaintext.

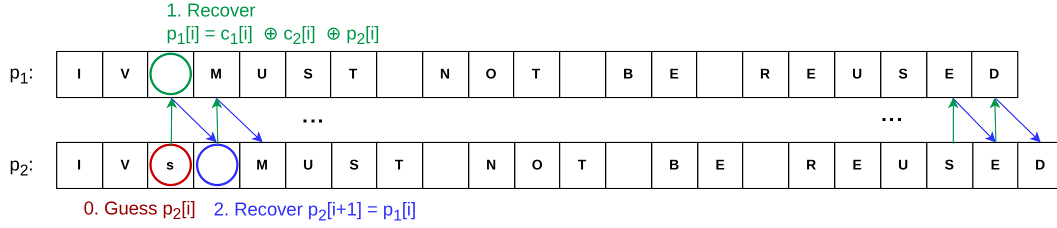


Figure 8: Illustrating the plaintext recovery attack when an IV is reused in AES-GCM and the plaintexts are related by a single character shift.

For concreteness, suppose that a file p_1 is stored encrypted on the server and that the user modifies it by adding exactly one character at an offset i^* from the beginning of the file. As a result, the server learns two ciphertexts, c_1 and c_2: the encryption of p_1 and of the updated file p_2, respectively. By comparing the two ciphertexts, the server can learn the value of i^* (since c_1 and c_2 are identical up to that position).

As shown in Figure 8, the modification causes all the characters after the character in position i^* to shift one position to the right. Consequently,

$$p_1[i] = p_2[i+1] \tag{1}$$

for all indices $i \geq i^*$. For each guess g of the added character, the server does the following (see Figure 8):

1) set $p_2[i^*] = g$.
2) for each $i \geq i^*$, set $p_1[i] = c_1[i] \oplus c_2[i] \oplus p_2[i]$ and $p_2[i+1] = p_1[i]$.

Repeating these steps yields a pair of candidate plaintexts (p_1, p_2) for each guess g of the added character. A language model can then be used to automatically detect which candidate pair contains meaningful text. As a result, the server can recover all the plaintext characters beyond position i^*.

This attack can be generalised to the situation where a larger number of contiguous characters have been added or deleted. This number can be approximated by comparing ciphertext lengths (since AES-GCM ciphertext lengths directly leak plaintext lengths).

This method for "unzipping" the plaintexts when characters have been inserted (or deleted) and an IV reused is not new, but in fact dates back to Tiltman's cryptanalysis of the German Lorenz cipher at Bletchley Park during World War II.[8]

Reuse of the IV in AES-GCM also leads to a break of integrity [18]. In short, with high probability and with low effort, an adversary can recover the AES-GCM authentication key from two ciphertexts if an IV is reused. Given this key, and a portion of keystream recovered using the confidentiality attack given above, an adversary can go on to forge valid AES-GCM ciphertexts for chosen plaintexts. We refer to [18] for further details.

Consequences. An adversary with read access to encrypted files can leverage the IV reuse to mount a plaintext recovery attack on modified files. An adversary that can also modify a user's file storage can then use the attack of [18] to forge validly AES-GCM-encrypted files.

8. See https://billtuttememorial.org.uk/codebreaking/the-tiltman-break/.

4. Proof of Concept Attack Implementations

We implemented all attacks presented in this work and tested them using a self-hosted Nextcloud server instance over which we had full control (as appropriate in the E2EE setting). This allowed us to test the attacks in a controlled environment and verify their effectiveness. The attacks were performed against version 3.6 of the desktop client,[9] this being the latest stable version at the time of our analysis.

The PoCs modified the server behavior by edits to its source code. This is consistent with the setting where the service provider itself is to be considered malicious. Furthermore, since the server code is loaded on each client request, an adversary gaining control over a Nextcloud server can alter its code and behavior in real-time, without needing to restart the server. No changes were made to the client code.

The PoCs for the key insertion attack and the ghost key attack follow exactly the pseudocode in Figure 6 and 7, respectively. To build a PoC exploiting the IV reuse, we considered the simplified case shown in Figure 8 in which a single character is added to or deleted from a text file. Our PoC brute-forces on the deleted character and uses a language model to recover the correct underlying plaintext, as described in Section 3.3. We also implemented Joux's attack [18] to recover the AES-GCM authentication key on IV reuse. Combining the two attacks, our PoC can then both recover the file plaintext and change it to an arbitrary string (of the same length as the original). The source code of the PoCs is publicly available.[10]

5. Mitigations

This section describes the mitigations that we suggested to Nextcloud as part of our disclosure, as well as the measures implemented by Nextcloud to address the vulnerabilities. Nextcloud implemented mitigations for the key insertion attack in version 3.8 of their client and for the other vulnerabilities in version 3.6.5 of their client [?], [?], [?].

5.1. Mitigation of the Key Insertion Attack

As mentioned in the attack description (see Section 3.1), the key insertion attack exploits the fact that metadata keys are encrypted using RSA-OAEP, and hence not

9. https://github.com/nextcloud/desktop/releases/tag/v3.6.0
10. https://anonymous.4open.science/r/nc-poc-release-C86D

```
CHECKSUM(m, k*, [Tf]):
  Given: the user's mnemonic m, the metadata key k*, and the
  map of encrypted file metadata [Tf]
  Returns: the checksum cs
1 buff ← " " // Initialize an empty buffer
2 buff ← buff || m
3 for ofn, _ ∈ [Tf]
4    buff ← buff || ofn
5 buff ← buff || k*
6 cs ← SHA256(buff)
7 return cs
```

Figure 9: Checksum introduced by Nextcloud to authenticate the metadata key k^* and the obfuscated file names.

authenticated. The use of asymmetric cryptography to encrypt metadata keys was introduced to allow a folder-sharing feature.

Suggested Mitigation. In order to achieve authentication of the encrypted metadata keys, while still allowing folder sharing through the use of public-key cryptography, a signcryption scheme can be used to protect metadata keys instead of RSA-OAEP. A secure signcryption scheme provides both confidentiality, unforgeability, and non-repudiation [42]. Confidentiality (which is already achieved with RSA-OAEP) ensures that an adversary without access to the private key is not able to retrieve an encrypted metadata key. Unforgeability provides the additional necessary guarantee of origin authentication for the encrypted key. If metadata keys were signcrypted, each user with access to a shared folder would be able to check who generated the encrypted metadata key, thereby preventing the server from inserting malicious keys.

Implemented Mitigation. Nextcloud's security team decided to introduce a short-term patch in order to prevent the key insertion attack and gain more time to design and deploy a completely new version of the E2EE module with secure file sharing. The patch consists of computing a checksum over part of the folder metadata and the owner's mnemonic with the CHECKSUM procedure in Figure 9. The checksum consists of a SHA-256 hash over the concatenation of the mnemonic m, the obfuscated file names ofn, and the metadata key used to encrypt the file metadata k^*. Since m is only known to the user, the idea is that only the legitimate user should be able to generate cs.

The checksum is computed at the end of the ENCRYPTFOLDERMETADATA in Figure 2 and stored on the server as part of the folder metadata. The DECRYPTFOLDERMETADATA in Figure 3 is modified to recompute the checksum and compare it with the one provided by the server.

Discussion. Although we could not find an attack on this mitigation, we note that hash checksums generally do not provide cryptographic unforgeability guarantees. The correct primitive to use in order to authenticate the metadata based on the mnemonic would have been a MAC. More precisely, a key should be derived from the mnemonic and used as input to a secure MAC to authenticate the encrypted metadata key.

Additionally, authentication using symmetric-key cryptography based on the mnemonic does not follow the requirements posed by Nextcloud for their sharing feature. Because the mnemonic is kept secret by each user, only the user who created the folder metadata is able to authenticate it. As a consequence, the recipient of a shared E2EE folder cannot verify the origin of the folder metadata nor of the folder content. Therefore, if folder sharing was to be enabled together with this temporary patch, the recipient of a shared folder could not authenticate the metadata key, and would hence still be vulnerable to attacks from a malicious server. In conclusion, unless metadata keys can be authenticated by the recipients (for example using signcryption), E2EE folder sharing cannot be implemented securely.

We note that Nextcloud announced in [31] that v3.8 of the Nextcloud client introduces folder sharing for E2EE folders. However, the feature is still not functional in the current version (v3.10). In particular, when attempting to share a folder, clients do not encrypt the metadata key with the public key of the recipient. Hence the recipient is not able to decrypt the metadata of the shared folder, and therefore does not gain access to any files. No additional mechanisms have been introduced in this version of the client to allow recipients to authenticate the folder metadata without knowledge of the sharer's mnemonic.

Hence, the introduced mitigation is merely a patch to the key insertion attack. On its own, it is not sufficient to enable secure file sharing which is the reason why PKE was introduced in the first place.

5.2. Mitigation of the Ghost Key Attack

This vulnerability stems from the fact that the map $\mathsf{T_k}$ used by the client to store metadata keys allocates a default metadata key of all zeros if accessed at an uninitialized index.

Suggested Mitigation. The attack can be easily prevented by having the client verify that the map entry has been initialized before accessing the map at a specified identifier. In general, when developing end-to-end encryption systems in a malicious server threat model, the inputs provided by the server should not be trusted and hence always checked by the client.

Implemented Mitigation. As suggested, additional checks on the server input were introduced. After decrypting $[\mathsf{T_k}]$ in Figure 3, the client checks that the decrypted map $\mathsf{T_k}$ is not empty. Additionally, in the DECRYPTFOLDERMETADATA procedure, all files are decrypted with the metadata key at the highest initialized index in $\mathsf{T_k}$, rather than with the key indicated by the file metadata. This prevents maliciously added file metadata entries from causing the client to access the map at an uninitialized index. Together with the fact that the client checks that $\mathsf{T_k}$ is not empty, this ensures that the map is only accessed at valid indices.

After the mitigation, clients still support metadata key rotation. However, they no longer support file keys for different files being encrypted with different metadata keys.

Discussion. Note that none of the mitigations (suggested or implemented) for the key insertion attack help prevent the ghost key attack. The mitigations against the key insertion attack aim to authenticate metadata keys, thereby preventing an adversary (such as a MiTM attacker or a malicious or compromised server) from modifying or adding a new metadata key to the folder metadata. However, in the ghost key attack, the all-zero key is inserted by the client itself and not by the adversary. Even if metadata keys were authenticated, the verification of their authenticity would occur during decryption in line 3 of Figure 3. Once the metadata keys are decrypted and put into the key map T_{k}, the client trusts that the map only contains authentic keys. However, in the ghost key attack, the client inserts the all-zero key directly into T_{k}, thereby circumventing any authenticity checks.

5.3. Mitigation of IV Reuse

The attack can be easily prevented by always sampling the IV used for file encryption uniformly at random, both when encrypting new files and when re-encrypting modified files.

Nextcloud patched this vulnerability by generating a fresh random IV as well as a new file key for each file encryption.

Note that to prevent our attack it would have been sufficient to just re-sample the IV. Nextcloud chose to re-sample both the IV and the file key, such that (in a future where folder sharing is actually possible) security for updated files is ensured against a removed user. That is, since updated files are encrypted with new file keys, a user who has their access to a shared folder revoked does not learn the new file keys, and hence cannot decrypt the modified files.

6. Conclusion

We analyzed the E2EE features offered by Nextcloud and found three vulnerabilities; each leading to devastating attacks that completely break the confidentiality and integrity of E2EE files. Moreover, we can draw the following broader lessons.

Treat your own code and infrastructure as adversarial. While the ghost key attack relies on an odd behavior of the object used to store metadata keys, the root cause of this vulnerability is more fundamental. Here, had the server inputs been checked before use by the client, the attack would have been prevented. This points to the lesson that, in the E2EE setting where the servers should be considered adversarial, developers need to distrust all server actions and all server-generated inputs. This mindset might be difficult to adopt because it requires developers to produce client code that distrusts server code that they themselves may have written.

Do not beta test security-critical features. Releasing features early, as beta, or as minimum viable products (MVPs), may help attract users and gather support from the developer community. The latter is particularly valuable in an open-source project such as Nextcloud. However, prematurely deploying security features that are only partially implemented can introduce vulnerabilities. For example, deploying a server-rooted PKI while leaving the development of a certificate transparency log as future work undermines the basic security assumptions on which the system relies. Moreover, committing to insecure design choices can lead to an over-complicated solution later on and may necessitate complex patches when vulnerabilities are found. In an extreme case, as with Nextcloud's file sharing feature, an important functionality had to be disabled altogether until a better solution can be developed.

Secure primitives do not imply secure systems. While the adoption of secure primitives and carefully audited implementations is picking up pace in real-world deployments such as Nextcloud, this does not suffice if the chosen primitives do not meet the requirements of the application or are composed incorrectly. For example, our key insertion attack exploits the mismatch between what the designers presumably expected from the primitive (data origin authentication and confidentiality), and what the primitive actually offers (confidentiality). Generalizing from this example, designers need to understand the security guarantees offered by a primitive, and understand how to properly combine primitives in order to achieve more complex security properties. Similarly, developers need to know under what assumption(s) these security guarantees hold. This would have prevented the IV reuse vulnerability in file encryption.

Do not "Design→Release→Break→Patch". Systems should be designed using a proactive approach to security, cf. [36]. This alternative approach requires designers to first produce formal security models capturing the security goals and adversarial capabilities, then specify the system in full, and finally develop security proofs showing that the system meets its goals under well-defined assumptions concerning its cryptographic components. Only once this phase is complete, developers can start their work. Of course, this approach still leaves a gap between formal specification and implementation, but this approach prevents specification-level flaws. Pursuing it is also considerably more involved and less agile than the current practice where design and development go hand-in-hand, and where there is often no separation between designers and developers. It requires highly specialist knowledge on the part of the designers, or the recruitment of applied cryptographers to the design team.

Future Work. Our work on Nextcloud and the recent work on MEGA [1], [2], [17] have shown that well-established companies and open-source projects with millions of users struggle to provide their users with the intended security guarantees. At the same time, the cryptographic community has managed to model and prove the security of important protocols for client-server communication (e.g. TLS1.3 [11], [12], [14]) and E2EE messaging (e.g. Signal [7]). An important goal for future work would be to reach a similar state for E2EE cloud storage. This would be an ambitious and complicated goal to attain, even without the more advanced security properties typically targeted in the literature today. However the result – when adopted by vendors – would greatly improve the security of cloud

storage systems compared to the observed state in today's deployed systems.

Acknowledgments

We thank Nextcloud staff for their helpful cooperation during the disclosure process.

References

[1] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. Caveat implementor! Key recovery attacks on MEGA. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 190–218. Springer, Heidelberg, April 2023.

[2] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: malleable encryption goes awry. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 146–163. IEEE, 2023.

[3] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45. Springer, Heidelberg, August 1998.

[4] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111. Springer, Heidelberg, May 1995.

[5] Hanno Böck, Aaron Zauner, Sean Devlin, Juraj Somorovsky, and Philipp Jovanovic. Nonce-Disrespecting adversaries: Practical forgery attacks on GCM in TLS. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association.

[6] Lara Bruseghini, Daniel Huigens, and Kenneth G. Paterson. Victory by KO: Attacking OpenPGP using key overwriting. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 411–423. ACM Press, November 2022.

[7] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33(4):1914–1983, October 2020.

[8] The Qt Company. Qmap class. https://doc.qt.io/qt-6/qmap.html.

[9] The Qt Company. Qmap class, access operator. https://doc.qt.io/qt-6/qmap.html#operator-5b-5d.

[10] The Qt Company. Qt documentation. https://doc.qt.io/qt-6/.

[11] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1773–1788. ACM Press, October / November 2017.

[12] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In *2016 IEEE Symposium on Security and Privacy*, pages 470–485. IEEE Computer Society Press, May 2016.

[13] Anders P. K. Dalskov and Claudio Orlandi. Can you trust your encrypted cloud?: An assessment of SpiderOakONE's security. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 343–355. ACM Press, April 2018.

[14] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *Journal of Cryptology*, 34(4):37, October 2021.

[15] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 260–274. Springer, Heidelberg, August 2001.

[16] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. *Journal of Cryptology*, 17(2):81–104, March 2004.

[17] Nadia Heninger and Keegan Ryan. The hidden number problem with small unknown multipliers: Cryptanalyzing MEGA in six queries and other applications. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 147–176. Springer, Heidelberg, May 2023.

[18] Antoine Joux. Authentication failures in NIST version of GCM. *NIST Comment*, page 3, 2006.

[19] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 230–238. Springer, Heidelberg, August 2001.

[20] Joshua Mason, Kathryn Watkins, Jason Eisner, and Adam Stubblefield. A natural language approach to automated cryptanalysis of two-time pads. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *ACM CCS 2006*, pages 235–244. ACM Press, October / November 2006.

[21] Mega. Online privacy for everyone, 2023. https://mega.io/.

[22] Steve Morgan. Data attack surface report. cybersecurity ventures, 2020.

[23] Nextcloud. Security and authentication. https://nextcloud.com/blog/whitepapers/security/, Last accessed on 2022-11-30.

[24] Nextcloud. German federal administration relies on nextcloud as a secure file exchange solution, 2018. https://nextcloud.com/blog/german-federal-administration-relies-on-nextcloud-as-a-secure-file-exchange-solution/.

[25] Nextcloud. Nextcloud grew customer base 7x, added over 6.6 million lines of code and doubled its team in 2017, 2018. https://nextcloud.com/blog/nextcloud-grew-customer-base-7x-added-over-6-6-million-lines-of-code-and-doubled-its-team-in-2017/.

[26] Nextcloud. End-to-end encryption RFC, 2021. https://github.com/nextcloud/end_to_end_encryption_rfc/tree/master.

[27] Nextcloud. Encryption and hardening, 2022. https://nextcloud.com/encryption/.

[28] Nextcloud. Nextcloud about page, 2022. https://nextcloud.com/about.

[29] Nextcloud. Nextcloud ceo kicks off nextcloud conference with keynote speech, 2022. https://nextcloud.com/blog/nextcloud-ceo-kicks-off-nextcloud-conference-with-keynote-speech/.

[30] Nextcloud. Nextcloud threat model, 2022. https://nextcloud.com/security/threat-model/, Last accessed on 2022-09-12.

[31] Nextcloud. Desktop 3.8: End-to-End Encryption levels up with sharing and file-drop, 2023. https://nextcloud.com/blog/desktop-3-8-end-to-end-encryption-levels-up-with-sharing-and-file-drop/.

[32] Nextcloud. Desktop clients misbehaves with end-to-end encryption when the server returns an empty list of metadata keys, April 2023. URL removed to preserve author anonymity.

[33] Nextcloud. Initialization vector reuse in end-to-end encryption allows a malicious server admin to break manipulate and access files, April 2023. URL removed to preserve author anonymity.

[34] Nextcloud. Lack of authenticity of metadata keys allows a malicious server to gain access to E2EE folders, April 2023. URL removed to preserve author anonymity.

[35] Kevin "Kenny" Niehage. Cryptographic vulnerabilities and other shortcomings of the Nextcloud server side encryption as implemented by the default encryption module. Cryptology ePrint Archive, Report 2020/1439, 2020. https://eprint.iacr.org/2020/1439.

[36] Kenneth G. Paterson and Thyla van der Merwe. Reactive and proactive standardisation of TLS. In Lidong Chen, David A. McGrew, and Chris J. Mitchell, editors, *Security Standardisation Research - Third International Conference, SSR 2016, Gaithersburg, MD, USA, December 5-6, 2016, Proceedings*, volume 10074 of *Lecture Notes in Computer Science*, pages 160–186. Springer, 2016.

[37] PreVeil. Preveil security and design a description of the Preveil system architecture, 2023. https://www.preveil.com/wp-content/uploads/2019/10/PreVeil_Security_Whitepaper-v1.5.pdf.

[38] Proton. Secure cloud storage and file sharing, 2023. https://proton.me/drive.

[39] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 373–390. Springer, Heidelberg, May / June 2006.

[40] WisCase Cyber Research Team. Over 80 us municipalities' sensitive information, including resident's personal data, left vulnerable in massive data breach. Trend Micro Research, July 20, 2021. https://www.wizcase.com/blog/us-municipality-breach-report/, Last accessed on 2023-05-05.

[41] Tresorit. Tresorit encryption whitepaper, 2023. https://cdn.tresorit.com/202208011608/tresorit-encryption-whitepaper.pdf.

[42] Yuliang Zheng. Digital signcryption or how to achieve cost(signature & encryption) $\ll$ cost(signature) + cost(encryption). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 165–179. Springer, Heidelberg, August 1997.