OmniFileStore Encryption

Design Review

Prepared for: Ken Case
Chief Executive Officer

Omni Development Inc 1000
Dexter Ave N,
Seattle, WA 98109

June 7th, 2016

Disclaimer

No trademark, copyright, or patent licenses are expressly or implicitly granted (herein) with this analysis, report, or white paper. All brand names and product names used in this document are trademarks, registered trademarks, or trade names of their respective holders. Leviathan Security Group is not associated with any other vendors or products mentioned in this document.

Confidentiality Notice

This document contains information confidential and proprietary to Leviathan Security Group. The information may not be used, disclosed, or reproduced without the prior written authorization of Leviathan and those so authorized may only use the information for the purpose of evaluation consistent with authorization. Reproduction of any section of this document must include this legend.

Table of Contents

Engagement Summary 4

The Product 5

Design Review 6

- Threat Model 6
- Entropy Sources 7
- Choice of Algorithms 8
- CTR Mode 9
- Crypto Agility 10
- Compression-Related Attacks 11
- File Length Disclosure 11
- Unauthorized File Overwrite 12
- Key Rollover and Forward Secrecy 12
- Key Encryption Key Protection 13

Summary 14

1 Engagement Summary

The Omni Group publishes a cloud document storage system for the Mac OS X and iOS platforms. Recognizing that it is beneficial to provide encryption for stored client documents, they have designed a cryptosystem tailored to their particular application. Among other constraints, the system is designed such that the client controls the encryption key and that key is unknown to The Omni Group.

On May 17th, The Omni Group engaged Leviathan Security Group to conduct a design-level review of the proposed cryptosystem (OmniFileStore) to be used in the OmniFocus product. Omni provided documentation of the proposed design for the OmniFileStore in the form of diagrams, narrative descriptions with justification and future development plans, and example Python source code, current as of May 18th, 2016. Using this information, Leviathan has documented a number of requirements and considerations related to the use of this design in the target platform.

2 The Product

The OmniFileStore encryption system is intended to be used in a variety of products, including OmniFocus, OmniPresence, MailDrop, and Quick Entry. These products use a document storage system involving a collection of a compacted database containing many files, and individual files not yet added to the database. Associated with these are several metadata files describing key information and synchronization state. The file database is synchronized *en bloc* to each of a user's devices via The Omni Group's application software, which authenticates to and communicates with a hosted web service using TLS.

The role of the encryption system in The Omni Group's product offering is to supplement extant authentication in the protection of a user's files. The current implementation solely relies on transport encryption to protect information during transmission. Once uploaded, a user's files are stored unencrypted and are susceptible to compromise. The Omni Group proposes to introduce a user derived and controlled encryption key to protect the user's files during transport and storage.

Users must have access to their files from multiple devices that can be added and removed dynamically. Multiple device support requires that encryption and decryption keys are available to the user when they use any device to access the service, even if they do not have access to any other device they have previously used. Further, The Omni Group finds it desirable from a performance perspective to have paged random access into files and compressed databases.

As with the vast majority of cryptographically secured document storage systems, the fundamental goal of the system presently under consideration is to prevent disclosure of sensitive information to anyone who does not possess the decryption key. In this case, the contents of any and all documents stored in the system, as well as the titles and other similar metadata of those files, is classified as sensitive. The length of files and other data entities are also considered sensitive if and only if the length would disclose non-trivial information about the data entity.

The Omni Group's products are developed for Mac OS X and Apple iOS platforms. Because of the design goal that the user retains control over the encryption keys they use, cryptographic operations including encryption, decryption, and key generation will be handled on such devices. Where possible, The Omni Group has made use of platform APIs and libraries to handle cryptography in order to minimize opportunities for implementation error.

3 Design Review

Our analysis of the design documents provided is organized by topical area. For each area, we consider how the proposed system supports the security of the proposed system with respect to the design goals. We follow our standard methodology that first considers the *Threat Model* the design faces and then evaluates *Entropy Sources*, *Algorithm Choice*, and *Cypher Modality*.

3.1.1 Threat Model

The threat model used to guide the design of the cryptosystem correctly assumes compromise of the synchronization service. This is vital, since the primary purpose of the cryptosystem is to protect user data even in the event of a compromise.

A compromise that results in disclosure of cleartext user data would have strongly negative external consequences to The Omni Group, including reputational damage and a decrease in users' perceived value of The Omni Group's products and services. Further, storing cleartext user data increases the risk of compromise of that data during the company's day-to-day operations, which creates a need for specific policies and processes to mitigate internal threats.

The impact of cleartext user-data disclosure is further compounded by the fact that the nature of the data is entirely undefined. It is possible and even likely that many different types of sensitive information will be stored on The Omni Group's systems, including *inter alia* personally identifiable information, personal health information, financial and payment industry information, privileged legal information, and intellectual property owned by users. There is no effective technical measure by which The Omni Group could prevent such use, and so it should be taken into account in the threat model.

In their threat model, The Omni Group specifically identifies a number of aspects of their service's attack surface, under the assumption that a user's account is compromised and a hypothetical attacker is able to disclose arbitrary data associated with at least that user's account from The Omni Group:

- Disclosure of the frequency of changes to documents, the size of those changes, and the rough time of the changes
- The ability of an attacker to introduce additional data into a user's working set for later coalescing into their main document database, and the attendant plaintext-guessing attack resultant from compression use
- Possession of one of the devices used to access files, whether a current or old device, and full control over that device
- Potential abuse of certain privileged user accounts (e.g. administrative accounts) to roll back files to previous versions
- The potential for users with write-only access to a user's working set of documents to abuse this privilege to make changes to existing documents by overwriting them
- Disclosure of metadata related to the surrogate filename established when a file is added to a user's repository (e.g. to their database)

While not specifically identified in the threat model section of the provided documentation, Leviathan notes that the design considers the following additional aspects of the attack surface:

- Disclosure of cleartext due to IV-keypair reuse in CTR mode, and other block cipher mode related attacks

- The existence of unencrypted files in a user's repository which were introduced by an entity not privy to the user's encryption key, which remain unencrypted until interacted with by the user
- Disclosure of sensitive information from files which are not encrypted as a matter of policy, including a file storing usage metrics and device synchronization state (.client files), and incoming files from products like Mail Drop (.inbox files)
- Compromise of the user's password, and the attendant disclosure of current and previous file encryption keys

Each aspect of the attack surface identified in the threat model must be specifically addressed by the design in order to mitigate threats against the security of user data stored in The Omni Group's systems.

3.1.2 Entropy Sources

A source of strong entropy is an absolute requirement for the generation of key material and for the use of certain cryptographic algorithms, including CTR mode that is chosen for encryption in the present system. Because the cryptosystem will be implemented by client software running on Mac OS X and various versions of iOS, the chosen mechanism for entropy generation must be available on these platforms.

The implementation particulars of OmniFileStore encryption dictate that a weak or predictable key generated on one device will likely be used to encrypt or re-encrypt the user's data in totality, such that the security of keys generated on the system with the weakest entropy source will be the security of the system as a whole as applied to that user's data.

Because the supported platforms are closed-source, it is necessary to rely on vendor documentation to validate entropy assertions. Unfortunately, very little public documentation has been made available by Apple for this purpose. It is, however, known that Mac OS X uses a Yarrow PRNG to produce the platform-provided random data stream exposed via /dev/random, and that it is generally seeded from the timing of I/O and user interaction events.

Apple's implementation of Yarrow provides 192 bits of internal state, which provides a 160-bit absolute ceiling on the entropy of random numbers generated by the system until additional state information is added to the PRNG. Moreover, the PRNG must be seeded with some initial value for its output to be unpredictable.

We refer to Apple's developer documentation for information on how to best handle secure random number generation on their platform.[1]

On Mac OS X, seeding and entropy addition are both handled by the *SecurityServer* service, and as long as this service is running, it is likely that the random number generator has some entropy available.

On iOS versions 2 and greater, the SecRandomCopyBytes API function is provided as a wrapper for reading bytes from the platform PRNG /dev/random. The algorithm and initial entropy sources used have changed from version to version but are not documented.

While it is possible that the random number generator implementations in iOS and Mac OS X are in fact suitable for key generation, it is very difficult to make any assertion of this from the limited documentation available. However, it is possible to increase the security of the system to an

---

[1] https://developer.apple.com/library/ios/documentation/Security/Conceptual/cryptoservices/RandomNumberGenerationAPIs/RandomNumberGenerationAPIs.html

acceptable level by providing a means to assure the availability of entropy to user devices. This would avoid problems like predictable keys being generated shortly after device booting, but would do little to protect against weak PRNG attacks whereby an attacker is able to use previous PRNG output to predict future PRNG output.

Entropy could be added by writing it to /dev/random, which is the mechanism used by the SecurityServer service to add additional state to the PRNG. Such additional entropy could be sourced from an API provided by The Omni Group. Naturally, it is worthwhile to protect such random numbers, and this is accomplished to a certain extent by the fact that provision of random data from userspace to the kernel does not overwrite PRNG state but is combined with it, so that the new state depends on both the previous state and the new input; additional security could be provided by decrypting the random number issued by the hypothetical API with the user's passphrase-derived key, so that the actual value is unknown to anyone not in possession of the user's passphrase. Such a method would prevent cold boot style attacks and very long running output analysis attacks on the device PRNG intended to limit the selection of candidate keys used to encrypt user data.

Because Yarrow-160 is a 160-bit algorithm, it is not suitable for generating 192-bit or 256-bit keys without reseeding; this constraint should be borne in mind if in fact key sizes larger than 128 bits are used.

It is worth mention that very little specific documentation is available on the way OS X and iOS generate random numbers; information in http://opensource.apple.com shows the yarrow implementation used through OS X 10.9, but as of OS X 10.10 it has been refactored elsewhere and it is no longer clear that yarrow continues to be used.

Leviathan notes with concern that Yarrow is composed of SHA-1 and 3DES; though three-key 3DES used by Yarrow-160 is currently deemed acceptable by NIST, SHA-1 is currently deprecated in SP 800-131A Rev 1.[2] Further, Yarrow-160 is not one of the NIST-recommended PRNG algorithms.[3] Leviathan also notes that Yarrow is deprecated by its authors.[4] Nonetheless, no current attacks relevant to the security of these algorithms as used in Yarrow-160 are publicly known.

Current NIST recommendations allow data encryption and key wrapping keys to be used to encrypt new data for up to two years, and do not require them to be completely retired until 3 years.[5] In keeping with this, it is not necessary to roll keys over any more frequently than recommended; this affords The Omni Group an opportunity to wait for users to use a Mac OS X device where relatively more entropy is available and thereby avoid generating long-lived encryption keys on iOS devices.

3.1.3 Choice of Algorithms

The design document indicates the following algorithms chosen for OmniFileStore cryptography:

- Key encryption key (KEK) derivation: PBKDF2
- Key encryption: AES-128 with AESWRAP
- Data encryption: AES-128 with CTR
- Data integrity: AES-CTR-SHA-256-HMAC
- Deterministic random bit generator (DRBG): Yarrow-160

---

[2] http://dx.doi.org/10.6028/NIST.SP.800-131Ar1
[3] http://dx.doi.org/10.6028/NIST.SP.800-90Ar1
[4] https://www.schneier.com/academic/yarrow/
[5] http://dx.doi.org/10.6028/NIST.SP.800-57pt1r4

The KEK is distinguished from the data encryption key (DEK) so that it is possible to periodically change the DEK without the user needing to select a new passphrase, and so that it is possible to change the user's passphrase without needing to re-encrypt all that user's stored data. The present design supports those objectives.

All of the algorithms chosen, with the exception of Yarrow-160, are widely recognized as being adequate for their respective purpose, including by NIST standards, *ibid*. The previous discussion of entropy sources proposes a method to mitigate the total entropy weakness of Yarrow-160.

The choice of AESWRAP for key encryption is more appropriate than AES-128-CTR, because it obviates the need for IV choice, is designed specifically for key encryption, and has platform support. In our opinion, AESWRAP actually decreases implementation complexity and risk, despite adding another cryptographic primitive to the design.

PBKDF2 use as a key derivation function (KDF) is discussed in NIST SP 800-132.[6] It is a relatively old key derivation algorithm, but nonetheless currently recommended. The NIST guidelines lead Leviathan to recommend that in this case, the salt length used for PBKDF2 be not less than 128 bits, and salts not be shared between users. Additionally, the same recommendations lead us to recommend The Omni Group use as many rounds as consistent with performance on supported devices, but in no case less than 1,000. The choice of salt length avoids limiting the global keyspace even if users choose passphrases having less than 128 bits of total entropy. The uniqueness of the salt and a high number of rounds together increase the cost to attackers of generating key search tables.

A practical attack on a cryptosystem using a KDF to generate KEKs from passphrases is to begin with a list of likely passphrases, generate keys using the KDF and known parameters (salt, rounds, and count of output bytes), and test each of these keys. If used as we describe, the burden to an attacker for generating such keys must be borne for each user to be attacked, and requires more CPU time as the number of PBKDF2 rounds increases. Such an attack is essentially a keyspace limitation attack.

Though AES-256 is available and provides greater security than AES-128, the effective security of the system is limited by the user's selected passphrase which is in any case likely to have less than 128 bits of entropy, much less 256 bits. Considering this and the insufficiency of the platform PRNG for generating 256-bit keys, 128 bit keys would appear to be the appropriate choice.

Separation of the MAC and DEK keys is appropriate in this instance, and the choice of HMAC algorithm is appropriate in that it is at least as strong as the other algorithms used. Applying HMAC to encrypted data is the most optimal design choice in this case and provides adequate data integrity verifiability, since it is applied to both the segment data and IV.[7]

3.1.4 CTR Mode

Through the use of CTR mode, a block cipher can be used to encipher a stream. CTR mode also supports random access, because the key for one block does not depend on the previous block, but this capability is not often expressed by implementations designed to encrypt streams or files.

CTR mode also depends on a unique IV being chosen; the IV combined with a counter is enciphered with the key, and this value is XORed with the plaintext to produce ciphertext. If a single IV-key pair is used to encrypt different data, the security of the system is weakened; for instance, if an attacker can guess the contents of one such encrypted block or part thereof, they can recover the key or part thereof used to encipher the other encrypted block. For this reason, it is absolutely vital that IVs are

---

[6] http://dx.doi.org/10.6028/NIST.SP.800-132

[7] Krawczyk, H. (2001). The order of encryption and authentication for protecting communications (or: How secure is SSL?). *21st Annual International Cryptology Conference*, 310-331. doi:10.1007/3-540-44647-8_19

unique; they should be generated in a way that assures their uniqueness with a high degree of probability and an attacker should not be able to manipulate them to cause a collision.

The easiest way to enforce IV uniqueness is to generate them randomly. The platform PRNG provides sufficient entropy for this purpose, if it is in a seeded state.

The IV used in CTR mode is generally only part of a block size, since the counter is concatenated to the IV. In practice, it is common to use a completely random IV, which effectively initializes the counter to a nonzero value. Because of the design of CTR mode, the IV-counter pair used for each block must be unique to avoid the aforementioned plaintext recovery attack, and so it is necessary to ensure not only that IVs are unique but also that the incrementing of the counter does not cause collisions. This limits the amount of data that can be encrypted with CTR mode. For example, a 64GB file encrypted with AES-128-CTR will consume 2^{32} IV-counter tuples (because a 64GB file contains that many 128-bit blocks), leaving 96 bits of the IV for random selection. Assuming 64GB files, the odds of collision of randomly-selected IVs assuming uniform distribution are less than 10^{28} per IV; the convention of choosing IVs at random therefore provides adequate security against collisions.

The above calculations are consistent with The Omni Group's design choice of a 32-bit counter and a 96-bit IV. Though it is extremely unlikely that any chunk will approach 64GB in size (because the intention in breaking the files into chunks is to allow paged random access), the security gain from a more random IV does not likely offset the cost of deviating from the usual IV length of 12 bytes; the uniqueness of IVs is reasonably assured under this scheme.

CTR mode is therefore appropriate for the purpose in which it is used, assuming it is implemented correctly and adequate entropy is available and used at the time of IV generation.

Splitting files into chunks (segments) to be encrypted separately is not theoretically necessary with CTR mode since it provides random access, but doing so will not decrease security (provided IV uniqueness is maintained), and appears to offer some benefit through the reduction of implementation complexity. However, if a new IV is not chosen for each successive chunk, the security of the encryption will be broken.

Another constraint posed by the design of CTR mode is that storing the DEK-encrypted IV-counter tuple will allow the block indexed by that counter to be decrypted, since CTR mode decryption operates by XOR of the ciphertext with the result of AES block encryption of the IV-counter tuple with the DEK. Because this system avoids encrypting the IV, it is not vulnerable to this type of attack.

3.1.5 Crypto Agility

It is critical that cryptosystems account for changes in algorithm, since the security of an algorithm is dependent on the state of cryptanalysis work on that algorithm. Further, new algorithms and advances in computing power are constantly created. Crypto agility is the ability to change the algorithms used by a cryptosystem. Many factors determine crypto agility, but in general, factors that make algorithm changes difficult tend to decrease it, and factors that make algorithm changes simple tend to increase it. For example, hardcoding cryptographic algorithms coupled with long support cycles decrease crypto agility, whereas protocol design with pluggable cryptographic algorithms increase it.

The use of an algorithm field in the description of the document key management file provides most of the scaffolding required to support crypto agility. Provided that the layout of document encryption headers (particularly IV and MAC length) are dependent on the algorithm choice, it will be possible to change algorithms when necessary. Other considerations are the update cycle of the application, and the ease of re-encrypting data with a new algorithm when one is introduced.

3.1.6 Compression-Related Attacks

Data disclosure using compression-assisted plaintext guessing is a relatively common attack on cryptosystems. Besides the very well-known CRIME attack, the BREACH attack is a good example of this threat.[8] The essential requirements for exploitation of such a data disclosure are the ability to inject arbitrary data pre-encryption, and the ability to observe the effect of that data injection on total encrypted blob size.

This type of attack only presents a threat to the security of OmniFileStore encryption insofar as these requirements can be satisfied by the attacker. However, a defense in depth model requires The Omni Group to assume that an attacker has the ability to download a client's encrypted blob and thereby discover the size. Whether data injection is possible is highly use-dependent, but in all cases would require the user to actively incorporate malicious data into their document store while other aspects of the system are compromised such that the attacker has real-time document database size information. The user would have to be an active (though unwitting) participant in the attack. An attacker with the ability to bypass this user interaction would almost certainly have access to the key encryption key as well, since the most likely mechanism for such automation would be access to the user's device.

An attack such as this is made more difficult by churn in the size of the database; even if that churn is completely random, the burden on the attacker is increased because it becomes necessary to gather more samples to reduce out the randomness. Further, the attacker has a very low degree of certainty as to when their malicious data is incorporated and when it is not.

For these reasons, the degree of threat posed by compression-related attacks is minimal for OmniFileStore. While it would be preferable to avoid compressing data pre-encryption, it may not be practical, and the cost is likely not justified by the risk posed by such attacks.

Other partial mitigations are possible; for instance, a random-length uncompressible blob could be added on each synchronization. This would provide a weak mitigation to this type of attack by forcing the attacker to gather more data. Quantizing file size would provide a still weaker mitigation and is almost certainly not worth the added complexity, since the effectiveness of such a strategy varies sharply according to file and guessable string length.

3.1.7 File Length Disclosure

In many applications, data length can be used to make inferences about the nature of data even when it is encrypted. For example, many web APIs follow a well-defined protocol whereby certain types of requests and responses have specific lengths; this can allow an attacker to infer the type of transaction taking place, or with one additional layer of indirection, to infer the possible values of a variable-length parameter.

Because the data stored in OmniFileStore is highly arbitrary and mostly coalesced into a compressed database, it is unlikely that an attacker would be able to infer much useful information from database size; these attacks are more of a concern for transactional data. However, quantizing the length of data files would provide mitigation in cases where the attacker has detailed information about stored files and is able to view both the before and after states of a synchronization containing only one change which happens to result in a predictable change to compressed data size.

Coalescing data into a large database that is opaque pre-decryption also mitigates the possibility of an attacker inferring whether a stored item is a specific known document given the file size. Adding a

---

[8] http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf

random amount of padding to the end of files stored outside the database would add uncertainty to that inference.

Both of these scenarios appear very unlikely in practice, and it is unlikely that mitigating them is justified based on the risk, but mitigating them would marginally improve security.

3.1.8 Unauthorized File Overwrite

The Omni Group identified that some interfaces capable of adding third-party-originated data to a user's database could be abused to overwrite legitimate files. There are two aspects to this attack that require mitigation: attacker-caused data loss and fraudulent insertion of data.

Data loss can be prevented by ensuring that old versions of files are retained if a third-party origination method is used to overwrite them. This should be handled in application logic: any API allowing third-party-originated data should check to ensure no pre-existing file would be overwritten. Document versioning also provides other benefits, such as preventing data loss in the event a user overwrites their own data accidentally or due to the action of malware (including ransomware). Using a separate namespace to store third-party-originated data is also a strong mitigation.

Fraudulent insertion and modification of data can be handled by the use of a MAC, which is currently called for in the design documents provided. A MAC-protected metadata file indicating which files have had a MAC applied to them would prevent an attacker from overwriting encrypted and authenticated legitimate files with unencrypted and unauthenticated fraudulent files without detection.

3.1.9 Key Rollover and Forward Secrecy

One of the stated design goals was the avoidance of long-term secrets. This means that MAC keys, DEKs, and KEKs must be changed periodically. As mentioned in the discussion of entropy, NIST standards recommend these keys all be retired after at most two years, and that any data encrypted with them be re-encrypted after three years such that the key can no longer be used for decryption. We see no reason to deviate from that recommendation, though the tenure of a key can certainly be shorter.

One aspect of the threat model of particular importance to key rollover is that it makes the defense-in-depth assumption that the user's database, including the server-stored document key management file, is in the possession of an attacker. Once an attacker has the database, key tenure and rollover periods cease to be relevant because the attacker effectively prolongs them indefinitely. However, a shorter key lifetime is capable of protecting the user from continued compromise of new data consequent to a previous disclosure of encryption keys where that compromise is undetected. If a compromise is detected, keys must be rolled over immediately.

In the present design, key rollover is of necessity handled on the user's device, including key generation. iOS devices have frequently been subjected to low startup entropy attacks in the past; an example is given in 2014 by Tarjei Mandt.[9] This is not sufficient evidence to deem the platform unsuitable for generating encryption keys, but it casts sufficient aspersions on the security of the platform to warrant preferring to generate keys on OS X. However, key rollover may be performed wherever sufficient entropy is available. Another required aspect of key rollover (and indeed key use) is the ability to scrub the key from the device when that key is no longer being operated upon.

Forward secrecy refers to the concept whereby compromise of a key exchange key or secret does not result in compromise of all keys negotiated with it. This concept is of particular importance to

---

[9] http://blog.azimuthsecurity.com/2014/03/attacking-ios-7-earlyrandom-prng.html

TLS, since the concrete application of forward secrecy in that context is the maintenance of the security of previously-exchanged traffic in light of a certificate private key being stolen. It is only possible to provide forward secrecy when the data encryption key is ephemeral, such that the information used to derive it is not meaningful even given the certificate private key used to negotiate the exchange.

Derivability of the DEK from the user's KEK is a necessary property of OmniFileStore, and compromise of the KEK (e.g. disclosure of the user's passphrase and PBKDF2 parameters) is sure to result in a compromise of past data. It is not possible to make such a key ephemeral, because the user must be able to recover their own data given knowledge of their secret. A user and an attacker knowing the user's secrets are not differentiable in this case.

3.1.10 Key Encryption Key Protection

The user's key encryption key, as derived from their passphrase, is the lynchpin of the security of OmniFileStore. It is best protected by not keeping copies of it, scrubbing it from memory after each use, and ensuring users are aware that their passphrase must be kept secret. A compromise of the user's passphrase could result in a compromise of all data stored at the point in time when the passphrase was valid, if the database was also stolen.

While this is not addressed in the design document, it would be prudent to require the user's passphrase be different from any user account password they use elsewhere to access The Omni Group's services. This will limit the possibility for cross-resource password compromise.

4 Summary

In general terms, it appears that the design posed by The Omni Group for OmniFileStore encryption is reasonable and consistent with the posed threat model, provided that the constraints and considerations outlined in this report are accounted for.

Particular attention should be paid to the handling of the user's passphrase and key encryption key, the sources of entropy available on the user's devices, and the implementation constraints of CTR mode.

We recommend that an implementation review be undertaken once the software is substantially complete, in order to verify that the considerations outlined in this document are correctly accounted for, and to discover any implementation-level issues in the product before release. However, the design of the product appears to be in line with current practice.